LLM-based Generation of Solidity Smart Contracts from System Requirements in Natural Language: the AstraKode Case

Gabriele De Vito*, Damiano D'Amici†, Fabiano Izzo†, Filomena Ferrucci*, Dario Di Nucci*
*Software Engineering (SeSa) Lab – University of Salerno, Salerno, Italy
†AstraKode S.r.l., Pescara, Italy
gadevito@unisa.it, damiano.damici@astrakode.tech,
fabiano.izzo@astrakode.tech, fferrucci@unisa.it, ddinucci@unisa.it

***Abstract*—As blockchain technology continues to evolve, the need for accessible solutions for developing smart contracts has grown, especially for non-technical users. This paper addresses practitioners' challenges in generating Solidity smart contracts from natural language requirements within the AstraKode Blockchain no-code platform (AKB). Our goal is to lower the barrier of entry into smart contract development, making it more accessible to users with limited technical expertise. We propose three methods, i.e., Naive Generation, Augmented Generation, and Enhanced Generation, each utilizing large language models to streamline the code generation process. These methods cater to different user needs, from rapid prototyping to handling complex business scenarios, improving accessibility and usability within AKB. We demonstrate their practical relevance, potential, and limitations in addressing real-world challenges in smart contract development through empirical evaluations and practitioner feedback. Thanks to collaboration with academia and effective knowledge transfer, these methods provide innovative solutions to the challenges of smart contract generation. Furthermore, they have been integrated into AKB to enhance user services, ultimately promoting the development and deployment of secure and efficient smart contracts in the industry.**

***Index Terms*—Smart Contracts, No-Code Platform, Code Generation, Large Language Model**

I. Introduction

Smart contracts are self-executing contracts with the terms of the agreement directly written into code. They enable decentralized and automated transactions across various industries, significantly enhancing efficiency and trust [1], [2]. The adoption of blockchain technology and smart contracts has rapidly grown in sectors such as finance, supply chain management, healthcare, and digital identity verification [3]. However, the complexity and technical expertise required to develop smart contracts can create barriers for non-technical users, limiting their ability to leverage these innovative solutions [3]. No-code platforms have emerged as a promising solution to democratize access to blockchain technology, enabling users with limited technical skills to create smart contracts without extensive programming knowledge [3], [4]. These platforms provide intuitive interfaces and tools that simplify the development process [3]. However, existing no-code platforms often have limitations, such as reduced flexibility and technical precision, emphasizing the need for improved solutions to meet users' diverse needs better [3].

Recent advancements in large language models (LLMs) have shown promise in automating code generation from natural language descriptions. These models can bridge the gap between non-technical users and smart contract development by generating high-quality code snippets [5], [6]. However, the limited research on automatic Solidity smart contract generation [7], [8] underscores the need for further exploration of LLMs' potential in this field.

The AstraKode Blockchain (AKB) platform aims to leverage LLMs to enhance its model-driven approach to enterprise blockchain development. The platform offers unique features, including visual development environments like the Network Composer for designing blockchain networks and the Smart Contract IDE for creating and deploying smart contracts. However, despite these advanced features, there remain challenges in enabling non-technical users to effectively generate Solidity smart contracts from natural language requirements, which can hinder the platform's accessibility and usability.

To address these challenges, this paper introduces three methods to generate Solidity smart contracts from natural language requirements: *Naive Generation*, *Augmented Generation*, and *Enhanced Generation*. *Naive Generation* focuses on rapid prototyping of smart contracts, addressing industry challenges related to speed and accessibility for users with limited technical expertise. *Augmented Generation* ensures contract completeness and correctness, which are critical for enterprise applications. This method balances speed, completeness, and correctness, making it suitable for scenarios where quality is paramount. *Enhanced Generation* is designed for complex business scenarios within the AKB platform, providing a structured approach to generating comprehensive

FINANZIATO CON IL CONTRIBUTO DELLA REGIONE ABRUZZO A VALERE SUL PR FESR ABRUZZO 2021-2027 Obiettivo di Policy 1 – Un'Europa più Competitiva e Intelligente Obiettivo Specifico 1.1 - Sviluppare e rafforzare le capacità di ricerca e di innovazione e l'introduzione di tecnologie avanzate Azione 1.1.1. Sostegno a processi di ricerca, innovazione, sviluppo e trasferimento Avviso Pubblico Intervento 1.1.1.1: "Sostegno a progetti di Ricerca e Innovazione delle imprese afferenti ai Domini tecnologici della Strategia Regionale di Specializzazione Intelligente RIS3 Abruzzo 21-27". Codice Unico Progetto: C29J24000070007

smart contracts with detailed specifications. Collectively, these methods enhance the capabilities of the AKB platform by offering flexible solutions tailored to different user needs and application contexts.

We performed an empirical evaluation to assess the effectiveness and efficiency of the proposed methods. Our empirical evaluation demonstrated that they effectively generate Solidity smart contracts from natural language requirements. The Enhanced Generation method stood out for its close alignment with the ground truth in terms of code length, complexity, and maintainability, making it ideal for complex scenarios. Practitioner feedback also highlighted its strengths in completeness and adherence to requirements.

This paper's key contributions include the development of innovative methods that address the challenges of smart contract generation and provide practical relevance for industry practitioners. By leveraging LLMs, these methods aim to streamline the code generation process and reduce barriers for non-technical users.

Structure of the paper. The remainder of this paper is organized as follows: Section II provides background and related work; Section III details the proposed methods; Section IV presents the design of the empirical evaluation; Section V discusses the results and implications; Section VI reports the threats to validity; and Section VII concludes the paper with future research directions.

II. Background and Related Work

This section covers smart contracts and no-code platforms in blockchain, highlights the Astrakode Blockchain platform's approach to simplifying development, and reviews studies on using LLMs for automating smart contract creation.

A. Smart Contracts and No-Code Platforms

Blockchain is a decentralized public ledger that records unalterable transactions in a chain of blocks, validated by network nodes. It enables trustless transactions without intermediaries like banks, using algorithms such as Proof of Work (PoW) [9] and Proof of Stake (PoS) to achieve consensus and maintain user privacy via virtual addresses [10]. Key features include decentralization, immutability, persistence, and anonymity [1], [2]. Smart contracts are a significant advancement in the evolution of blockchain technology. They are self-executing agreements encoded in software, facilitating automated, trustless interactions without intermediaries. Due to their transparency and immutability, they are suited to applications like decentralized finance, supply chain management, and digital identity verification, though their complexity can be a barrier for non-programmers. No-code platforms have emerged to address these challenges, empowering individuals with limited coding skills to design and manage smart contracts via intuitive visual interfaces using drag-and-drop features, templates, and pre-built components [3]. These platforms democratize access to blockchain technology and enable broader engagement in creating decentralized application (dApps) [3]. They reduce development time and costs, allowing for rapid prototyping without extensive programming knowledge, and include built-in validation and error-checking to mitigate flawed contracts [3]. However, they face challenges, including potential user misunderstandings, code vulnerabilities, and vendor lock-in, which can limit flexibility and control [3].

B. Astrakode Blockchain

AKB is a no-code platform that simplifies enterprise blockchain development by abstracting the complexities of blockchain networks and smart contracts. It maintains technical precision and ease of use by leveraging ontologies, metamodels, and formal grammar. The platform's engine validates models, generates necessary artifacts, and produces deployment-ready code. AKB provides two visual environments to streamline development: the Network Composer, for designing and deploying blockchain networks, and the Smart Contract IDE, for developing and deploying Solidity-based contracts. These tools support major blockchain technologies like Hyperledger Fabric and Solidity, allowing enterprises to choose the best fit for their needs. With built-in validation and automated auditing tools, AKB enables secure and stable blockchain applications for enterprise needs [11].

C. Code Generation

Code generation automates the process of converting high-level requirements into executable code, using various techniques to meet specified constraints. This section examines traditional program synthesis and recent AI-driven advancements, especially the role of LLMs in automating smart contract creation.

1) Program Synthesis and AI Approaches: Early research in code generation encompasses both deductive [12], [13] and inductive [14]–[17] approaches. The deductive synthesis method relies on the premise that a thorough and precise formal specification of the user's intent is provided. However, this requirement often proves to be more complex and demanding than the development of the program itself. In contrast, the inductive synthesis approach utilizes inductive specifications such as input/output pairs and examples. A comprehensive overview of significant advancements in program synthesis can be found in a survey by Gulwani et al. [18].

Recently, there has been a growing trend among researchers to integrate neural networks into the code generation process. Machine-learning approaches, built upon the ideas of inductive synthesis, enhance the ability to learn patterns from large datasets of code examples. For instance, Sun et al. [19] suggest a grammar-based structural convolutional neural network to manage long dependencies within code effectively. Bolin et al. [20] propose a dual learning framework that simultaneously trains both the code generation and code summarization models, aiming for improved outcomes in both areas. However, these studies acknowledge limitations such as dependency on predefined grammar and challenges with complex code structures, suggesting areas for future research.

2) LLMs Code Generation: Transformers [21] have revolutionized natural language processing, increasing interest in applying them to code generation. Contemporary investigations into code generation methodologies can be broadly classified into three distinct groups: sequence-based methods, tree-based techniques, and pre-trained models. Sequence-based approaches treat code as token sequences, using language models to generate code incrementally based on input descriptions. For example, Hashimoto et al. [22] introduced a retrieval-based model that enhances code generation by using analogous code snippets as supplemental input. Tree-based techniques create a hierarchical code representation from an input description, often called a parse tree or Abstract Syntax Tree (AST). Yin et al. [23] introduced 'Tranx,' a semantic parser using a transition-based neural model to generate actions for tree construction, which builds the AST.

Pre-trained models are trained on extensive datasets and then fine-tuned for tasks like code generation. Several variants have been proposed, such as encoder-decoder (i.e., AlphaCode [24]), encoder-only (i.e., CodeBERT [25]), and decoder-only [26]–[28]. However, it is the OpenAI GPT series, including GPT-3.5 [27] and GPT-4 [28], which are decoder-only models, that have shown the most promise in this field [5], [6]. Despite their success, variability in model outputs has raised concerns about consistency. Ouyang et al. [29] found significant differences in correctness, test outputs, syntax, and code structure from the same instruction across multiple requests in ChatGPT's code generation.

Regarding automated smart contract generation, to the best of our knowledge, only a few studies focus on using LLMs to facilitate the creation of smart contracts textual descriptions [7], [8]. Napoli et al. [7] used the CO-STAR methodology to optimize prompt creation for high-quality outputs and assess the correctness of generated contracts with Slither, a vulnerability detection tool. They tested their pipeline on a single property rental agreement, achieving a 98.1% compilability rate with minimal impact from the LLM's temperature. However, requirements must be structured to be processed, only a single contract has been tested, and the completeness of the generated code has not been evaluated, which is essential in complex scenarios requiring nuanced business logic. Karanjai et al. [8] assessed the capabilities of ChatGPT and Google PaLM2 for generating smart contract code in Solidity. Their study systematically assesses the quality of the generated code in terms of validity and correctness using a dataset of 1,349 contracts sourced from GitHub. The results indicate that PaLM2 produced incomplete code in 415 cases and non-compilable code in 519 instances. GPT-3.5 generated incomplete code in 788 cases and non-compilable code in 413 instances.

D. Research Gaps and Motivation of Our Work

Despite advancements in smart contract development and no-code platforms, critical gaps that require further research remain. First, no-code platforms, while making blockchain technology more accessible, often sacrifice flexibility and precision, limiting their use in complex scenarios. Second, current smart contract generation methods, especially those using LLMs, show promise but face completeness, correctness, and determinism issues. Indeed, the non-determinism of LLMs raises reliability concerns for enterprise applications, where errors can have serious repercussions. Moreover, existing research primarily focuses on generating simple smart contracts from textual descriptions with structured requirements, often overlooking the different users' needs and complexities and nuances required in more intricate business logic scenarios.

The AKB platform simplifies blockchain development through its model-driven approach and visual environments. However, it still faces obstacles in enabling non-technical users to generate smart contracts from natural language requirements. This gap underscores the necessity for innovative methodologies bridging the divide between high-level user requirements and technically precise smart contract code. Our work aims to address these gaps by providing tailored solutions for diverse user needs, improving the AKB platform's accessibility and usability.

III. The Proposed Methods

This section describes the methods designed to generate Solidity smart contracts, starting from natural language requirements and specifically conceived for integration with the AKB platform.

A. Prompt Engineering Best Practices

In designing our prompts, we adhered to three principal guidelines for prompt engineering, as recommended by Ekin et al. [30]. First, we ensured clarity of intent by incorporating examples, instructions, or both to demonstrate the desired outcomes. Second, we provided explicit constraints to help the LLM generate responses that adhere to the desired limitations. For instance, instructing the LLM to use specific libraries helps ensure that the generated code is relevant and usable in the intended context. Lastly, we provided high-quality data, ensuring the necessary samples to help the model discern and follow a consistent pattern. These guidelines have been proven to help the LLM produce more accurate and relevant results.

We utilized a few patterns proposed by White et al. [31]. Specifically, we employed the Persona pattern, which involves assigning a specific role or identity to the language model, such as an "expert in smart contracts and Solidity." The model is better equipped to produce relevant and authoritative responses by maintaining a consistent persona. We also used the Template pattern, which allows the user to specify a template for the output (i.e., JSON), which the LLM fills in with content.

Finally, we exploited Zero-shot Chain-of-Thought [32] to improve the LLM's ability to perform complex reasoning through intermediate reasoning steps. For instance, correcting code generated by the model might involve first identifying potential issues in the Solidity syntax, then outlining the logical flow of the contract to ensure it meets the specified requirements, and finally revising the code accordingly.

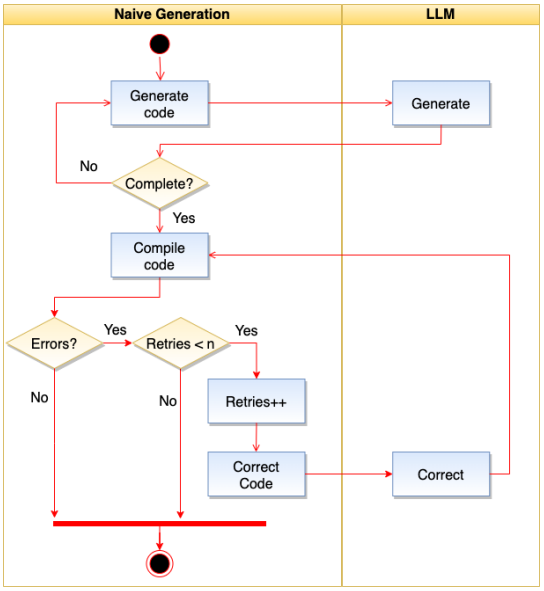

Fig. 1: Naive Generation Method.

B. Method 1: Naive Generation

The *Naive Generation* method aims to facilitate rapid prototyping of smart contracts, particularly for users who may not have extensive coding skills, bridging the gap between technical complexity and user accessibility. However, this approach inevitably sacrifices the completeness of the generated code, as the focus is on quickly generating functional prototypes rather than producing fully comprehensive code from the outset. This method follows a multi-step process to transform natural language requirements into executable Solidity code, as shown in Figure 1.

Naive Prompt Template

Act as an expert in smart contracts and Solidity. Your task is to generate the Solidity code to meet the requirements I'll provide after "REQUIREMENTS".
INSTRUCTIONS {instructions}
OUTPUT FORMAT
Answer with the solidity code, after "SOLIDITY CODE".
LOCAL LIBRARIES {localLibraries}
REQUIREMENTS {requirements}
SOLIDITY CODE:

Fig. 2: Naive Prompt Template.

The system requirements and the existing contracts within the user's project are initially fed into the LLM to generate the Solidity code, as shown in Figure 2. The inclusion of the project's libraries instructs the model to possibly reuse the project's contracts during code generation. The method incorporates an iterative "continue" approach to address the token limit constraints of the LLM, which may result in incomplete code snippets.

Correct Code Prompt Template

Act as an expert in smart contracts and Solidity. Your task is to make compilable the following code.
INSTRUCTIONS {instructions}
CODE {code}
The compiler reported the following errors.
ERRORS {errors}
The local libraries imported are reported in the following.
LOCAL LIBRARIES {localLibraries}
Let's think step by step

Fig. 3: Correct Code Prompt Template.

This approach involves feeding the incomplete code back into the LLM for further completion, allowing the model to generate additional code segments until the entire smart contract code is produced. This iterative refinement process ensures the generated code is comprehensive and accurate despite the initial truncation. After the code generation phase, the method compiles the Solidity code using the Solidity Compiler (solc). During the compilation process, any compilation errors detected by solc are identified. The method iterates by leveraging the LLM to provide corrective suggestions to resolve these errors, as shown in Figure 3.

This final step aims to enhance the quality and functionality of the code by leveraging the LLM for error correction, ensuring it meets the required standards and specifications.

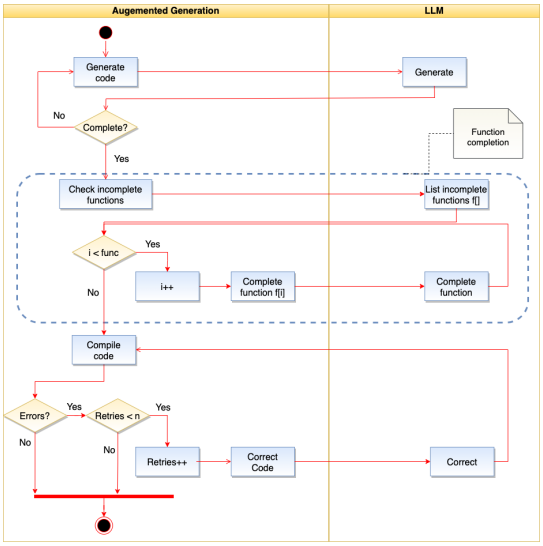

Fig. 4: Augmented Generation Method.

C. Method 2: Augmented Generation

Ensuring the completeness and correctness of smart contracts is critical, especially for enterprise applications. The *Augmented Generation* method aims to maintain prototyping speed while ensuring the final output is robust and reliable. It addresses

these concerns by building upon the *Naive Generation* approach and introducing an intermediate step designed to enhance the quality and completeness of the generated Solidity code, as illustrated in Figure 4.

Incomplete Code Detection Prompt Template

Act as an expert in smart contracts and Solidity. Your task is to check if the Solidity code I'll provide after "CODE:" is incomplete or contains placeholder logic, incomplete branches, etc.
CODE {code}
LIBRARIES {libraries}
OUTPUT FORMAT:
Identify functions with incomplete code or placeholder comments, especially those using structures like for, if, or else with placeholder logic. Provide reasons for incompleteness.
INCOMPLETE FUNCTION NAMES:

Fig. 5: Incomplete Code Detection Prompt Template.

Indeed, after the initial code generation using the *Naive method*, the *Augmented generation* approach introduces an iterative refinement step to identify and complete any placeholder logic or incomplete code segments. It leverages a specialized prompt to exploit the LLM to analyze for incompleteness or placeholder logic, as shown in Figure 5.

This prompt instructs the LLM to identify any incomplete functions or placeholder logic within the provided code. The output lists the names of incomplete functions along with motivations for why they are considered incomplete.

Complete Function Code Prompt Template

Act as an expert in smart contracts and Solidity. Your task is to complete the code I'll provide after "ORIGINAL CODE:".
INSTRUCTIONS: {instructions}
INCOMPLETE FUNCTION {incomplete_function}
LOCAL LIBRARIES {libraries}
ORIGINAL CODE {code}

Fig. 6: Complete Function Code Prompt Template.

Next, the method parallel completes the identified incomplete functions using another specialized prompt, as shown in Figure 6. This prompt guides the LLM in generating the missing logic for the incomplete functions, ensuring the code is comprehensive and executable. The iterative process involves feeding the incomplete code into the LLM, generating the required logic to complete the functions. This process continues until all incomplete segments are resolved, resulting in a fully functional and complete Solidity smart contract. As shown in Figure 3, once the code is complete, the approach proceeds to the compilation phase using solc. Like the *Naive generation* method, any compilation errors detected by solc are addressed using the LLM for corrective suggestions.

D. Method 3: Enhanced Generation

The previous methods, including the *Augmented Generation* approach, may still produce incomplete code due to output constraints inherent in LLMs. Most LLMs, including popular models like GPT-4, are limited in their output capacity, typically capping at around 4,000 tokens. This limitation poses a significant challenge when generating complex constructs such as smart contracts, which often require detailed specifications and extensive code segments to be fully realized. The issue is particularly problematic in complex business scenarios, where the completeness and correctness of smart contracts are paramount. Even minor omissions or errors in the generated code can lead to severe operational and financial repercussions.

Therefore, the *Enhanced Generation* method introduces intermediate steps that transform the system requirements into different representations of the contract definition, focusing specifically on the functions. By breaking down the contract generation process into more granular steps, we can generate more detailed and comprehensive functions, ultimately leading to a more robust and complete Solidity smart contract.

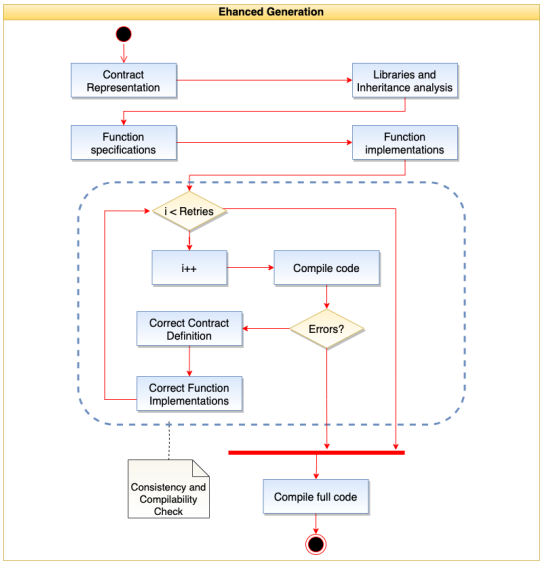

Fig. 7: Enhanced Generation Method.

As depicted in Figure 7, the method consists of several steps:

Contract Elements Generation Prompt Template

You are an expert of smart contracts. Your task is to provide the specifications needed by a Solidity Developer to develop the smart contract that meets the requirements provided by the user.
INSTRUCTIONS: {instructions}
OUTPUT FORMAT: {JSON Format }
REQUIREMENTS: {requirements}

Fig. 8: Contract Elements Generation Prompt Template.

1) **Step 1: JSON Contract Representation.** The process begins by generating a JSON representation of the

Library and Inheritance Analysis Prompt Template

Act as an expert in smart contracts and Solidity. Your task is to update the contract specifications that I'll provide to make it consistent in terms of inheritance. I'll provide the user requirements, the contract specifications, the user library, and its code.
INSTRUCTIONS: {*instructions*}
OUTPUT FORMAT: {*JSON Format*}
REQUIREMENTS: {*requirements*}
CONTRACT SPECIFICATIONS: {*contract_json*}
LIBRARY: {*library*}

UPDATED JSON OF THE CONTRACT:

Fig. 9: Library and Inheritance Analysis Prompt Template.

Detailed Function Specifications Prompt Template

You are an expert of smart contracts. Your task is to provide the detailed specifications for a function of a smart contract. The specifications must help a Solidity developer to develop the function.
INSTRUCTIONS: {*instructions*}
OUTPUT FORMAT: {*output format*}
REQUIREMENTS: {*requirements*}
CONTRACT SPECIFICATIONS: {*contract_json*}
FUNCTIONS TO IMPORT: {*functions_to_import*}
Function: {*function_json*}

SPECS:

Fig. 10: Detailed Function Specifications Prompt Template.

contract. By using a specific prompt, as shown in Figure 8, the LLM produces a JSON that includes the state variables, events, modifiers, functions, and contracts to import. This representation serves as a blueprint that identifies the general elements of the contract that need to be generated and the functions required to meet the system requirements. This step ensures a clear and organized structure for the subsequent steps.

2) **Step 2: Library and Inheritance Analysis.** Next, the method checks the contracts to import, focusing on both the project's libraries and standard libraries such as OpenZeppelin. During this step, the LLM is instructed, as shown in Figure 9, to analyze inheritance and reusability issues that may arise from the imported libraries. The contract definition obtained in Step 1 is updated to accommodate inheritance and reusability to ensure that the final contract leverages existing libraries effectively.
3) **Step 3: Detailed Function Specifications.** As shown in Figure 10, the method parallel processes the list of identified functions in the contract definition exploiting the LLM to generate detailed specifications.
   Each function specification includes a signature, including the access control requirements and ownership control, a description of the function parameters, the logic the function must follow and the events to emit, the error handling logic, the dependencies from the imported libraries, and other elements of the contract.
   The contract definition obtained in Step 1 is then updated to incorporate these detailed function specifications, ensuring a comprehensive and well-defined contract structure.
4) **Step 4: Function Implementation.** The method processes the list of function specifications in parallel to generate their implementations. At the end of this step, the contract definition is updated with the newly implemented functions, resulting in a more complete and functional contract.
5) **Step 5: Consistency and Compilability Check.** The method generates a contract skeleton (the Solidity code for the contract, including imports, state variables, events, etc., but without the function definitions) to ensure the consistency and compilability of the contract definition. The function implementations obtained in Step 4 are then added to create the overall Solidity code for the contract. The method iterates to check if the code compiles. If the code does not compile, the method uses the compilation errors to 1) parallel fix the function implementations and 2) Update the contract definition, e.g., its imports, state variables, events, and mappings, to make it consistent with the new implementations.
6) **Step 6: Final Compilation and Correction.** If the code obtained in Step 5 still does not compile, the method proceeds with the final compilation phase, similar to the first two methods. During this phase, any remaining compilation errors detected by `solc` are addressed using the LLM for corrective suggestions. This iterative refinement ensures that the final code is complete, functional, and meets the required standards and specifications.

E. LLM Choice

The proposed approaches are designed to be easily employed with different LLMs. For empirical evaluation purposes, we chose the GPT-4o model, which is accessible via the OpenAI API. We opted for GPT-4o because we expected enhanced model performance with this newer version, as noted by OpenAI[1]. This decision allows us to take advantage of the latest advancements in language modeling technology, ensuring our methods benefit from greater accuracy and robustness. For model configuration, we set the temperature parameter to a low value to minimize the likelihood of the language model generating incorrect or unpredictable outputs. A lower temperature setting makes the language model's text generation less random and more deterministic, thereby improving the reliability and consistency of the results. Additionally, Ouyang et al. [29] found that a smaller temperature setting leads to more stable code generation in LLMs. Based on these findings, we

[1]OpenAI GPT-4o. https://openai.com/index/hello-gpt-4o/

set the temperature parameter to 0, which is also recommended according to the OpenAI documentation[2].

Fig. 11: Chatbot Integration in AKB.

F. Integration with AstraKode Blockchain

Integrating the Solidity smart contract LLM generator into the AKB platform enhances the development workflow at three key stages. First, as shown in Figure 11, a chatbot feature helps users generate customized Solidity contracts based on natural language requirements during project creation, simplifying the setup process. At this stage, users can select the generation method (i.e., the proposed methods) based on their subscription and needs. Second, AKB's visual development environment allows users to refine the generated contracts and ensure they align with their intended architecture. Custom RESTful services allow users to invoke the proposed generation methods from an Angular-based UI interface. Finally, the platform parser converts generated Solidity code into visual models, enabling users to visualize and refine contracts within the platform, bridging the gap between textual and visual development.

IV. Design of the Empirical Evaluation

This section describes the empirical experiments we conducted to answer our research questions with he *goal* of investigating the effectiveness and efficiency of the proposed methods in generating Solidity smart contracts from natural language requirements. We posed the following research questions:

> ***RQ1:*** *To what extent do the proposed LLM-based methods effectively generate Solidity smart contracts from requirements expressed in natural language?*

This question aims to evaluate the effectiveness of each method in generating high-quality Solidity code.

> ***RQ2:*** *What performance trade-offs are associated with the LLM-based proposed methods for generating Solidity smart contracts?*

[2]OpenAI documentation. https://platform.openai.com/docs/api-reference

This question seeks to understand the computational efficiency and resource requirements of the methods that must balance the quality of the generated code with the performance trade-offs to ensure their practicality in industry settings.

A. Dataset

The dataset used for our experiments consists of 23 groups of smart contract requirements across various domains, such as sports, medical, tourism, gaming, DeFi, and lottery. We have a corresponding ground truth contract for each set of requirements, i.e., a manually developed Solidity contract that fulfills the specified requirements.

B. RQ_1: Effectiveness of the Methods

We employ both quantitative and qualitative assessments to comprehensively evaluate the generated contracts. First, we use the dataset described in Section IV-A and the REST services designed for each method to generate the Solidity code given the requirements. We perform five iterations for each method, given the non-deterministic nature of LLMs, and collect the following quantitative metrics:

- **Compilability (Co)** indicates whether the generated code compiles successfully without errors.
- **Tree Edit Distance (TD)** measures the structural similarity between the generated code and the ground truth.
- **Cyclomatic Complexity (CC)** assesses the complexity of the control flow within the generated code through the number of its linearly independent paths. Smart contracts with lower cyclomatic complexity are easier to understand and less risky to modify.
- **Coupling (Cp)** measures how many classes a class uses. A lower coupling is usually better.
- **Cohesion (Ch)** measures the cohesion of methods within a class, with lower values indicating higher cohesion and better modularity. In detail, we compute LCOM4, i.e., Lack of Cohesion of Methods 4.
- **Maintainability Index (MI)** is a composite metric that evaluates the ease of maintaining software. Higher values indicate better maintainability.
- **Comment Density (CD)** indicates the ratio between the total comment lines and the total lines of code.
- **LOC (LO)** counts the lines of code.
- **nSLOC (nS)** counts the lines of code without comments.

We calculate a normalized score for each code quality metric except Compilability and Tree Edit Distance. For metrics where higher is better, the score is $\frac{\text{Generated Value}}{\text{Ground Truth Value}+1}$; otherwise, the score is $\frac{\text{Ground Truth Value}+1}{\text{Generated Value}+1}$. We compute descriptive statistics, including mean, median, and standard deviation, for each metric to provide an overview of the central tendency and variability of each method's performance. Afterward, we conduct statistical tests through the ANOVA test to compare the mean achieved by all methods for each metric, helping to determine if there are statistically significant differences. ANOVA assumes independence, meaning averages for each contract must be independent. It also assumes normality of the distribution, with the Kruskal-Wallis test as an alternative if normality is a

TABLE I: Descriptive Statistics for Code Quality Metrics and Methods

	Naive			Augmented			Enhanced		
Metric	**Avg.**	**Med.**	**St. D.**	**Avg.**	**Med.**	**St. D.**	**Avg.**	**Med.**	**St. D.**
Co	0.99	1.00	0.00	0.98	1.00	0.00	**1.00**	**1.00**	**0.00**
LO	0.83	0.83	0.21	0.86	0.85	0.19	**1.22**	**1.22**	**0.32**
nS	0.84	0.84	0.21	0.88	0.88	0.19	**1.03**	**1.03**	**0.26**
CC	1.16	1.15	0.34	**1.15**	**1.13**	**0.33**	1.15	1.16	0.36
Cp	**1.01**	**1.00**	**0.34**	1.01	1.02	0.29	1.09	1.10	0.30
Ch	**1.12**	**1.11**	**0.49**	1.18	1.18	0.57	0.91	0.91	0.52
MI	**1.10**	**1.11**	**0.38**	1.08	1.08	0.34	1.03	1.03	0.36
CD	0.21	0.22	0.31	0.10	0.06	0.13	**1.94**	**1.85**	**2.27**
TD	**5.59**	**5.57**	**4.19**	5.34	5.26	3.18	7.50	7.65	3.70

TABLE II: Statistical Test Summary for Code Quality Metrics

Metric	**P-Value**	**Effect Size**	**Post-hoc**
LO	<0.001	Large	Enhanced vs. Naive/Augmented
nS	0.005	Medium	Enhanced vs. Naive/Augmented
CC	0.969	Very Small	None
Cp	0.712	Very Small	None
Ch	0.040	Medium	Naive vs. Enhanced Augmented vs. Enhanced
MI	0.012	Medium	Naive vs. Enhanced Augmented vs. Enhanced
CD	<0.001	Large	Enhanced vs. Naive/Augmented
TD	0.052	Small	None
Co	0.602	Very Small	None

concern. Lastly, homogeneity of variance is required, as verified by Levene's test, to ensure equal variances across groups. If the ANOVA reveals significant differences, a post-hoc analysis using Tukey's Honestly Significant Difference test is performed to identify which methods differ significantly. We also involve five experienced practitioners in smart contract development with at least ten years of experience to provide insightful feedback on the quality and practical applicability of the generated contracts. They review the generated contracts and provide qualitative feedback on completeness and adherence to requirements, using a Likert scale ranging from 1 (poor) to 5 (excellent). To analyze the qualitative metrics, we focus on the practitioners' feedback and compute the average rating for each method to provide a quantitative summary of their perceptions.

C. RQ_2: Efficiency and Performance Trade-offs of the Methods

We collect several metrics while executing the proposed methods and generating, as follows:

- **Total Tokens Consumed (TTC)** is the number of input and output tokens used by the LLM for code generation, reflecting resource consumption.
- **Total Compilation Retries (TTR)** is the number of recompilations required to achieve a compilable code.
- **Total Generation Time (TGT)** is the time taken in seconds to generate the code, indicating efficiency.

As described in the context of RQ1, we compute descriptive statistics and conduct statistical tests for all metrics to determine if there are significant differences in performance.

V. Results and Discussion

We present the results achieved for each research question (see our online repository [33] for details).

A. RQ_1: Effectiveness of the Methods

Table I reports the descriptive statistics for the code quality metrics (as reported in Section IV-B) and methods compared to the ground truth. The Naive Generation method achieves nearly 100% successful compilation. Contracts are concise, with LOC (mean 0.83, median 0.83, Std. Dev. 0.21) and nSLOC (mean 0.84, median 0.84, Std. Dev. 0.21) indicating consistent code length. Cyclomatic Complexity statistics (mean 1.16, median 1.15, Std. Dev. 0.34) reflect a relatively complex control flow. Coupling (mean 1.01, median 1.00, Std. Dev. 0.34) and Cohesion (mean 1.12, median 1.11, Std. Dev. 0.49) metrics show moderate interdependence and room for improved organization. The Maintainability Index values (mean 1.10, median 1.11, Std. Dev. 0.38) suggest the codebase is generally maintainable. Comment Density tendency (mean 0.21, median 0.22, Std. Dev. 0.31) shows the code contains relatively few comments. Tree Edit Distance results (mean 5.59, median 5.57, Std. Dev. 4.19) indicate moderate structural differences from the ground truth.

The Augmented Generation method improves with nearly 98% successful compilation. The LOC (means 0.86, median 0.85, Std. Dev. 0.19) and nSLOC (mean 0.88, median 0.88, Std. Dev. 0.19) values imply moderate detail. The mean Cyclomatic Complexity is 1.15 (median 1.13, Std. Dev. 0.33), suggesting stable control flow. Coupling (mean 1.01, median 1.02, Std. Dev. 0.29) and Cohesion (mean 1.18, median 1.18, Std. Dev. 0.57) tendencies indicate room for improved organization. The mean Maintainability Index is 1.08 (median 1.08, Std. Dev. 0.34), showing general maintainability. Low Comment Density values (mean 0.10, median 0.06, Std. Dev. 0.13) reflect concise documentation. The mean Tree Edit Distance is 5.34 (median 5.26, Std. Dev. 3.18), suggesting moderate structural similarity to the ground truth.

The Enhanced Generation method aligns well with the ground truth. It achieves a mean LOC of 1.22 (median 1.22, Std. Dev. 0.32) and nSLOC of 1.03 (median 1.03, Std. Dev. 0.26), capturing necessary complexity and detail. The mean Cyclomatic Complexity is 1.15 (median 1.16, Std. Dev. 0.36), indicating stable control flow. Coupling (mean 1.09, median 1.10, Std. Dev. 0.30) and Cohesion (mean 0.91, median 0.91, Std. Dev. 0.52) values show balanced organization. The mean Maintainability Index is 1.03 (median 1.03, Std. Dev. 0.36), implying general maintainability. The Comment Density tendency is high (mean 1.94, median 1.85, Std. Dev. 2.27), enhancing readability. The mean Tree Edit Distance is 7.50 (median 7.65, Std. Dev. 3.70), demonstrating reasonable structural similarity with some variability.

The statistical analysis, whose results are reported in Table II, confirms some initial findings and reveals new insights into the methods' performance. The Kruskal-Wallis test was used for all metrics due to the weak power of the Levene's test (0.43), which indicated a lack of homogeneity of variances, making ANOVA unsuitable. Significant differences in LOC, nSLOC, and Comment Density indicate that the Enhanced Generation method produces more detailed and well-documented code than the Naive and Augmented methods. In contrast, Cyclomatic Complexity and Coupling show no significant differences, suggesting similar control flow and code dependency across methods. Cohesion shows significant differences, with the Enhanced method differing from both Naive and Augmented methods, indicating variability in internal organization. The Maintainability Index reveals significant differences between the Enhanced method and both Naive and Augmented methods, though it may prioritize other aspects over maintainability. Tree Edit Distance and Compilability metrics confirm high success rates with no significant differences, maintaining consistency across all methods. Overall, all methods reliably produce compilable code. The Enhanced Generation method produces code that closely aligns with the ground truth, especially in LOC and nSLOC. It also improves Comment Density, enhancing readability. Despite significant differences in the Maintainability Index, the lower mean and median values suggest that the Enhanced method may prioritize other aspects over maintainability. No significant differences in Cyclomatic Complexity, Coupling, and Cohesion indicate similar control flow and organization across methods. Finally, Tree Edit Distance shows no significant differences, indicating structural similarity is less impactful.

Regarding qualitative metrics, practitioners found the Naive Generation valuable for rapid prototyping (with a mean of 3.9 for completeness and adherence) but limited in meeting requirements. The Augmented Generation scored 4.1, showing better accuracy for quality-sensitive scenarios. The Enhanced Generation scored highest at 4.3, proving effective for comprehensive, requirement-aligned contracts, ideal for complex scenarios. These qualitative results align with quantitative findings, confirming the advantages of the Enhanced Generation.

> **Answer to RQ_1**
>
> *The Naive Generation method is effective for rapid prototyping but may lack completeness. The Augmented Generation method improves structural similarity and control flow, aligning better with the ground truth. The Enhanced Generation method is the most effective, closely matching the ground truth in code length, complexity, and maintainability, making it ideal for complex scenarios.*

B. RQ_2: Efficiency and Performance Trade-offs of the Methods

Table III reports the descriptive statistics for the efficiency metrics reported in section Section IV-C for each method.

The Naive Generation method consumes a mean of 4,167.10 tokens (median 3,161.20, Std. Dev. 2,959.39), indicating consistent performance with some variability. It requires minimal recompilation, with a median of 0 retries, reflecting the method's efficiency in generating compilable code on the first attempt. The average generation time is 17.44 seconds (median 15.72 seconds, Std. Dev. 9.59), revealing moderate variability in execution time, which is suitable for rapid prototyping.

The Augmented Generation method consumes a mean of 14,053.98 tokens (median 12,463.80, Std. Dev. 6,616.95), showing more variability than the Naive method. It requires minimal recompilation, with a median of 0 retries. The average generation time is 73.49 seconds (median 64.31 seconds, Std. Dev. 34.44), reflecting higher computational demands due to the method's focus on enhancing code completeness.

The Enhanced Generation method consumes a mean of 79,313.64 tokens (median 38,395, Std. Dev. 111,709.06), exhibiting high resource usage and variability due to contract complexity. It requires a median of 0.20 recompilation retries, showing relative efficiency. The average generation time is 45.38 seconds (median 33.57 seconds, Std. Dev. 31.86), balancing resource use and time efficiency while generating detailed contracts. Notably, using user-defined libraries, employed in only three contracts across all methods, has a limited impact on performance (see Table IV).

Regarding the statistica tests, we performed the Kruskal-Wallis in all cases, due to the normality assumption violation for the ANOVA test. The tests revealed significant differences among methods for the efficiency metrics (see Table V). The Enhanced method consumes the most tokens and requires more retries, indicating its focus on comprehensive contract generation. The Naive method is the fastest and requires the least retries, while the Augmented method takes the longest time but balances retries with improved code quality. Post-hoc analyses confirm significant differences between the Enhanced method and the others across these metrics.

> **Answer to RQ_2**
>
> *The Naive Generation method is ideal for rapid prototyping, but it may lack completeness. The Augmented Generation method balances quality and performance, aligning better with the ground truth. The Enhanced Generation method is resource-intensive but delivers the most detailed contracts suitable for complex scenarios.*

VI. Threats to Validity

We acknowledge a few threats to the validity of our study, potentially impacting our findings.

The choice of GPT-4o could have impacted internal validity as the LLM may introduce biases inherent to this specific model. To mitigate this threat, we ensured that the prompts used for code generation were designed following best practices in prompt engineering, aiming for clarity and consistency across different tasks. Additionally, we aimed to minimize output variability by setting the temperature parameter to zero, promoting more reliable and consistent results.

TABLE III: Descriptive Statistics for Efficiency Metrics and Methods

	TTC			TTR			TGT		
Method	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**
Naive Generation	**4,167.10**	**3,161.20**	**2,959.39**	**0.10**	**0.00**	**0.35**	**17.44**	**15.72**	**9.59**
Augmented Generation	14,053.98	12,463.80	6,616.95	0.37	0.00	0.84	73.49	64.31	34.44
Enhanced Generation	79,313.64	38,395.00	111,709.06	0.88	0.20	1.23	45.38	33.57	31.86

TABLE IV: Efficiency Metrics for Contracts Using User-Defined Libraries

	Naive Generation			Augmented Generation			Enhanced Generation		
Contract	**TTC**	**TTR**	**TGT**	**TTC**	**TTR**	**TGT**	**TTC**	**TTR**	**TGT**
Game Contract	11,790.20	0.00	21.94	24,924.00	0.00	87.37	495,131.20	3.00	135.45
Delegation Manager	11,271.80	0.00	20.71	27,314.80	2.60	143.53	270,938.20	3.80	85.50
Scholarship Manager	3,511.80	0.00	15.82	12,786.80	0.20	64.31	92,722.40	1.20	55.90

TABLE V: Statistical Test Summary for Efficiency Metrics

Metric	**P-Value**	**Eff. Size**	**Post-hoc**
TTC	$<$0.001	Large	Naive vs. Enhanced Augmented vs. Enhanced
TTR	0.002	Small	Naive vs. Enhanced Augmented vs. Enhanced
TGT	$<$0.001	Large	Naive vs. Augmented/Enhanced Augmented vs. Enhanced

Construct validity could be threatened by the iterative nature of the code generation process, particularly in methods involving multiple iterations and refinements. Variability in LLM outputs across iterations could introduce inconsistencies, affecting the reproducibility of results [34]. Therefore, we conducted multiple iterations for each method and averaged the results. Additionally, by using a low-temperature setting, we aimed to enhance the determinism of the LLM outputs, supporting more consistent and reliable outcomes.

External validity is threatened by the specific context of the AKB platform, which may not fully represent other no-code platforms or environments for smart contract development. To mitigate this threat, we designed the proposed methods to be adaptable and potentially applicable to other platforms by focusing on general principles of LLM-based code generation. Future work could explore the application of these methods in different environments to further validate their generalizability.

Another external validity threat arises from the dataset used for empirical evaluation. The dataset, comprising 23 groups of smart contract requirements across various domains, may not cover the full spectrum of complexity and diversity encountered in real-world applications. To address this, we selected various domains to ensure a broad representation of typical use cases. However, future studies could benefit from expanding the dataset to include more complex and varied scenarios to validate the methods further.

The metrics used to assess the quality and performance of the generated smart contracts could threaten conclusion validity, as the employed quantitative metrics, including compilability, tree edit distance, and cyclomatic complexity, may not capture all code quality and maintainability aspects. Therefore, we complemented the quantitative analysis with qualitative feedback from experienced practitioners, providing a more holistic assessment of the generated contracts. This dual approach helps to balance the limitations of each method, offering a more nuanced understanding of the results.

VII. Conclusions and Future Work

We explored the application of LLMs to generate Solidity smart contracts from natural language requirements within the AKB platform. We focused on three distinct methods—*Naive Generation*, *Augmented Generation*, and *Enhanced Generation*—each designed to address different aspects of smart contract development, from rapid prototyping to handling complex business scenarios. The empirical evaluation demonstrated that while the *Naive Generation* method is practical for quick prototyping, it may lack completeness in more complex scenarios. The *Augmented Generation* method offers a balanced approach, improving structural similarity and control flow, making it suitable for quality-sensitive applications. The *Enhanced Generation* method emerged as the most effective but less efficient, closely aligning with ground truth regarding code length, complexity, and maintainability, and is particularly well-suited for comprehensive contract generation in intricate business contexts. Our research has significant implications for the democratization of blockchain technology. By integrating these methods into the AKB platform, we provide non-technical users with powerful tools to generate robust smart contracts, thereby lowering barriers to entry and fostering broader adoption of blockchain solutions. This work highlights LLMs' potential to bridge the gap between natural language requirements and executable smart contract code, offering a pathway for more accessible and efficient blockchain development. Based on our findings, future work could focus on several key areas. First, expanding the dataset to encompass more complex smart contract scenarios would further validate the methods and enhance their robustness. Additionally, investigating other advanced LLMs might enhance performance and accuracy. Furthermore, combining LLM-generated code with traditional verification could boost security and correctness.

REFERENCES

[1] M. Li, J. Weng *et al.*, "Crowdbc: A blockchain-based decentralized framework for crowdsourcing," *IEEE transactions on parallel and distributed systems*, vol. 30, no. 6, pp. 1251–1266, 2018.

[2] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International journal of web and grid services*, vol. 14, no. 4, pp. 352–375, 2018.

[3] S. Curty, F. Härer, and H.-G. Fill, "Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: a structured literature review," *Software and Systems Modeling*, vol. 22, no. 6, pp. 1857–1895, 2023.

[4] A. Rasti, A. A. Anda *et al.*, "Automated generation of smart contract code from legal contract specifications with symboleo2sc," *Software and Systems Modeling*, pp. 1–30, 2024.

[5] N. M. S. Surameery and M. Y. Shakor, "Use chat gpt to solve programming bugs," *International Journal of Information Technology and Computer Engineering*, no. 31, pp. 17–22, 2023.

[6] S. Jalil, S. Rafi *et al.*, "Chatgpt and software testing education: Promises & perils," in *2023 IEEE ICSTW*. IEEE, 2023, pp. 4130–4137.

[7] E. A. Napoli *et al.*, "Leveraging large language models for automatic smart contract generation," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2024, pp. 701–710.

[8] R. Karanjai, E. Li, L. Xu, and W. Shi, "Who is smarter? an empirical study of ai-based smart contract creation," in *2023 5th Conference on BRAINS*. IEEE, 2023, pp. 1–8.

[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[10] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[11] F. Izzo and D. D'Amici, "Harnessing no-code blockchain for defi: A microcredit case study on astrakode blockchain," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 433–436.

[12] C. Green, "Application of theorem proving to problem solving," in *Readings in Artificial Intelligence*. Elsevier, 1981, pp. 202–222.

[13] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, pp. 151–165, 1971.

[14] A. W. Biermann, "The inference of regular lisp programs from examples," *IEEE transactions on Systems, Man, and Cybernetics*, vol. 8, no. 8, pp. 585–600, 1978.

[15] D. E. Shaw, W. R. Swartout, and C. C. Green, "Inferring lisp programs from examples." in *IJCAI*, vol. 75, 1975, pp. 260–267.

[16] D. C. Smith, *Pygmalion: a creative programming environment.* Stanford University, 1975.

[17] P. D. Summers, "A methodology for lisp program construction from examples," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 161–175, 1977.

[18] S. Gulwani, O. Polozov *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[19] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7055–7062.

[20] B. Wei, G. Li *et al.*, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

[21] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[22] T. B. Hashimoto, K. Guu, Y. Oren, and P. S. Liang, "A retrieve-and-edit framework for predicting structured outputs," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[23] P. Yin and G. Neubig, "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation," *arXiv preprint arXiv:1810.02720*, 2018.

[24] Y. Li, D. Choi *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[25] Z. Feng, D. Guo *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[26] A. Radford, "Improving language understanding by generative pre-training," 2018.

[27] T. Brown *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[28] OpenAI, "Gpt-4 technical report," *arXiv:2303.08774*, 2023.

[29] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," *ACM Transactions on Software Engineering and Methodology*, 2023.

[30] S. Ekin, "Prompt Engineering For ChatGPT: A Quick Guide To Techniques, Tips, And Best Practices," 2023.

[31] J. White, Q. Fu *et al.*, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[32] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[33] G. De Vito, D. D'Amici, F. Izzo, F. Ferrucci, and D. Di Nucci, "Online repository: Dataset and results." November 2024. [Online]. Available: https://github.com/gadevito/AKB_RPK

[34] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.