Benchmarking gas-saving patterns in AI-generated DeFi smart contract

Andhika Nugraha Wira Pratama
and Arya Wicaksana*

Department of Informatics,
Universitas Multimedia Nusantara,
Tangerang, 15810, Banten, Indonesia
Email: andhika.nugraha@student.umn.ac.id
Email: arya.wicaksana@umn.ac.id
*Corresponding author

Abstract: Integrating artificial intelligence (AI) like the large language model (LLM) for smart contract auto-generation standardises performance and security, reduces human error, and offers accessibility for non-developers. In decentralised autonomous systems (DASs) like decentralised finance (DeFi), the ability to AI-generate smart contracts strengthens the decentralisation and automation characteristics of the applications. In order to increase the effectiveness of a smart contract's fully decentralised and autonomous development, this study benchmarks gas-saving patterns in AI-generated DeFi smart contracts. Three DeFI smart contract development scenarios: token generation (ERC-20), tokenised vault (ERC-4626), and flash loan (ERC-3156), and the state-of-the-art LLMs (Code Llama and Code Llama – Python) are explored to study the gas-saving patterns of AI-generated smart contracts. These results help optimise DeFi smart contracts created by AI regarding gas fees for the same operations.

Keywords: Blockchain; Code Llama; DeFi; decentralised finance; LLM; large language model.

Reference to this paper should be made as follows: Pratama, A.N.W. and Wicaksana, A. (2026) 'Benchmarking gas-saving patterns in AI-generated DeFi smart contract', *Int. J. Agile Systems and Management*, Vol. 19, No. 5, pp.1–26.

Biographical notes: Andhika Nugraha Wira Pratama is a System Engineer with a Bachelor's degree in Computer Science, proficient in Python programming and Unity C# game development. He has professional experience in systems engineering and software development, applying technical skills to design and implement engineering solutions. His academic and practical competencies span core computer science disciplines, software engineering practices, and interactive media development, reflecting a multidisciplinary foundation in technology and programming environments.

Arya Wicaksana is an Associate Professor in the Department of Informatics at Universitas Multimedia Nusantara, Indonesia. He holds a Master's degree in VLSI Engineering from Universiti Tunku Abdul Rahman, where he contributed to the institution's first successful ASIC multi-processor SoC design using 0.18 µm technology. His research focuses on blockchain applications and

computational intelligence, particularly decentralised autonomous systems. With prior experience as an IC Design Engineer in Malaysia and China, he combines academic and industrial expertise and is an active IEEE and ACM member.

1 Introduction

The rise of decentralised finance (DeFi) has reshaped the financial landscape, introducing a paradigm shift with the utilisation of smart contracts (Chen and Bellavitis, 2020). These self-executing contracts on blockchain networks underpin various financial services, providing a decentralised alternative to traditional banking systems (Shah et al., 2023). As DeFi gains prominence, optimising smart contract efficiency becomes paramount, particularly in addressing the persistent challenge of high transaction costs associated with gas fees (Nelaturu et al., 2021). One significant limitation of current blockchain-based DeFi systems is the heavy reliance on human developers to create smart contracts. The intricate nature of coding smart contracts demands high technical expertise, hindering broader accessibility to DeFi for individuals without programming backgrounds. This reliance on human developers creates a bottleneck in the expansion of DeFi, including the potential for human errors like inefficient codes.

The potential of AI-generated DeFi smart contracts by leveraging large language models (LLMs) could create smart contracts autonomously and optimise their functionality. LLMs like the state-of-the-art Code Llama (Rozière et al., 2023) are powerful enough to generate texts, including codes. Code Llama can be trained and fine-tuned for DeFi smart contract generation, reducing reliance on human developers and fostering a more inclusive and efficient decentralised financial ecosystem. The integration of AI technologies stands to streamline operations, increase accessibility, and contribute to the self-executing nature of smart contracts, aligning with the core principles of decentralisation within the DeFi space. The question is whether or not AI-generated smart contracts are equal to or better than human-developed ones require benchmarking. This paper focuses on benchmarking the gas fees, which contributes directly to the token economics of the applications.

Benchmarking smart contracts, particularly those involving gas-saving patterns, is crucial for human-made and AI-generated smart contracts in DeFi. Gas costs significantly impact transaction efficiency and affordability (Li et al., 2023). By comparing gas-saving patterns, developers can improve coding practices, optimise transaction costs, and enhance scalability. AI-generated smart contracts also require benchmarking to understand and mitigate their resource-intensive nature. It helps identify optimal gas-saving strategies, aligning AI-generated contracts with cost-effectiveness, sustainability, and scalability principles, contributing to the DeFi ecosystem's evolution.

This paper extends the benchmarking strategy proposed in Marchesi et al. (2019) for gas-saving patterns in AI-generated DeFi smart contracts. The state-of-the-art Code Llama is studied to auto-generate the DeFi smart contracts, both Code Llama and Code Llama – Python DeFi smart contracts. The contributions of this paper are

i Code Llama and Code Llama – Python implementation on auto-generating DeFi smart contracts: ERC-20, ERC-4626, and ERC-3156

ii benchmarking approach and test bench for auto-generated DeFi smart contracts focusing on gas-saving patterns

iii evaluation and analysis of the AI-generated DeFi smart contracts using the benchmark program.

Through a series of DeFi contract development cases, the goal is to benchmark the AI-generated smart contracts in DeFi applications: token generation (ERC-20), tokenised vaults (ERC-4626), and flash loans (ERC-3156). The benchmark aims to provide valuable insights into the gas-saving patterns of AI-generated smart contracts, aiding in optimising smart contracts for better gas consumption.

The rest of this paper is structured as follows. The Introduction section sets the background and motivation for this work. The Preliminaries section presents related works, concepts on DeFi smart contracts, Code Llama, and gas-saving patterns. The Methods section explains the dataset collection, test bench design and implementation, and testing and evaluation. The Results and Discussion section presents empirical evidence based on three DeFi tokens and discusses the findings. Finally, the Conclusion section summarises the findings and outlines future works.

2 Preliminaries

2.1 Related works

An evaluation framework was proposed by Aldweesh et al. (2018) to examine the incentives that are aligned between the execution time of smart contracts and the associated rewards for miners. Running the Python Ethereum client on a Mac specifically for benchmarking produces encouraging results. The analysis shows a significant variation, about a factor of 50, in the compensation per CPU second among the various functions found in Ethereum's well-known contracts. Contract generation is a one-time activity that can yield profits up to six times higher than regular contract operations. These differences could lead to incentives that need to be aligned, which could affect how reliable the blockchain is.

An impartial benchmark tool for Ethereum called OpBench was presented by Aldweesh et al. (2021). It offers a simple way to check whether the gas rewards for operational code (opcodes) match the CPU time used. OpBench, designed for Parity (Rust), Go-Ethereum (GoLang), and PyEthApp (Python), exposes non-uniform gas-to-CPU ratios; some opcodes show differences of up to a factor of 10. According to the study, Parity performs better than the other clients for most opcodes; however, whether Linux or Windows is preferred depends on the Ethereum client.

BCTMark is a comprehensive framework that Saingre et al. (2020) introduced to enable accurate comparisons of blockchain technologies in virtual networks. The research explores how much energy smart-contract execution uses using the novel framework, a significant energy-intensive feature of modern blockchains. The research uses measurements and modelling of Ethereum smart contract energy consumption to extract insights from tests and analyse one year's actual Ethereum transactions. The research also

clarifies how contract calls and replication can affect energy expenses, focusing on the energy consumption of Ethereum smart contracts over one year.

Li et al. (2023) proposed a new learning-based method, ExpenGas, based on evolutionary computation-based machine learning, to detect gas-expensive operation patterns through pre-trained techniques and multi-crucial data flow graphs. Testing on 21,981 smart contract files, ExpenGas achieved 83.05% accuracy and 91.96% recall, significantly better than current methods. This approach addresses the challenges of reusing methods across different patterns in smart contracts.

Marchesi et al. (2019) provided design patterns and tips to help save gas in developing Smart Contracts on Ethereum. The gas mechanism, which manages execution costs, is challenging to estimate in advance, posing risks to transaction failure. Gas, which represents real money, requires gas-saving techniques. Although most principles of sound programming and optimisation apply to Solidity, some peculiarities in the language make it more challenging to optimise code. By applying these patterns and advice, developers can solve or mitigate the challenges typical of blockchain development. Table 1 summarises the key focuses and contributions of related works and highlights the distinct approach taken in this paper.

Table 1 Summary of related works and contributions

Authors/Works	*Focus*	*Contribution*
Aldweesh et al. (2018, 2021), Li et al. (2023), Saingre et al. (2020)	Benchmarking existing, human-written smart contracts	Highlighting the performance and features of traditionally written smart contracts
Marchesi et al. (2019)	Proposing gas-saving patterns	Providing optimisation techniques for smart contracts to enhance efficiency
This paper	Benchmarking AI-generated smart contracts and incorporating gas-saving patterns	Testing the efficiency and performance of AI-generated contracts against human-written contracts, contributing to the literature by addressing AI and blockchain intersection

2.2 DeFi smart contract

Within the blockchain ecosystem, smart contracts are a revolutionary technology that offers a secure and automated method of transaction facilitation. These self-executing contracts have specified rules and circumstances that ensure parties to a transaction fulfil their obligations without intermediaries. DeFi, however, is made to displace conventional financial systems. There are no middlemen to approve transactions for DeFi applications because no banks or other organisations manage the money. It is possible to create DeFi protocols that forbid involvement and manipulation (Shah et al., 2023).

DeFi has its account, a personal wallet, specifically a digital address that can be used to store and transfer tokens and smart contracts independently. DeFi transactions also require the user to pay a fee in the form of gas to make a successful order or exchange. Customers also would make payments and exchanges by establishing a business relationship with a bank. Placing and buying an order then requires contacting the bank with the possibility of failing the process if there is an error between parties. Since DeFi works independently and requires less effort and time. Smart contracts in DeFi need to

have correct functionality based on stated requirements for DeFi applications, as changes on smart contracts are impossible to make once the smart contract is deployed on the blockchain (John et al., 2023).

Several smart contracts have already been applied to DeFi and decentralised applications (apps), with different types of smart contracts for DeFi, such as NFTs, Decentralised Exchanges (DEX), debt collection, and flash loans. Defi Smart contracts often include their tokens that users can use as proof of ownership, such as company stock, or as currency, similar to everyday coins in the form of standard ERCs (Ethereum Request for Comments) such as ERC-20, ERC-4626, and ERC-3156. These ERCs are a set of interfaces, contracts, and utilities, which are core contracts that implement the behaviour specified in each ERC.

2.2.1 ERC-20

The most well-known and extensively used efficient standard for managing tokens, which specifies how token exchanges must operate, is the ERC-20 standard, which is only used for exchangeable tokens. Tokens are value assets based on blockchain technology, and smart contracts control how they are created, destroyed, and exchanged. Tokens can stand in for convertible resources like cash, time, or business shares. ERC-20 has sets of standard functions and events, which can be seen in Tables 2 and 3 (Vogelsteller and Buterin, 2015).

Table 2 ERC-20 functions

Function	*Description*
name()	This function returns the name of the token
symbol()	The function returns the symbol of the token
decimals()	This function returns the number of decimals the token uses to get its user representation
totalSupply()	This function returns the total token supply
balanceOf(owner)	This function returns the account balance of another account with address owner
transfer(to, value)	This function transfers value amount of tokens to address to, and must fire the transfer event
transferFrom(from, to, value)	This function transfers value amount of tokens from address from to address to, and must fire the Transfer event
approve(spender, value)	This function allows spender to withdraw from the account multiple times, up to the value amount
allowance(owner, spender)	This function returns the amount which spender is still allowed to withdraw from owner

Table 3 ERC-20 events

Event	*Description*
Transfer(from, to, value)	Must trigger when tokens are transferred, including zero value transfers
Approval(owner, spender, value)	Must trigger on any successful call to approve(spender, value)

2.2.2 ERC-4626

ERC-4626, an extension of ERC-20, proposes a standard interface for token vaults. Many different contracts, e.g., lending markets, aggregators, and tokens that are intrinsically interest-bearing, can use this standard interface, which adds a variety of subtleties. Implementing a compliant and modular token vault requires navigating these possible problems. ERC-4626 has sets of standard functions and events, which can be seen in Tables 4 and 5 (Santoro et al., 2021).

Table 4 ERC-4626 functions

Function	*Description*
asset()	This function returns the address of the underlying token used for the vault for accounting, depositing, withdrawing
totalAssets()	The function returns the total amount of underlying assets held by the vault
convertToShares(assets)	This function returns the amount of shares that would be exchanged by the vault for the amount of assets provided
convertToAssets(shares)	This function returns the amount of assets that would be exchanged by the vault for the amount of shares provided
maxDeposit(receiver)	This function returns the maximum amount of underlying assets that can be deposited in a single deposit call by the receiver
previewDeposit(assets)	This function allows users to simulate the effects of their deposit at the current block
deposit(assets, receiver)	This function deposits assets of underlying tokens into the vault and grants ownership of shares to receiver
maxMint(receiver)	This function returns the maximum amount of shares that can be minted in a single mint call by the receiver
previewMint(shares)	This function allows users to simulate the effects of their mint at the current block
mint(shares, receiver)	This function mints exactly shares vault shares to receiver by depositing assets of underlying tokens
maxWithdraw(owner)	This function returns the maximum amount of underlying assets that can be withdrawn from the owner balance with a single withdraw call
previewWithdraw(assets)	This function allows users to simulate the effects of their withdrawal at the current block
withdraw(assets, receiver, owner)	This function burns shares from owner and send exactly assets token from the vault to receiver
maxRedeem(owner)	This function returns the maximum amount of shares that can be redeemed from the owner balance through a redeem call
previewRedeem(shares)	This function allows users to simulate the effects of their redemption at the current block
redeem(shares, receiver, owner)	This function redeems a specific number of shares from owner and sends assets of underlying token from the vault to receiver
totalSupply()	Returns tht teotal number of unredeemed vault shares in circulation
balanceOf(owner)	Returns the total amount of vault shares the owner currently has

Table 5 ERC-4626 events

Event	*Description*
Deposit(sender, owner, assets, shares)	Must be emitted when tokens are deposited into the vault via the mint and deposit methods
Withdraw(sender, receiver, owner, assets, share)	Must be emitted when shares are withdrawn from the vault by a depositor in the redeem or withdraw methods

2.2.3 ERC-3156

ERC-3156 is a flash loan smart contract, a transaction in which a borrower smart contract receives assets from a lender smart contract on the understanding that the assets are returned before the transaction expires, along with an optional fee. Interfaces for lenders to accept requests for flash loans and for borrowers to temporarily take over transaction execution within the lender are specified in this ERC. ERC-3156 only has sets of standard functions, as seen in Table 6 (Cañada et al., 2020).

Table 6 ERC-3156 functions

Function	*Description*
maxFlashLoan(token)	Indicates the amount of money that is available for lending
flashFee(token, amount)	Represents the cost of a particular loan and returns the fee charged for a loan of amount token
flashLoan(receiver, token, amount, data)	The process of initiating a transfer amount of token to receiver before the callback to the receiver
onFlashLoan(initiator, token, amount, fee, data)	Approval of a flash loan and returns the keccak256 hash of 'ERC3156FlashBorrower.onFlashLoan'

2.3 Code Llama

Code Llama, which is Llama 2 focused on coding, is modified via training on datasets containing only code-related content. This version includes a more extended sampling period from the code-related dataset, which leads to improved coding performance. Code Llama demonstrates competence in writing code and explaining it in written form. It can respond to commands in both written and code form. Users can propose the development of a function that generates the Fibonacci sequence, for example (Rozière et al., 2023).

The program is helpful for debugging and code completion, and it supports several popular programming languages, including Python, C++, Java, PHP, TypeScript (JavaScript), C#, and Bash. With 500 billion code tokens and code-related data, Code Llama is trained on three distinct sizes: 7B, 13B, and 34 B. The 7B and 13B base and instruct models can fill in missing parts (FIM), enabling them to augment existing code seamlessly and facilitating tasks like code completion without additional setup (Rozière et al., 2023).

A version designed specifically for Python, called Code Llama – Python, is customised using a large dataset that contains 100 billion Python tokens, acknowledging the importance of Python in the AI community and its everyday use in code generation. Every model meets different needs in terms of latency and serving. For example, the 7B model runs well with just one GPU. The 34B model produces better results and offers

better coding support. The more minor 7B and 13B models put speed first and are ideal for jobs requiring quick turnaround times, like instantaneous code completion (Rozière et al., 2023).

2.4 Gas-saving patterns

Gas-saving detection approaches in the context of smart contracts can be categorised into several strategies based on their underlying methodologies. Marchesi et al. (2019) provide the following gas-saving patterns for smart contract design.

1 *External transactions*: Patterns related to writing contracts and sending transactions from external addresses, such as JavaScript programs that use the Web3.js standard library. The design mentioned in this category is related to proxy, data contract, and event log.

2 *Storage*: Patterns related to the use of storage for long-term data storage. The design mentioned in this category is related to limiting storage, packing variables, and Booleans.

3 *Saving space*: Patterns related to memory and storage space conservation. The designs mentioned are related to variable type, data type, variable size, default value, minimising on-chain data, and external functions.

4 *Operations*: Patterns related to the gas utilised in the tasks carried out by smart contract functions. The designs mentioned are related to limiting external calls, using internal function calls, using fewer functions, libraries, short circuits, short constant strings, modifiers, redundant operations, line swapping, and writing values.

5 *Miscellaneous*: Patterns that could not fit into the earlier ones. The design mentioned in this category is related to freeing storage and optimising code.

A method of benchmarking smart contracts by Aldweesh et al. (2018) uses a local blockchain network, an Ethereum virtual machine (EVM), or a public blockchain Testnet locally to deploy the smart contracts for execution. Several elements, e.g., transaction validation overhead, signature validation overhead, and proof of work computation, could interfere if the benchmark uses Testnet.

3 Methods

3.1 Dataset collection

The data needed in this research are a collection of text inputs, an instruction of DeFi actions, and a collection of existing smart contracts and the auto-generated smart contracts based on the selected scenarios for the benchmark program to compare and evaluate. Achieving that requires a Solidity smart contract source code dataset currently functional in the Ethereum blockchain network. Hugging Face provides a dataset containing 113k slither-audited smart contracts (Rossini, 2022). The pragma solidity version used for the code generation is 0.8.0 as 0.8.x is currently the latest series of the solidity version.

Since there are massive amounts of smart contracts in the database, figuring out what a smart contract does by manually analysing the source codes one by one is not very effective and takes a lot of effort and time. The workaround is to utilise the GPT 3.5 Turbo and GPT 4.0 APIs. Texts of to-the-point natural language instructions are generated based on the smart contract's requirement, purpose, and source code. With that method, 6,003 data are gathered by processing them from index to index in a week with an average of 800 data per day, including a 0.5-second delay per index to prevent API overload or rate limit errors. The DeFi scenario chosen from the dataset can be seen in Table 7.

Table 7 DeFi smart contract development scenarios

	Attempt	
Development scenario	*Code Llama*	*Code Llama – Python*
Smart contract to create a token called 'Xian Yearn Finance' (XYF) with 18 decimals. The purpose of the smart contract is to allow the token holders to deposit, withdraw, and work with their tokens.	20	20
Smart contract to enable depositing and withdrawing tokens to/from a Balancer pool. The smart contract should implement the functions the InteractiveAdapter and ProtocolAdapter interfaces require.	20	20
Smart contract to facilitate borrowing and lending of tokens. The purpose of the contract is to provide a decentralised lending platform where users can borrow and repay tokens. It also includes functionality for buying and selling tokens at a specified rate.	20	20

3.2 Benchmarking smart contract

The static analysis approach is used to optimise the gas-saving patterns for the AI-generated DeFi smart contracts. The main benchmarking criteria for the test bench are source code completion and gas consumption. The developed benchmarking approach is a weighted scoring system that outputs average scores from scoring multiple contracts. The criteria that determine the test bench can be seen in Table 8.

Table 8 Smart contract benchmarking criteria

Criteria	*Possible Values*
Complete source code	Yes/Partial/Failed
Packing variables	Yes/No
Packing Booleans	Yes/No
Storage type	uint*/uint256/Mixed
Data type	Mapping/Array/Mixed
Variable size type	Fixed/Dynamic
Default value	Default/Initialised
Functions	Public/External/Mixed
Storage limiting	ERC storage standards
Minimise on-chain data	ERC storage standards

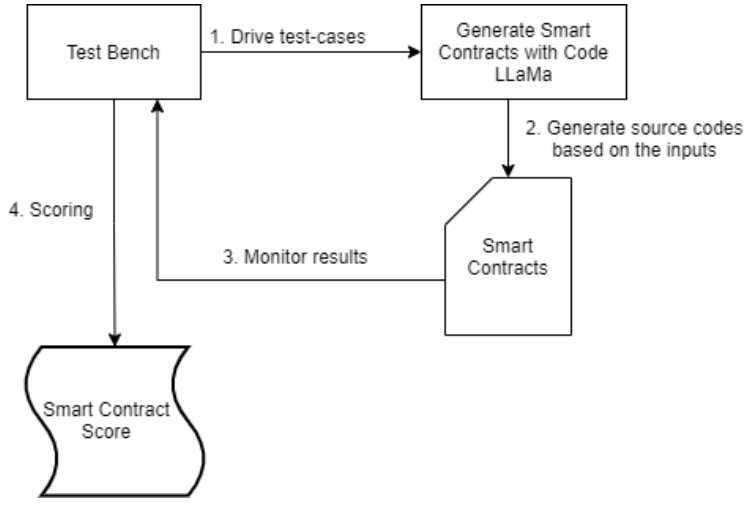
The scoring is defined by increasing the score each criterion met up to 100 (maximum). Each criterion has a default weight of 0.1 with a total of 1 sum weight, set into a variable to allow modifications. The criteria set are explained here:

- *Complete source code*: Determines whether the LLM generated a full source code. This criterion has three possible values. A fully generated smart contract (100), partially generated smart contract (50), and not generating a smart contract result in a failed generation (0).
- *Packing variables*: Variable packing is arranging variables so that more than one can fit in a single slot. 32-byte (256-bit) contiguous slots are used for storage in Solidity contracts. Assigning a uint256 before uint8 counts as an unpacked variable. This criterion has two possible values. The generated smart contract packed its variables (100) and not packed the variables (0).
- *Packing Booleans*: Boolean variables are represented as uint8 (unsigned integer of 8 bits) in Solidity. To store them, though, would only require one bit. Assigning the Booleans with a single function counts as packed Booleans. This criterion has two possible values. The generated smart contract uses packed Booleans (100) and not using packed Booleans (0).
- *Storage type*: EVM runs 256 bits at a time, so using unsigned integers smaller than 256 bits is cheaper than directly assigning uint256 (assigning uint without a number is the same as uint256). This criterion has three possible values. The generated smart contracts use small unsigned integers below 256 bits (100), used 256 bits but also used mixed unsigned integers below it (50), and every variable used only 256 bits (0).
- *Data type*: Solidity has two data types: mapping and array. Mapping data type is cheaper than an array. This criterion has three possible values. The generated smart contracts use only mapping for its data type (100), mixed data type (50), and only arrays (0).
- *Variable size type*: Fixed array size is cheaper than a dynamic array. This criterion has two possible values. The generated smart contracts used only fixed array sizes (100) and only dynamic array sizes (0).
- *Default value*: Initialising all variables upon creation is a recommended practice in software engineering. However, in Ethereum, this costs gas. This criterion has two possible values. The generated smart contracts use the default value for zero variables upon initialising (100) and initialising 0 (0).
- *Functions*: Solidity copies array arguments to memory instantaneously in public functions, but external functions can read straight from call data. While allocating memory is costly, reading from call data is cheap. This criterion has three possible values. The generated smart contracts used only external functions (100), used mixed functions (50) and only used public functions (0).
- *Storage limiting*: Since storage is the most expensive type of memory, it is best to use as little of it as possible. The criteria values are determined by the DeFi scenario smart contract's standards.

- *Minimise on-chain data*: Minimising on-chain data is beneficial as storage costs a very high amount of gas. Only the essential data should be kept on-chain; all other data should be stored off-chain. The DeFi scenario smart contract's ERC standards determine the criteria values.

Creating and evaluating the auto-generated smart contracts can be done manually. However, the risk of human error is undoubtedly high, and it would only be efficient if the creation and evaluation are done correctly. Thus, the test bench is specifically designed for this purpose, developed in Python to generate and drive the test cases. The test bench workflow can be seen in Figure 1.

Figure 1 Test bench workflow

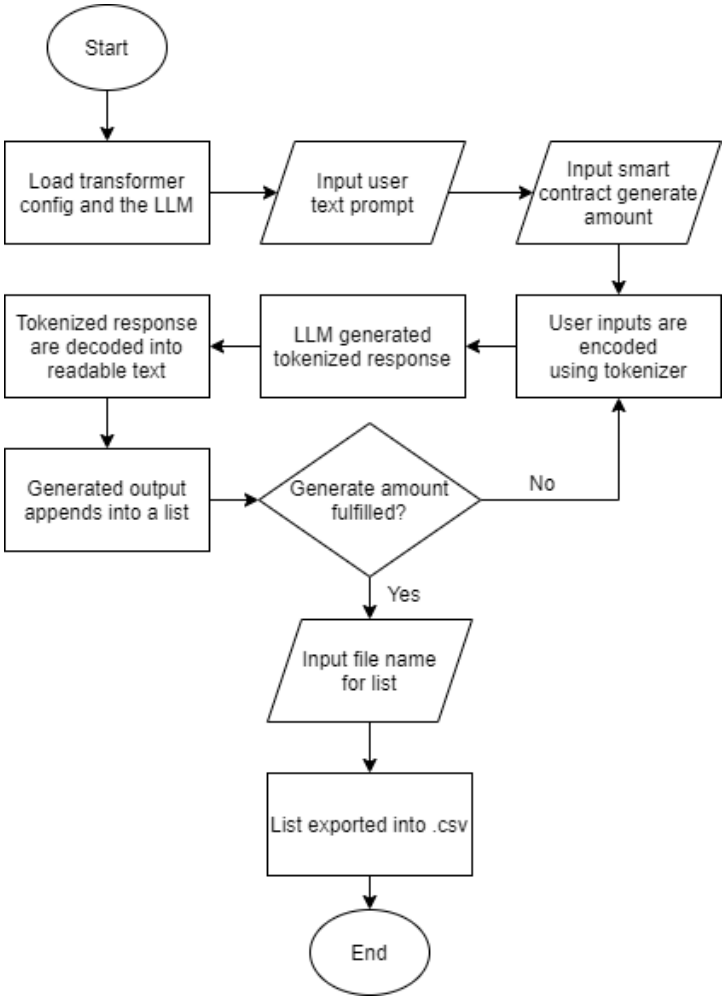


The test bench starts driving three different DeFi scenarios to the LLM. The LLM auto-generates Solidity smart contracts according to the driven input test case. The test bench monitors the resulting smart contracts and scores them. Regex is used for this purpose. Further details on the test bench workflow are broken down into two parts: driver and monitor.

3.2.1 Test bench: driver

The driver drives the test case into the LLM to generate the smart contracts. The Code Llama and Code Llama – Python could be loaded locally or through the cloud. The driver prompts the user to input the auto-generation text instruction for the model, including how many smart contracts are to be generated. The driver encodes the input into tokens using a tokeniser for the LLM to generate a response. The LLM generates a tokenised response, decoded back by the driver into readable text for the user to see. The driver continues the process until all smart contracts are generated. The generated smart contracts are put into a list and exported into a.csv file for the monitor to use later. The workflow of the driver can be seen in Figure 2.

Figure 2 Driver workflow



3.2.2 Test bench: monitor

The monitor captures the generated smart contracts and manually checks their creation status to mark them with the keywords complete, unfinished, and failed. The monitor prompts the user to choose which ERC scenario and set criteria weight for the test bench. The monitor proceeds to evaluate and score the smart contracts based on the weight for each criterion and the criteria with sets of predetermined Regex. The workflow of the monitor can be seen in Figure 3.

3.3 Model implementation

The smart contract auto-generation process utilises autoregressive generation with 7 billion parameters for both Code Llama and Code Llama – Python. The model uses Hugging Face's generation with the LLM method. The process of implementing the model is as follows:

1 Install the transformers and bitsandbytes library in the notebook.

2 Set the bitsandbytes quantisation parameters for efficient memory usage.

3 Load the selected model using the transformers library with the bits and bytes quantisation.

4 Set the tokeniser and the model generation parameters.

The bitsandbytes library is used for the quantisation configuration. The parameters are based on (Sala, 2023) fine-tuned training settings, used to auto-generate the smart contracts with Python 3 Google GPU in a memory-efficient way. The bitsandbytes parameters quantisation configuration can be seen in Table 9.

Figure 3 Monitor workflow

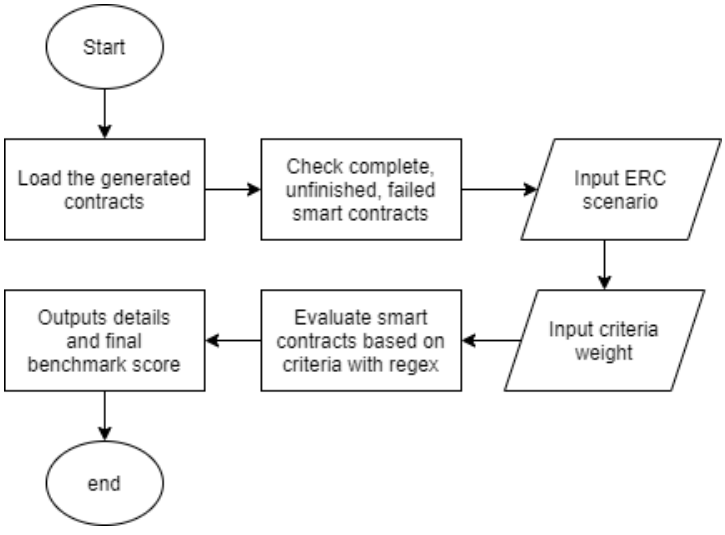


Table 9 Bitsandbytes quantisation configuration for the models

Parameter	*Value*
Use 4bit	true
bnb_4bit_compute_dtype	float16
bnb_4bit_quant_type	nf4
use_double_nested_quant	true
load_in_8bit_fp32_cpu_offload	true

After the model is loaded with transformers and is set with the bitsandbytes parameter settings, the tokeniser is used to preprocess the text inputs for the LLM to generate responses. The parameters for the tokeniser can be seen in Table 10. Different outputs are expected when generating multiple results in a single session. In order to achieve that, the mode generation parameters are set in Table 11.

Table 10 Tokeniser parameters for LLM for response generation

Parameter	*Value*
Return_tensors	pt
truncation	true

Table 11 Response generation parameters for generating different outputs

Parameter	*Value*
do_sample	true
max_new_tokens	2000
top_p	0.9
temperature	0.9
pad_token_id	tokenizer.eos_token_id

3.4 Testing and evaluation

Testing and evaluating both models use three DeFi smart contract development secenarios: token generation, tokenised vaults, and flash loans. Each scenario represents different ERC standards: ERC-20, ERC-4626, and ERC-3156. The models are expected to auto-generate smart contracts that fit the targeted standard and complete source code with full functionalities and optimised gas fees. The three scenarios for testing and evaluation can be seen in Table 12.

Table 12 DeFi scenarios for testing and evaluation purpose

No.	*Scenario*	*Expected result*
1	Create a crypto token called ‘Xian Yearn Finance’ (XYF)	Crypto token smart contract that fits ERC–20
2	Deposit and withdraw tokens to/from balancer pool	Tokenised vault smart contract that fits ERC–4626
3	Facilitate borrowing and lending of tokens	Flash loan smart contract that fits ERC–3156

The benchmarking method uses Regex, except for the first criterion: smart contract code completion. For this criterion, manual input is inserted to add a keyword to unfinished and failed DeFi smart contracts instead of implementing Regex. Then, the second until the eighth of the benchmark criteria, which are packing variable, packing Booleans, storage type, data type, variable size type, default value, and functions, use the same Regex to evaluate all DeFi scenario smart contracts. The complete source code criteria keyword can be seen in Table 13, and the set of Regex from general criteria can be seen in Table 14.

Table 13 Complete source code criteria and keyword

Criteria	*Keyword*
Complete generation	none
Unfinished generation	UNFINISHED_SC
Failed generation	NOT_A_SC

Table 14 Regex checking for all criteria

Criteria	*Regex*
Packing variable	(uint|uint(256))+ public [A-Za-z0–9]+ = [0–9]+;(\n)+\s + uint(8|16|32|128)
Packing Booleans	(uint|uint[0–9])+\([0–9]\)
Storage type	(uint|uint256) public
	uint(?!(?:256))\d + public
Data type	mapping\([^)]*\) public [A-Za-z0–9]+;
	[A-Za-z0–9]+\[\] public [A-Za-z0–9]+;
Variable size type	(uint|string|int)\[[0–9]+\]
	[A-Za-z0–9]+\[\]
Default value	public [A-Za-z0–9]+ = 0;
Functions	function [A-Za-z0–9]+\([^)]*\) external
	function [A-Za-z0–9]+\([^)]*\) public

Some criteria require only one Regex while some requires two. The Regex is further explained in the following to understand more in depth:

- *Packing variable Regex*: The Regex searches storage variables that is not packed. For instance, a uint256 or default uint is initialised before a smaller size of uint* such as uint8.
- *Packing Boolean Regex*: The Regex searches Boolean variable that uses packing Boolean function, which can be identified by searching Boolean variables with parentheses such as uint256().
- *Storage Type Regex*: The first Regex searches any storage variable that is using the default or initialised uint256, while the second one searches for smaller size uint below 256 bits.
- *Data Type Regex*: The first Regex searches for any usage of mapping for the data type, while the second regex searches for any usage of array for the data type. Variable Size Type Regex: The first Regex searches any fixed storage variables, while the second regex searches for any dynamic storage variables.
- *Default Value Regex*: The Regex searches any variable that initialised an already default value, which is zero.
- *Functions Regex*: The first Regex searches any function that uses external call, while the second Regex searches any function that uses public call.

The score is determined by how the smart contracts met the following criteria. Storage limiting and minimising on-chain Regex require a specific approach tailored to each scenario as the ERC standard-based evaluation. It determines whether the smart contract suits the ERC storage and on-chain data standard. The scoring for the ERC standard is 100 for all standard functions and events that exist. Minimising on-chain data, Regex searches for any excessive functions or events that are not the core standards of any particular ERC. The test bench sets 10 excessive data maximum is allowed. Every excessive on-chain data decreases the benchmark score by 10.

The study focuses on evaluating LLMs' performance in generating DeFi contracts rather than the efficacy of prompt engineering. While the examples suggest potential improvements through advanced prompt techniques, this remains outside the scope of the current research.

3.4.1 Scenario 1: 'Xian yearn finance' (XYF) token

In this scenario, the models must generate a smart contract that creates a token called 'Xian Yearn Finance' (XYF). The purpose of the smart contract is to allow the token holders to deposit, withdraw, and work with their tokens. The initial supply of the token is set to 1000 XYF tokens. The user prompt for this scenario can be seen in Table 15. Evaluation of the generated smart contracts for Scenario 1 uses the Regex in Table 14, and Tables 16 and 17 specifically for the ERC-4626 standards.

Table 15 Scenario 1 user input prompt

Input prompt
<<SYS>>Please write a full solidity smart contract to create a token called 'Xian Yearn Finance' (XYF). The purpose of the smart contract is to allow the token holders to deposit, withdraw, and work with their tokens. The initial supply of the token is set to 1000 XYF tokens using pragma solidity 0.8.0

Table 16 Storage limiting Regex for ERC-20

Regex
function name\([^)]*\)
function symbol\([^)]*\)
function decimal\([^)]*\)
function totalSupply\([^)]*\)
function balanceOf\([^)]*\)
function allowance\([^)]*\)
function transfer\([^)]*\)
function approve\([^)]*\)
function transferFrom\([^)]*\)
event Transfer\([^)]*\)
event Approval\([^)]*\)

Table 17 Minimise on-chain data Regex for ERC-20

Regex
function (?!totalSupply|_totalSupply|balanceOf|_balanceOf|allowance|_allowance| transfer|_transfer|approve|_approve|transferFrom|_transferFrom|decimal|_decimal| symbol|_symbol|name|_name)\w+\([^)]*\)
event (?!Approval|Transfer)\w+\([^)]*\)

3.4.2 Scenario 2: depositing and withdrawing tokens

In this scenario, the models must generate a smart contract that allow users to deposit tokens to a Balancer pool using the 'deposit' function and withdraw tokens from the Balancer pool using the 'withdraw' function. The smart contract should also provide the ability to get the balance of a token locked on the Balancer pool by a specific account using the 'getBalance' function. The user prompt for this scenario can be seen in Table 18. Evaluation of the generated smart contracts for Scenario 2 uses the Regex in Table 14, and Tables 19 and 20 specifically for the ERC-4626 standards.

Table 18 Scenario 2 user input prompt

Input prompt
<<SYS>> Please write a full solidity smart contract to enable depositing and withdrawing tokens to/from a Balancer pool. The smart contract should implement the functions required by the InteractiveAdapter and ProtocolAdapter interfaces. It should allow users to deposit tokens to a Balancer pool using the 'deposit' function and withdraw tokens from the Balancer pool using the 'withdraw' function. The smart contract should also provide the ability to get the balance of a token locked on the Balancer pool by a specific account using the 'getBalance' function using pragma solidity 0.8.0

Table 19 Storage limiting Regex for ERC-4626

Regex
function asset\([^)]*\)
function totalAssets\([^)]*\)
function convertToShares\([^)]*\)
function convertToAssets\([^)]*\)
function maxDeposit\([^)]*\)
function previewDeposit\([^)]*\)
function deposit\([^)]*\)
function maxMint\([^)]*\)
function previewMint\([^)]*\)
function mint\([^)]*\)
function maxWithdraw\([^)]*\)
function previewWithdraw\([^)]*\)
function withdraw\([^)]*\)

Table 19 Storage limiting Regex for ERC-4626 (continued)

Regex
function maxRedeem\([^)]*\)
function previewRedeem\([^)]*\)
function redeem\([^)]*\)
event Deposit\([^)]*\)
event Withdraw\([^)]*\)

Table 20 Minimise on-chain data Regex for ERC-4626

Regex
function (?!asset|totalAssets|convertToShares|convertToAssets|maxDeposit|previewDeposit|deposit|maxMint|previewMint|mint|maxWithdraw|previewWithdraw|withdraw|maxRedeem|previewRedeem|redeem)\w+\([^)]*\)
event (?!Deposit|Withdraw)\w+\([^)]*\)

3.4.3 Scenario 3: lending and borrowing tokens

In this scenario, the models must generate a smart contract that provide a decentralised lending platform where users can borrow and repay tokens. It also includes functionality for buying and selling tokens at a specified rate. The contract has governance features to allow for the management of the platform. The contract ensures the safety of token transfers through safe transfer functions. The user prompt for this scenario can be seen in Table 21. Evaluation of the generated smart contracts for Scenario 3 uses the Regex in Table 14, and Tables 22 and 23 specifically for the ERC-3156 standards.

Table 21 Scenario 3 user input prompt

Input prompt
<<SYS>> Please write a full solidity smart contract to enable depositing and withdrawing tokens to/from a Balancer pool. The smart contract should implement the functions required by the InteractiveAdapter and ProtocolAdapter interfaces. It should allow users to deposit tokens to a Balancer pool using the 'deposit' function and withdraw tokens from the Balancer pool using the 'withdraw' function. The smart contract should also provide the ability to get the balance of a token locked on the Balancer pool by a specific account using the 'getBalance' function using pragma solidity 0.8.0

Table 22 Storage limiting Regex for ERC-3156

Regex
function maxFlashLoan\([^)]*\)
function flashFee\([^)]*\)
function flashLoan\([^)]*\)
function onFlashLoan\([^)]*\)

Table 23 Minimise on-chain data Regex for ERC-3156

Regex
function (?!maxFlashLoan|flashFee|flashLoan|onFlashLoan)\w+\([^)]*\)
event [A-Za-z0–9]+\([^)]*\)

4 Results and discussion

The results show that both Code Llama and Code Llama – Python models can generate smart contracts from three different DeFi scenarios. The DeFi smart contracts were also successfully benchmarked with an equal criterion weight set. Scenario 1 uses the gas-saving pattern criteria with the additional ERC-20 smart contract standards, as seen in Tables 24 and 25.

Table 24 Scenario 1: Code Llama benchmarking details

Complete Source Code	Complete: 18	Partial: 2	Failed: 0
Packing Variable	Yes: 19	No: 1	
Packing Boolean	Yes: 0	No: 20	
Storage Type	Full uint*: 0	Mixed: 17	Full uint256: 3
Data Type	Full mapping: 11	Mixed: 1	Full array: 1
Variable Size Type	Fixed: 0	Dynamic: 20	
Default Value	Default: 18	Initialised: 2	
Functions	Only external: 5	Mixed: 2	Only public: 12
Storage Limiting		1	
Minimise On-Chain Data		0	

Table 25 Scenario 1: Code Llama – Python benchmarking details

Complete Source Code	Complete: 17	Partial: 3	Failed: 0
Packing Variable	Yes: 20	No: 0	
Packing Boolean	Yes: 0	No: 20	
Storage Type	Full uint*: 0	Mixed: 13	Full uint256: 4
Data Type	Full mapping: 4	Mixed: 0	Full array: 0
Variable Size Type	Fixed: 0	Dynamic: 20	
Default Value	Default: 18	Initialised: 2	
Functions	Only external: 6	Mixed: 2	Only public: 10
Storage Limiting		0	
Minimise On-Chain Data		7	

In Scenario 1 results, it is shown that the models successfully generated 40 outputs, 20 each, where the Code Llama model took 51 min and 30 s in total. In contrast, Code Llama – Python took 30 min and 45 s to generate 20 DeFi smart contracts. Code Llama model performs better than the Code Llama – Python model based on the benchmarking

score. Both models can generate completed source codes with fully packed variables and mostly did not initialise default values. However, neither of the models packed the Booleans, used no fixed, variable size type, public callout for the functions, or mixed bit sizes for the storage type. Despite many completed source codes, Code Llama – Python model smart contracts mostly appear to use something other than data types since only 4 out of 20 smart contracts use mapping.

In contrast, the Code Llama model has many mapping data types. Only one smart contract is generated by the Code Llama model, which fits the ERC-20 storage standard. Code Llama also managed to generate the smart contracts without exceeding the maximum on-chain data, while the other has 7 out of 20 exceeding.

Scenario 2 uses the gas saving pattern criteria with additional of ERC-4626 smart contract standards, which the results can be seen from Tables 26 and 27.

Table 26 Scenario 2: Code Llama benchmarking details

Complete Source Code	Complete: 7	Partial: 1	Failed: 12
Packing Variable	Yes: 20	No: 0	
Packing Boolean	Yes: 0	No: 20	
Storage Type	Full uint*: 0	Mixed: 8	Full uint256: 3
Data Type	Full mapping: 0	Mixed: 0	Full array: 0
Variable Size Type	Fixed: 0	Dynamic: 20	
Default Value	Default: 20	Initialised: 0	
Functions	Only external: 6	Mixed: 2	Only public: 12
Storage Limiting		0	
Minimise On-Chain Data		4	

Table 27 Scenario 2: Code Llama – python benchmarking details

Complete Source Code	Complete: 11	Partial: 0	Failed: 9
Packing Variable	Yes: 20	No: 0	
Packing Boolean	Yes: 0	No: 20	
Storage Type	Full uint*: 0	Mixed: 9	Full uint256: 2
Data Type	Full mapping: 1	Mixed: 0	Full array: 0
Variable Size Type	Fixed: 0	Dynamic: 20	
Default Value	Default: 20	Initialised: 0	
Functions	Only external: 5	Mixed: 3	Only public: 3
Storage Limiting		0	
Minimise On-Chain Data		0	

In Scenario 2 results, it is shown that the models successfully generated 40 outputs, 20 each, where the Code Llama model took 53 min and 30 s in total. In contrast, Code Llama – Python surprisingly took 9 min and 36 s to generate 20 smart contracts, which is ridiculously faster than the other model. Despite having a low score in this scenario, the Code Llama – Python model performs slightly better than the Code Llama model based on the benchmarking score. Both models show an equal number of generating complete

source code, but most failed. However, both models generated all smart contracts with packed variables and default values. However, neither model packed their Booleans, and all the smart contracts used dynamic variable size types, the same as Scenario 1. Almost all smart contracts did not specify their data type; only one of the smart contracts from Code Llama – Python model manages to generate mapping only. Code Llama model generates more smart contracts using external callouts for its functions, while the other model generates an equal number of functions. Both of the models partially met the ERC-4626 standards. Code Llama – Python also auto-generated the smart contracts with no exceeding the maximum on-chain data, while the other has 4 out of 20 that exceed.

Scenario 3 uses the gas saving pattern criteria with additional of ERC-3156 smart contract standards, which the results can be seen from Tables 28 and 29. The results shown that the models successfully generated 40 outputs, with 20 each. The Code Llama model took a total of 52 min and 11 s, while Code Llama – Python took 37 min and 36 s to generate 20 smart contracts. Despite receiving a lower score in this scenario, almost the same as in Scenario 2, the Code Llama – Python model performed slightly better than the Code Llama model based on the benchmarking score. Code Llama – Python demonstrated its ability to generate most of the complete source codes, with fewer unfinished or failed smart contracts, whereas the other model produced mostly unfinished contracts. Both models generated all smart contracts with packed variables, initialising only one default value. However, neither model packed the Booleans, and all smart contracts used dynamic variable size types, as seen in Scenarios 1 and 2. Both models were able to generate most smart contracts with mixed uint bit sizes. In addition, the Code Llama – Python model's smart contracts mostly did not use data types, with only 4 out of 20 contracts using mappings, similar to Scenario 1.

Table 28 Scenario 3: Code Llama benchmarking details

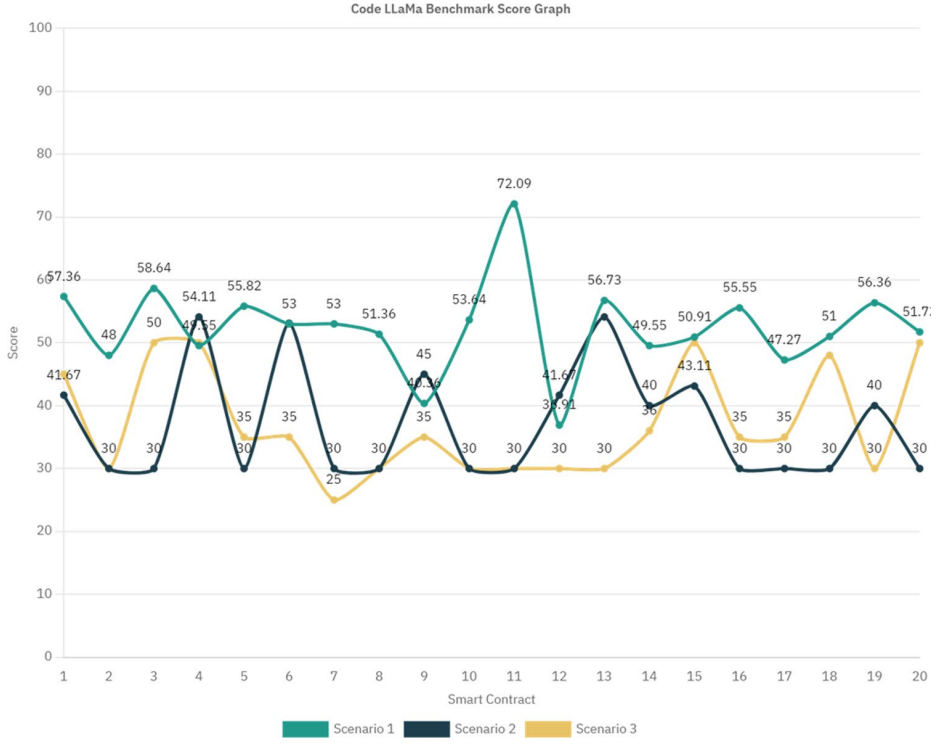
Complete Source Code	Complete: 6	Partial: 9	Failed: 5
Packing Variable	Yes: 20	No: 0	
Packing Boolean	Yes: 0	No: 20	
Storage Type	Full uint*: 0	Mixed: 16	Full uint256: 0
Data Type	Full mapping: 6	Mixed: 2	Full array: 2
Variable Size Type	Fixed: 0	Dynamic: 20	
Default Value	Default: 19	Initialised: 1	
Functions	Only external: 2	Mixed: 2	Only public: 8
Storage Limiting		0	
Minimise On-Chain Data		11	

In contrast, the Code Llama model has slightly more mapping data types with a couple of mixed arrays. Both models generate more public function callouts than external ones; some are mixed. Both models struggle to create smart contracts without exceeding the maximum on-chain data allowed, as Code Llama's 11 out of 20 contracts exceed the limit and Code Llama – Python's 8 out of 20 exceed the limit. The whole Code Llama and Code Llama – Python individual benchmarking score of the DeFi scenarios is in a form of a graph, which can be seen in Figures 4 and 5.

Table 29 Scenario 3: Code Llama – python benchmarking details

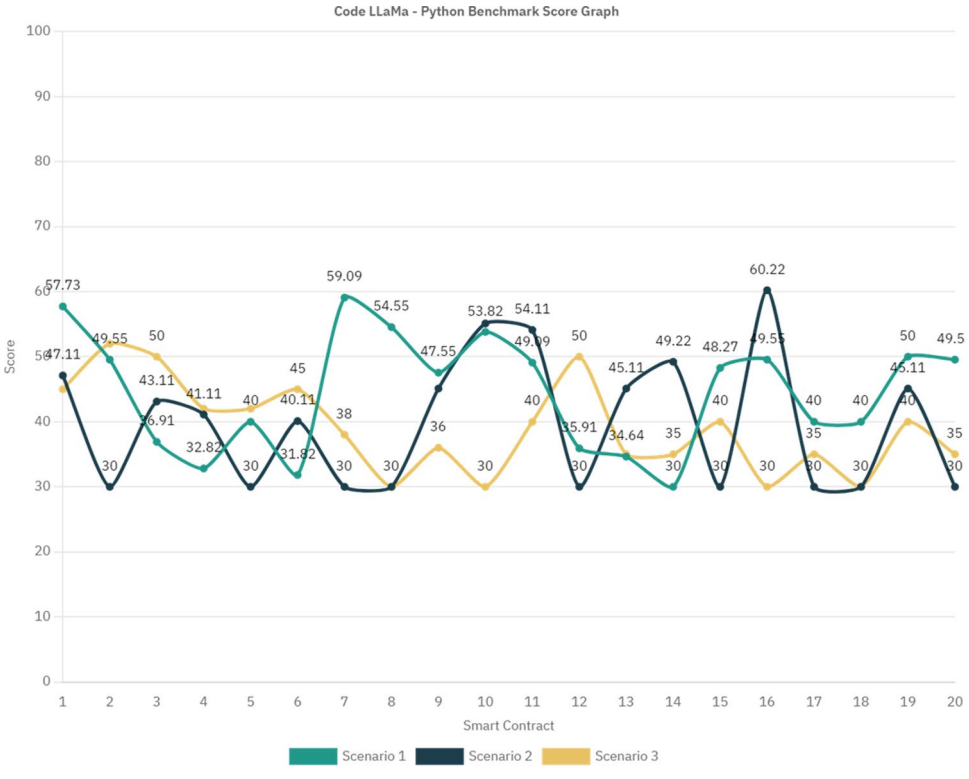
Complete Source Code	Complete: 13	Partial: 4	Failed: 3
Packing Variable	Yes: 20	No: 0	
Packing Boolean	Yes: 0	No: 20	
Storage Type	Full uint*: 0	Mixed: 16	Full uint256: 0
Data Type	Full mapping: 4	Mixed: 0	Full array: 0
Variable Size Type	Fixed: 0	Dynamic: 20	
Default Value	Default: 19	Initialised: 1	
Functions	Only external: 2	Mixed: 4	Only public: 8
Storage Limiting		0	
Minimise On-Chain Data		8	

Figure 4 Code Llama benchmarking score (see online version for colours)



Based on the testing and evaluation using the three DeFi smart contract development scenarios, both state-of-the-art models of Code Llama and Code Llama – Python are useful in generating smart contracts. However, in most cases, the models are unable to generate a complete smart contract code for the targeted application. However, fine-tuning the models could improve their ability to produce a complete smart contract code conforming to the targeted ERC standard. Using the models without any fine-tuning in this study yields a success rate of 90% for Code Llama and 85% for Code Llama – Python for smart contract code completion.

Figure 5 Code Llama – Python benchmarking score (see online version for colours)



From the perspective of gas-saving patterns, the results across the three scenarios show that the generated smart contracts from both models do not meet the expected gas-saving standards. Several factors influence the benchmarking scores, as outlined below:

1 The test bench heavily relies on assertions using Regex, which is limited in detecting more than two items. Alternative methods, such as string matching with built-in functions or natural language processing tools, could enhance its versatility.

2 The benchmark system uses a weighted scoring approach with a total weight of 1, assigning equal weights (0.1) to each of the 10 criteria. Adjusting these default values could yield different scores based on specific needs.

3 The failure of both models to fully generate smart contract code in all three scenarios negatively impacts the benchmarking results. Fine-tuning the models for DeFi applications could address code completion issues and improve scores.

4 All generated smart contracts use dynamic variable sizes and omit packing for Booleans, which leads to lower benchmarking scores.

5 Faulty smart contracts often result in infinite function or variable generation, infinite random texts, or repeated input prompts. This affects several criteria, including storage type, data type, functions, storage limits, and minimising on-chain data.

6 Function names in smart contracts vary widely, meaning most do not adhere to ERC standards, impacting benchmarking results.

Several factors should be carefully considered in order to enhance the performance of LLMs in generating smart contracts:

1 Due to hardware limitations, both models use a 7-billion-parameter configuration, which may restrict performance in generating smart contracts.

2 The input prompt may not be specific enough for auto-generating the contracts, contributing to the high time consumption and incomplete or failed smart contracts.

3 ERC-4626 and ERC-3156 are less commonly used than ERC-20, and ERC-4626's extensive core function standards may have contributed to the models' inability to generate accurate contracts for that scenario.

4 The quality of the generated code is closely tied to the quality of the prompts. The results indicate that advanced prompt engineering techniques could significantly improve outcomes.

5 Given the task's specific nature – generating DeFi contracts from known smart contract patterns – the generated code may rely on memorisation. This limitation arises as the code generation is largely constrained to patterns found in the training dataset.

5 Conclusion

Benchmarking gas-saving patterns is important in developing smart contracts, which is useful not only for AI-generated smart contracts but also for human-developed ones. The test bench's application could be widened for decentralised autonomous systems (DASs) other than decentralised finance (DeFi). The state-of-the-art Code Llama models used here are not fine-tuned to demonstrate the initial readiness and availability of the technology for smart contract auto-generation. Decentralisation and automation are two main characteristics of DeFi, making AI-generated smart contracts promising with potential contributions like gas optimisation.

The static analysis approach used in the test bench, including the design and scoring, is the extension of previous work in the field to optimise the gas of smart contract design. Regex's use for assertions is limited, and improvement with other approaches like machine learning is required to conduct non-assertion checking. The results show that optimising gas-saving patterns in smart contracts is not a straightforward, simple problem. Different token standards pose different difficulties in optimising the gas, not to mention the approaches taken to optimise the gas.

Direct use of Code Llama models without any fine-tuning and limited hardware processing capabilities is to simulate scenarios in DeFi and most decentralised application development environments, where resources and accessibility are limited. Future work on improving the benchmark involves approaches other than Regex and a combination with other methodologies, e.g., dynamic analysis, machine learning, and gas profiling.

5.1 Practical implications for industry

Here are the practical implications of AI-generated smart contracts for the DeFi industry.

1 *Cost efficiency*: AI models like Code Llama, even in their current state, show the potential to automate smart contract generation, thereby reducing development time and costs. In the future, as AI models become more capable, they could significantly lower the need for manual coding and debugging, providing a more cost-effective alternative for DeFi projects. This would be particularly advantageous for smaller startups that lack the resources to employ specialised blockchain developers.

2 *Speed and scalability*: AI-generated contracts can be created much faster than traditional development methods. For instance, in Scenario 2, Code Llama – Python was able to generate contracts in a fraction of the time taken by the other model. This speed could accelerate the deployment of new decentralised applications (dApps), helping DeFi platforms to quickly iterate and scale their offerings. Moreover, optimised AI models could handle more complex scenarios and larger datasets more efficiently, further enhancing scalability.

3 *Wider adoption*: As AI-generated contracts improve, they could become a standard tool in DeFi development. Smaller projects and non-technical users could take advantage of these AI models to deploy smart contracts without needing to understand intricate programming. The automation of smart contract creation could increase trust in DeFi platforms, making them more accessible to a wider range of users and contributing to the adoption of decentralised finance.

4 *Gas optimisation*: One of the key challenges in DeFi is gas efficiency, which directly impacts transaction costs. While the current models were not fully optimised for gas-saving patterns, the potential for AI to generate gas-efficient contracts is clear. By refining the models to optimise gas usage and adhering to standards like ERC-20, ERC-4626, and ERC-3156, AI-generated contracts could significantly reduce transaction costs, making DeFi applications more attractive to users.

5 Reducing human error: As smart contract generation becomes more automated, human errors that commonly occur in manual coding could be reduced. This could enhance the security and reliability of smart contracts in DeFi, leading to fewer vulnerabilities and failures. Over time, the quality of AI-generated contracts could approach that of human-written contracts, if not surpassing them, particularly with fine-tuning and continuous model improvements.

References

Aldweesh, A., Alharby, M., Mehrnezhad, M. and van Moorsel, A. (2021) 'The OpBench ethereum opcode benchmark framework: design, implementation, validation and experiments', *Performance Evaluation*, Vol. 146, p.102168, https://doi.org/10.1016/j.peva.2020.102168

Aldweesh, A., Alharby, M., Solaiman, E. and van Moorsel, A. (2018) 'Performance benchmarking of smart contracts to assess miner incentives in ethereum', *2018 14th European Dependable Computing Conference (EDCC)*, https://www.researchgate.net/publication/328908738_Performance_Benchmarking_of_Smart_Contracts_to_Assess_Miner_Incentives_in_Ethereum

Cañada, A.C., Kobayashi, F., Fubuloubu, and Williams, A. (2020) *ERC-3156: Flash Loans*, Ethereum Improvement Proposals, https://eips.ethereum.org/EIPS/eip-3156

Chen, Y. and Bellavitis, C. (2020) 'Blockchain disruption and decentralized finance: the rise of decentralized business models', *Journal of Business Venturing Insights*, Vol. 13, p.e00151, https://doi.org/10.1016/j.jbvi.2019.e00151

John, K., Kogan, L. and Saleh, F. (2023) 'Smart contracts and decentralized finance', *Annual Review of Financial Economics*, Vol. 15, No. 1, pp.523–542, https://doi.org/10.1146/annurev-financial-110921-022806

Li, J., Zhao, Z., Su, Z. and Meng, W. (2023) 'Gas-expensive patterns detection to optimize smart contracts', *Applied Soft Computing*, Vol. 145, p.110542, https://doi.org/10.1016/j.asoc.2023.110542

Marchesi, L., Marchesi, M., Destefanis, G., Barabino, G., Tigano, D., Zhang, S., Lee, J-H., Wright, A.J., Jain, S.M., Feist, J., Grieco, G. and Groce, A. (2019) 'Design patterns for gas optimization in ethereum', *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, Vol. 7, pp.9–15, https://doi.org/10.1109/IWBOSE50093.2020.9050163

Nelaturu, K., Beillahi, S.M., Long, F. and Veneris, A. (2021) 'Smart contracts refinement for gas optimization', *2021 3rd Conference on Blockchain Research and Applications for Innovative Networks and Services (BRAINS)*, pp.229–236, https://doi.org/10.1109/BRAINS52497.2021.9569819

Rossini, M. (2022) *Slither Audited Smart Contracts Dataset*, https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C.C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J. and Synnaeve, G. (2023) *Code Llama: Open Foundation Models for Code*, https://arxiv.org/abs/2308.12950

Saingre, D., Ledoux, T. and Menaud, J-M. (2020) 'BCTMark: A framework for benchmarking blockchain technologies', *2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA)*, pp.1–8, https://doi.org/10.1109/AICCSA50499.2020.9316536

Sala, E.M. (2023) *LlaMa 2 7b 4-bit Python Coder*, https://huggingface.co/edumunozsala/llama-2-7b-int4-python-code-20k

Santoro, J., T11s, Jadeja, J., Cañada, A.C. and Doggo, S. (2021) *ERC-4626: Tokenized Vaults*, Ethereum Improvement Proposals, https://eips.ethereum.org/EIPS/eip-4626

Shah, K., Lathiya, D., Lukhi, N., Parmar, K. and Sanghvi, H. (2023) 'A systematic review of decentralized finance protocols', *International Journal of Intelligent Networks*, Vol. 4, pp.171–181, https://doi.org/10.1016/j.ijin.2023.07.002

Vogelsteller, F. and Buterin, V. (2015) *ERC-20: Token Standard*, Ethereum Improvement Proposals, https://eips.ethereum.org/EIPS/eip-20