Evaluating Capabilities and Perspectives of Generative AI Tools in Smart Contract Development

Shawal Khalid
Virginia Tech
Department of Computer Science
Blacksburg, USA
shawal@vt.edu

Chris Brown
Virginia Tech
Department of Computer Science
Blacksburg, USA
dcbrown@vt.edu

ABSTRACT

Smart contracts, self-executing agreements on the blockchain, have emerged as a transformative force in blockchain technology, automating agreements and enabling decentralized applications. However, the stakes are extraordinarily high—manual coding errors in smart contracts have repeatedly led to financial losses. Notable incidents, such as the DAO hack that resulted in a loss of approximately $50 million [36] and the Parity wallet vulnerability that froze approximately $280 million in assets [45], underscore the immense economic risks involved. To support manual development tasks, recent advancements in artificial intelligence (AI) powered by large language models (LLMs) have transformed how software is developed and maintained by automating various software engineering tasks.

This research explores the capabilities of generative AI tools for efficient and secure smart contract development. The methodology involves two phases: 1) we distribute a mixed methods survey for blockchain and smart contract developers (n = 114) to investigate their perspectives towards utilizing LLMs; and 2) we evaluate the effectiveness of generative AI tools, such as ChatGPT, Google Gemini, and ChainGPT, for smart contract development. This evaluation is based on comparing the LLM-generated smart contract code with human-written code, using a diverse dataset of smart contracts gathered from GitHub. Static analysis tools and unit testing are employed to validate the accuracy, correctness, efficiency, and security of the generated code. Our findings highlight the potential of these tools to accelerate smart contract development processes, while also emphasizing the need for human oversight, contributing to the advancement of blockchain technology and its applications.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; **Smart contracts**; • **Theory of computation** → *Blockchain.*

KEYWORDS

smart contracts, large language models, artificial intelligence, software engineering, blockchain, ChatGPT, ChainGPT, Gemini, generative AI

1 INTRODUCTION

Smart contracts are self-executing contracts with the terms of the agreement directly written into code, enabling automated and secure transactions without human intervention [52]. In various domains, smart contracts are being leveraged to automate processes, ensure compliance, and streamline operations [58]. For example, in the supply chain domain, IBM's Food Trust platform utilizes smart contracts to ensure food safety and traceability throughout the supply chain. When certain temperature or humidity conditions are not met during transportation, the smart contract can automatically trigger alerts or withhold payment until the issue is resolved [6].

However, smart contract development poses significant challenges, including coding complexity, vulnerabilities, and auditing difficulties, particularly in manual coding processes[55]. These challenges can increase the risk of vulnerabilities, security threats, and oversights, while complicating auditing and verification processes. A notable example is the 2016 DAO attack, which resulted in the theft of $60 million worth of Ether due to a vulnerability in the smart contract's code[69]. This incident highlights how even small coding oversights can lead to catastrophic financial losses, underscoring the need for rigorous smart contract development processes.

Recent advancements in artificial intelligence (AI) powered by large language models (LLMs) have transformed how software is developed and maintained through automating a variety of software engineering (SE) tasks [64]. Studies show generative AI tools powered by large language models (LLMs) are effective for supporting software development tasks such as code generation, documentation, and code translation [40]. In addition, LLM-powered tools, such as GitHub Copilot,[1] can generate code snippets or entire functions based on natural language prompts [39]. Further, adoption of these tools in SE is rapidly increasing. For example, according to the most recent StackOverflow Developer Survey, approximately 80% of developers are already using or plan to use AI-based SE assistants in their workflows [1].

While these tools are becoming indispensable in traditional software development, smart contract development presents unique challenges. Unlike traditional software, smart contracts operate on blockchain platforms, where immutability is a key feature. Once deployed, smart contracts cannot be easily altered, which makes it crucial to ensure their correctness and security upfront. This makes errors, bugs, and vulnerabilities particularly costly, as they cannot be corrected through updates or patches. Furthermore, smart contracts often handle high-value transactions and execute autonomously, increasing the risk associated with coding errors or security flaws.

The complexity and high-stakes nature of smart contract development highlight the potential value of LLMs in this domain. LLMs can assist in automating the generation of secure and precise smart contract code by integrating known best practices, potentially reducing human error during development. Prior work has explored how to leverage the capabilities of ChatGPT [3]—a state-of-the-art LLM-based chatbot—for smart contract development [48]. Our

[1] https://github.com/features/copilot

BSCI '25, August 25–29, 2025, Hanoi, Vietnam

ACM ISBN 979-8-4007-1412-2/2025/08.
https://doi.org/10.1145/3709016.3737802

project extends this work by providing a comparative analysis to evaluate the quality of across LLM-based systems and collecting insights from smart contract developers on their perceptions of utilizing LLMs for development tasks. **In particular, we aim to answer the following research questions (RQs):**

RQ1: What are the perceptions and attitudes of blockchain developers towards the utilization of generative AI tools for smart contract development?

RQ2: How effective are generative AI tools in generating smart contracts?

RQ3: What impact does customized training data for generative AI tools have on the quality and efficiency of smart contract generation?

To answer these questions, we divided our methodology into two interconnected phases. First, we conducted a sampling study [56], distributing a mixed-methods survey to gain insights from 29 blockchain developers on their experiences and perceptions of using LLMs for smart contract development. The survey results directly informed the second phase, where we explore the capabilities of LLMs for smart contract development. We investigate generated smart contracts from three LLMs—ChatGPT [3], Google Gemini [4],[2] and ChainGPT [2]. To assess LLM-generated smart contracts, **we gathered a dataset of blockchain-based projects (DApps)** hosted on GitHub to obtain publicly available smart contract files. To evaluate the effectiveness, of LLMs, we compared generated smart contracts against information recovered from existing software systems via the human-written smart contracts in our dataset. We used real-world blockchain-based static analysis tools, such as Solidity analyzer, Solhint, and the Remix Solidity analyzer, to extract details comparing the quality of LLM-generated smart contracts. Additionally, we performed unit testing to verify the functional correctness and behavior of the generated smart contracts. An overview of our methodology is presented in Figure 1.

Our findings highlight the potential of LLMs to accelerate smart contract development processes, while also emphasizing the need for human oversight and domain expertise to ensure the quality of generated code. Insights from industry practitioners on the potential use cases and challenges of integrating AI tools into smart contract development workflows show that—while developers acknowledge the potential benefits of using generative AI for tasks such as code generation, documentation, and prototyping—they expressed concerns regarding the trustworthiness and reliability of the generated code, particularly for mission-critical or high-value applications. We also demonstrate the abilities of generative AI tools for producing smart contracts, and further show that customized training data enhances the output of LLMs, in the context of smart contract development. This project contributes to the growing body of research on the application of generative AI in blockchain oriented software engineering (BOS). Additionally, we make our **DApps dataset and GitHub data available as part of a data replication package**,[3] ensuring accessibility for future researchers.

[2]formerly known as Google Bard

[3]https://github.com/shawalkhalid667/SmartContractAnalysis

2 BACKGROUND

2.1 Smart Contracts in Blockchain Technology

Smart contracts are self-executing agreements on the blockchain that automatically enforce the terms of a contract when predetermined conditions are met [57]. These contracts are stored and replicated across a decentralized network of nodes, ensuring transparency, immutability, and eliminating the need for intermediaries [13]. The Ethereum blockchain, introduced in 2015, was the first widely adopted platform for creating and deploying smart contracts, leveraging its Ethereum Virtual Machine (EVM) to execute contract code [13]. Smart contracts have found applications across various domains, including supply chain management, finance, and software licensing [10]. For instance, IBM's Food Trust platform utilizes smart contracts to ensure food safety and traceability throughout the supply chain [6]. Moreover, the Uniswap smart contracts processed about $7.17 billion per day in 2021 [12], illustrating the high stakes involved in smart contract operations. In the software engineering domain, smart contracts can facilitate software licensing agreements by automating payments based on usage or predetermined conditions [66]. The immutable and transparent nature of smart contracts, combined with their ability to automate processes and eliminate intermediaries, has made them a valuable tool for various industries seeking to streamline operations, reduce costs, and increase trust and accountability [70]. Recent empirical work has also highlighted the evolving practices and challenges faced by blockchain developers in building and maintaining secure smart contracts, pointing to the need for better tools and frameworks [28].

2.2 Generative AI and Software Engineering

Recent advancements in AI powered by LLMs have transformed software development [64]. Generative AI tools powered by LLMs have shown promising results in supporting software development tasks such as code generation, documentation, and code translation [15, 42]. Tools like GitHub Copilot,[4] powered by OpenAI's Codex model, can generate code snippets or entire functions based on natural language prompts, potentially improving developer productivity and reducing coding errors [21, 47]. Additionally, AI-powered code analysis and testing tools can help identify and mitigate vulnerabilities and bugs, improving the overall quality and security of software systems [15]. Researchers have also explored LLMs in specific domains of software development, such as fintech [67] and healthcare [18]. Complementary research has explored how social signals and community behaviors impact developer decision-making and adoption of blockchain tools, underlining the socio-technical complexity of this development landscape [30, 31]. We aim to investigate the perceptions and capabilities of LLMs in the blockchain domain, specifically for smart contract development.

2.3 Related Work

The application of generative AI tools in the domain of smart contract development remains largely unexplored, with limited research in this area. However, some studies have investigated the

[4]https://github.com/features/copilot

potential of leveraging AI and machine learning techniques for various aspects of blockchain and smart contract development. For example, Qin et al. [50] proposed a machine learning-based approach for detecting vulnerabilities in smart contracts, using a combination of static analysis and deep learning techniques—demonstrating promise in identifying potential vulnerabilities, such as reentrancy attacks and integer overflows. Similarly, Ni and colleagues implement machine learning techniques in CrashSCDet, a system to detect smart contracts that will cause runtime errors [41]. Tann et al. [59] explored the use of natural language processing (NLP) techniques to automatically generate smart contract specifications from natural language requirements. Their approach aimed to bridge the gap between non-technical stakeholders and developers, facilitating the translation of requirements into executable smart contract code.

More recently, researchers have explored the capabilities of LLMs for automatically detecting [26], classifying [35], and repairing [60] smart contract vulnerabilities, in addition to smart contract auditing [37] and formal verification [34]. While these studies have explored specific applications of LLMs in the context of smart contract development, there is still a lack of research on leveraging generative AI tools for creating smart contracts. The work most closely related to ours by Petrovic et al. proposes an automated model-driven smart contract generation approach that leverages ChatGPT [48]. Our project extends this work by: *a)* investigating the capabilities of ChatGPT for smart contract generation across a diverse domain of blockchain based projects (DApps); *b)* comparing ChatGPT smart contract generation with LLM-based systems; and *c)* collecting insights from blockchain developers on experiences, opportunities, and challenges for using LLMs in smart contract development. By investigating the perceptions and capabilities of generative AI tools, this research aims to explore their potential in addressing the challenges of smart contract development, contributing to the advancement of blockchain technology and its applications.

3 METHODOLOGY

To investigate the potential of generative AI tools for efficient and secure smart contract development, our research methodology involved two main phases: a developer survey and comparative analysis of AI tools.

3.1 Phase 1: Developer Survey

We distributed an online survey to gather insights from blockchain and smart contract developers regarding their perceptions and attitudes towards utilizing LLMs for smart contract development tasks (RQ1).

3.1.1 Survey Design. We distributed a mixed-methods survey to gain insights from developers on their experiences and perceptions of using LLMs for smart contract development. The survey consisted of 16 questions, including both multiple-choice and open-ended formats. We collected background information from participants, such as their blockchain and smart contract development experience, the industry in which they work, current job title, and familiarity with LLMs. The survey was designed to explore participants' perceptions of using LLMs for smart contract development, their reasons for or against using these tools, and the potential challenges and benefits associated with their adoption.

We received responses from 114 developers, which allowed us to gather comprehensive insights into the relevance of the survey questions and the key themes surrounding the adoption of LLMs. The questions covered topics such as familiarity with LLMs, perceived benefits and challenges of using LLMs for smart contract development, and willingness to adopt these tools in their workflows.

3.1.2 Participant Recruitment. To recruit participants, we used purposive sampling to target programmers with blockchain development experience. We distributed the survey through various channels, including online blockchain development communities, forums, and social media platforms, such as LinkedIn.[5] In addition, we sent recruitment emails to GitHub developers who were listed as contributors[6] to repositories for smart contracts in our dataset (see Section 3.2.1) with publicly available email addresses on their profile. Our study protocol for this phase was approved by our university's institutional review board (IRB) for human subjects research.

3.1.3 Participants. We received responses from 114 participants—all with blockchain development experience. Most participants (64%, $n = 73$) had between 0-1 years of blockchain experience, while 18% ($n = 21$) had 2-3 years, 12% ($n = 14$) had 3-4 years, and 4% ($n = 6$) had more than four years of experience. Among these, 4% ($n = 5$) participants were considered "experts" with very high familiarity with smart contract development. Participants' experiences spanned a wide variety of blockchain application domains, including gaming, education, finance, and other technologies. This sample also represents a variety of roles, including software developers and engineers, managers, consultants, security reviewers, and academic researchers. In addition, most participants ($n = 68$, 60%) had at least intermediate knowledge of AI and LLMs. 43 (38%) respondents reported using LLMs in smart contract development. Of those who responded to a follow-up question regarding which LLMs were used, 35 mentioned ChatGPT, 7 replied GitHub Copilot, 4 mentioned ChainGPT, 3 reported using OpenAI Codex, and 2 stated that they used Tabnine. Other tools were also mentioned by 2 participants. An overview of the survey participants is presented in Table 1.

3.1.4 Data Analysis. The survey responses were analyzed using a mixed-methods approach, incorporating both quantitative and qualitative techniques. Initially, the survey was publicly shared on LinkedIn to reach a broad audience of blockchain developers and researchers. However, this resulted in an unexpectedly high volume of responses, with a total of 6,401 submissions. Upon closer examination, a significant portion of these responses appeared to be spam or low-quality entries. After applying a systematic data-cleaning process, we retained 114 valid responses, indicating that approximately 98.2% of the submissions were spam.

To ensure data integrity, we implemented multiple filtering techniques:

[5] https://www.linkedin.com/feed/
[6] https://docs.github.com/en/repositories/viewing-activity-and-data-for-your-repository/viewing-a-projects-contributors

Figure 1: Overview of methodology

Table 1: Survey Respondent Demographics and Background (n=114)

Demographics	#	%
Blockchain Development Experience		
0-1 years	73	64.91%
2-3 years	21	18.42%
3-4 years	14	12.28%
> 4 years	6	4.31%
Industry		
Technology	67	58.77%
Gaming	12	10.53%
Education	11	9.65%
Finance	5	4.39%
Healthcare	4	3.51%
Other	15	13.15%
Role		
Researcher/Academic	43	37.72%
Developer/Engineer	28	24.56%
Project Manager	9	7.89%
Business Analyst	4	3.51%
Executive/Manager	5	4.39%
Other	25	21.93%
Knowledge of AI and LLMs		
Advanced Expertise	20	17.54%
Intermediate Knowledge	68	59.65%
Basic Understanding	21	18.42%
None	5	4.39%
AI Usage for Smart Contract Development		
Yes	43	37.72%
No	71	62.28%
LLMs Used (Follow-up)		
ChatGPT	35	30.70%
GitHub Copilot	7	6.14%

- *Duplicate and Automated Responses:* We removed duplicate entries by identifying repeated email addresses, IP addresses, and response timestamps. Entries from disposable email services were also flagged and excluded.
- *Response Quality Filtering:* We applied several heuristics to detect low-effort or automated responses, including:
  - *Completion Time:* Entries completed in an unrealistically short time were flagged as potential spam.
  - *Straight-Lining Detection:* Responses where the same option was selected across all Likert-scale questions were removed.
  - *Text-Based Analysis:* Open-ended responses containing gibberish, irrelevant content, or copy-pasted generic phrases were excluded.
- *Manual Review:* After applying automated filters, we conducted a manual review of borderline cases, focusing on open-ended responses to ensure meaningful engagement.

Following this data-cleaning process, we proceeded with the analysis using the refined dataset. For the quantitative analysis, we applied descriptive statistics to explore trends in closed-ended questions, allowing us to identify common patterns in participant experiences. For the qualitative analysis, we employed thematic analysis [16], a widely used method for identifying and interpreting recurring themes in textual data. One of the key themes that emerged was the concern over the accuracy and reliability of LLM-generated smart contracts, frequently mentioned by participants.

By integrating robust data-cleaning methods with a structured analytical approach, we ensured that our findings are based on high-quality, meaningful responses, enhancing the reliability and validity of the study.

3.2 Phase 2: Evaluation of Generative AI Tools

In the second phase, we evaluated the effectiveness of three generative AI tools, namely ChatGPT, Google Bard, and ChainGPT, for smart contract development (RQ2). We also specifically explore the capabilities of ChainGPT in comparison with the other tools to investigate the impact of customized training data relevant to blockchain applications on completing smart contract development tasks (RQ3).

3.2.1 Dataset Collection. To evaluate the effectiveness of generative AI tools for smart contract development, we gathered a diverse dataset of 102 decentralized applications (DApps) hosted on GitHub, spans across 17 distinct categories with their smart contracts written in solidity (.sol). This provide a diverse representation of DApp projects to ensure a representative sample of various contract types,

complexities, and use cases. The diversity of categories is further highlighted in the breakdown shown in Table 2, where the distribution of DApps across different categories is presented. We use this dataset to conduct an evaluation to investigate LLM-based smart contract development.

Table 2: DApps Dataset Categories Distribution

Category	Count	Category	Count
Finance	23	Governance	4
Exchanges	17	Media	3
Games	14	Identity	3
Social	9	Storage	4
Marketplaces	8	Security	5
Development	5	Other	5
Property	2	**Grand Total**	102

Our dataset includes various attributes such as DApp name, category, GitHub metrics (stars, watchers, forks, contributors, total commits), project status (active, archive), license information (MIT, MPL, Open-source, Apache, GPL), and specific details related to smart contract files and testing outcomes. This encompasses information such as the smart contract file name, links to our GitHub dataset, human written contract compilation status, tests passed and failed, time taken for testing, links to generated smart contracts by different AI models (Chat GPT, Gemeni, Chain GPT), and compilation status and testing outcomes of these generated contracts. These details allow for a thorough analysis of the performance and effectiveness of generative AI tools in smart contract development, including their ability to produce functional contracts and pass relevant tests. Key statistics of our dataset are presented in Table 3.

Additionally, we referred to the DApps Quality Characteristics Dataset[43] to supplement our dataset. We implemented various filters to guarantee the integrity and applicability of the data. We eliminated projects that lacked .sol files, had inaccessible public repositories, were not written in .sol lanaguage, contained broken links, or were not found or no longer existent. Additionally, we filtered out projects coded in outdated Solidity versions that are no longer compatible or verifiable. For instance, keywords like "constant" are obsolete in the context of function declarations in contemporary Solidity versions. Instead, developers should use "view" or "pure" depending on whether the function modifies the state or not.

3.2.2 AI Tools. For this evaluation, we assess the smart contract development capabilities of three AI tools—ChatGPT, Google Gemini, and ChainGPT. These systems, powered by LLMs trained on large amounts of data, take natural language input and produce output in a conversational dialogue.

Table 3: Distribution of GitHub Metrics in our DApps Dataset

Metric	avg	50%ile	75%ile	90%ile	max
Stars	98.5	17.5	80	336	867
Forks	76.5	9	56	307	814
Watchers	14.5	7.5	21	39	81
Contributors	9.7	3	10	26	104
Commits	1,121.8	88	507	1,492	47,711

ChatGPT. ChatGPT is a popular LLM-based chatbot provided by OpenAI trained on a large corpus of data [3]. Prior work shows ChatGPT can be effective for supporting various development tasks, including requirements engineering [64], software design [7], code generation [25], testing [24], debugging, and refactoring [8, 19]. Recent work has also investigated using ChatGPT for auditing smart contracts [65], detecting vulnerabilities [14], and smart contract generation [48]. We leveraged ChatGPT version 3.5 turbo plus.

Google Gemini. Google Gemini, formerly known as Google Bard, is an LLM-based assistant powered by Google AI [4]. Gemini has Code Assist functionality to support software development tasks.[7] Research suggests Gemini can successfully generate code for LeetCode[8] and GeeksforGeeks[9] programming contests [25]. Further, Gemini has shown capabilities to automate other software development tasks—including requirements engineering, design, and testing tasks in a comparison with GPT [11]. For this study, we use Gemini version 1.0 pro.

ChainGPT. ChainGPT is an AI chatbot designed to answer blockchain and cryptocurrency-related questions [2]. The platform is powered by models crafted for Web3, Blockchain, and Cryptocurrency applications.[10]—with the ability to solve complex problems in the cryptocurrency and blockchain domains [5, 62]. We used ChainGPT version, V1.6. To our knowledge, this is the first study to evaluate the code generation abilities of ChainGPT. While there is limited scholarly literature evaluating the capabilities of ChainGPT, the system incorporates a "Smart Contract Generator" to automate the creation of smart contracts.[11] We aim to evaluate this functionality, and explore the effects of customized training data on domain-specific applications of LLM-based development.

3.2.3 Prompts for Smart Contract Generation. For each generative AI tool, we provided prompts and instructions to generate smart contracts based on specific requirements and use cases from the smart contract file. Subsequently, a prompt for ChatGPT would be crafted to guide the generation of further smart contracts. The prompt generation process involved careful consideration of various factors to ensure the effectiveness and relevance of the generated contracts, with a particular focus on writing prompts in natural language.

(1) Identified key functionalities, data structures, and business logic.
(2) Determined contract types, input/output formats, and error handling mechanisms.
(3) Provided concise guidance on desired functionalities and behaviors.
(4) Added examples to clarify intended behavior.
(5) Refined prompts based on feedback and testing.

[7] https://cloud.google.com/gemini/docs/codeassist/overview
[8] https://leetcode.com/
[9] https://www.geeksforgeeks.org/
[10] https://www.chaingpt.org/#ecosystem
[11] https://www.chaingpt.org/blog/how-to-create-a-smart-contract-with-the-chaingpt-ai-generator

(6) Tested prompts to ensure effectiveness in guiding contract generation.

By following this step by step approach, researchers ensured that generative AI tools received clear and comprehensive guidance for generating smart contracts meeting desired requirements.

An example of a prompt used in the study is as follows, with other prompt examples uploaded in the replication package.

> *Write a Solidity smart contract named **SatoshiSignature** that implements the **IBEP20** interface. The contract should include implementations for the following:*
> - Total supply, decimals, symbol, and name of the BEP-20 token.
> - Ability to burn tokens and decrease the total supply.
> - Operations to transfer tokens between addresses.
> - Approval mechanism for allowing spending of tokens.
> - Ability to mint new tokens and assign them to an account.
> - Functionality to establish reflection fees for tokens.
> - Functions for buying tokens based on a specified rate.
> - Airdrop feature for distributing tokens to multiple addresses.
> - Owner's ability to start and pause the token sale.
> - Transfer functions that include automatic distribution of reflection fees to token holders.
> - A check to ensure that tokens are not transferred to the Uniswap router address when the reverseSwap flag is unset.
> - Implementation of the onlyOwner modifier for restricting certain operations to the contract owner.
>
> Ensure that the contract functions as specified, handling token transfers, reflections, burns, approvals, minting, airdrops, and owner privileges effectively.

This detailed prompt ensured that the AI tools could generate a smart contract aligned with the project's specific requirements.

3.2.4 Evaluation Procedure. We used generative AI tools to produce smart contracts based on the collected human-written samples in our dataset. The generated smart contract code was then subjected to an evaluation process to assess the quality and efficiency of LLM-based systems:

RQ2: Effectiveness. We performed a comparative analysis of the generated smart contract code across the three AI tools based on compiling code and unit testing to evaluate the capabilities of LLMs for generating smart contracts. Further, we compare the effectiveness of LLM-generated smart contracts with human-written code by evaluating the testing results with the original smart contracts in our dataset. This analysis allowed us to identify the strengths and limitations of each tool and provide recommendations for their effective utilization in smart contract development workflows. To measure the effectiveness of generative AI tools for smart contract development, we used the following metrics:

1. Compilation: Compiling source code is necessary to run programs. From our dataset, we observed how many smart contracts generated by the generative AI systems were able to compile. This was measured by the first author attempting to run the source code from human-written smart contracts on a random sample ($n = 30$) mined from GitHub and the LLM-generated smart contracts from each tool in the Remix IDE. If needed, minor changes were made to the source code to fix trivial issues (i.e., adding import statements for missing dependencies). We calculated the rate of success in compilation for each and use a Chi-squared test (χ^2) to quantitatively analyze differences between AI tools and the original smart contract.

2. Unit Testing: Prior work suggests smart contract testing is challenging, yet vital for verifying the correctness and behavior of programs that handle valuable assets and sensitive data [9]. We performed extensive unit testing to verify generated smart contracts across different scenarios and edge cases. We used manual and automatically-generated test suites to assess the effectiveness of LLM-generated smart contracts.

(1) *Manual Testing* The researchers manually generated unit tests for a subset of the dataset ($n = 30$) based on the requirements and use cases of smart contracts. This random sample—distinct from the sample used for the static analysis evaluation—was selected due to the manual efforts needed to analyze the smart contracts, generate test cases based on the specifications, and observe the results across to human-written and LLM-generated smart contracts.
(2) *Automated Testing* To augment our testing analysis, we also leveraged automatically generated unit tests to further verify the behavior of LLM-generated smart contracts. We used Remix IDE plugin, Solidity unit testing[12] to automatically generate unit tests for all of the remaining smart contracts in our dataset ($n = 72$). This framework automates the generation of unit tests and is integrated with the Remix development environment, developers can write, execute, and debug unit tests directly within Remix, streamlining the testing workflow [44]. Additionally, the framework includes built-in assertion libraries for defining assertions to validate the expected outcomes of contract functions and transactions.

For both types of testing, we analyze the percentage of test cases passed. We also use a Chi-squared test (χ^2) to observe differences between the AI-generated and human-written smart contracts. These results are compared across each of the LLM-based systems and the original human-written smart contracts in our dataset.

RQ3: Customized Training Data. To investigate the impact of customized training data in LLM-generated content, we compare the capabilities of ChainGPT—a model trained specifically for blockchain-related tasks— and ChatGPT.

1. Static Analysis: Static analysis tools—systems with the ability to analyze code without running the program—are crucial for detecting potential issues in smart contracts [20]. To analyze the quality of LLM-generated smart contracts, we employed industry-standard static analysis tools, including the Remix IDE Solidity static analyzer[13] and Solhint[14]. For instance, when analyzing a contract such as SocialNetwork.sol generated by ChatGPT, Remix IDE highlighted

[12]https://github.com/ethereum/remix-ide/blob/master/docs/unittesting.md
[13]https://remix-ide.readthedocs.io/en/latest/static_analysis.html
[14]https://protofire.github.io/solhint/

gas cost-related issues. Specifically, functions like SimpleSocialNetwork.createPost and SimpleSocialNetwork.createComment were flagged for requiring an infinite amount of gas. Gas refers to the computational cost of running a contract on the blockchain, and inefficient code can lead to higher fees. In this case, the flagged functions likely contained logic that could be optimized to reduce the amount of gas used, making the contract more efficient and cost-effective to execute.

Similarly, Solhint, another static analysis tool, identified concerns such as incompatible compiler versions and advised explicit visibility marking in functions. Additionally, both tools pointed out instances of similar variable names in different functions within the same contract, which could lead to confusion and potential errors during execution. Furthermore, guard conditions were recommended, with Solhint advising the use of "require(x)" for cases where the condition could be false due to invalid input or other factors.

As part of our data replication package, we included 30 analysis reports for transparency and reproducibility. We applied the static analysis tools on a random subset of smart contracts in our dataset ($n = 30$) to assess the generated code's adherence to best practices, identify potential vulnerabilities, and ensure compliance with coding standards. This was done to support manual efforts to run each of the static analysis tools on smart contracts to observe the number of reported issues and the types of errors reported. We use descriptive statistics and a Mann-Whitney U (U) test to compare the number and types of static analysis issues across the ChatGPT and ChainGPT-generated smart contracts from our dataset to explore the quality of LLM-generated output from these models.

2. Compilation and Testing We extend our compilation and manual and automated testing efforts to investigate the difference between smart contracts generated by ChatGPT and ChainGPT. For compilation, we use the entire dataset without manual interventions to observe the capabilities of the two AI assistants in generating accurate smart contracts. We compare the compilation success rate and the percentage of test cases passed—using a Chi-squared test (χ^2) to observe differences between smart contracts generated by LLMs trained on general (ChatGPT) and specific (ChainGPT) data.

3. Efficiency Evaluation: To investigate the efficiency of LLMs in smart contract generation, we measured the amount of time taken for each AI tool to produce code based on the given prompt. The time measurement was conducted using the Solidity Unit Testing plugin automatically upon compilation.

The findings from this evaluation process seek to increase our understanding of the capabilities and limitations of generative AI tools for smart contract development, as well as their potential impact on development processes and workflows. In particular, we examine the effects of customized training data on the performance of LLMs by evaluating the quality and efficiency of smart contract generation. By combining empirical data collection through the developer survey and rigorous evaluation techniques, our methodology aimed to provide a holistic and robust investigation of the research questions, contributing valuable insights to the field of blockchain-oriented software engineering.

4 RESULTS

The results of our study are presented in two parts, corresponding to the two phases of our methodology: the developer survey and the evaluation of generative AI tools for smart contract development.

4.1 RQ1: Blockchain Developer Perceptions

The online survey received responses from 114 blockchain and smart contract developers. We present participant perspectives on usage, benefits, challenges, and opportunities for using LLMs for smart contract development.

4.1.1 Usage and Perception. All participants except 10 ($n = 104$; 91.23%) reported having prior experience with LLMs. However, only 60.58% ($n = 63/104$) have used LLMs, with 87.3% reported using ChatGPT for smart contract development.

We observed mostly positive perceptions of using LLMs for smart contract development. All participants responded to a Likert scale question regarding the perception of using LLMs for smart contract development as *highly beneficial* or *potentially useful*. Few participants also reported reasons for not using LLMs for smart contract development, including (41.38%) mentioned the lack of awareness for LLM capabilities and (27.59%) for the lack of trust in AI-generated code. One participant mentioned they have not used it because they currently do not work in smart contract development, but plans to use it for smart contract generation in the future stating *"I worked on crypto projects before LLMs. Haven't gotten back into making smart contracts, but I'll probably start up again soon. I will definitely use ChatGPT and other LLMs to help me code"* (P10).

4.1.2 Benefits. When asked about the perceived benefits of using LLMs for smart contract development, the most commonly cited advantages were: Improved development efficiency by automating tasks ($n = 34$; 29.82%) and enhancing smart contract functionality to enable more complex smart contracts. For instance, P111 responded that LLMs can be *useful to get a headstart"* for code to generate smart contracts. Further, respondents noted that generative AI tools can be used to provide more advanced domain-specific functionality in smart contracts ($n = 27$; 23.68%). For example, P106 noted, *AI-generated smart contracts can manage complex financial instruments, such as dynamically adjusting the interest rates on loans or creating personalized investment strategies based on an individual's risk profile and market conditions*". One participant noted they did not see any benefits to integrating AI in smart contract development.

4.1.3 Challenges. On the other hand, there were numerous concerns and challenges expressed by the respondents with regard to using LLMs for smart contract generation. Responses include potential security risks and vulnerabilities in the generated code ($n = 36$; 31.58%), guaranteeing the accuracy and reliability of generated code ($n = 33$; 28.95%), adapting AI models to specific requirements in the smart contract domain ($n = 28$; 24.56%), legal and regulatory concerns ($n = 9$; 7.89%), and ethical issues ($n = 8$; 7.02%). Participants expressed concerns about the ability of LLMs to capture the nuances and complexities of this domain.

Table 4: Compilation Results

Type	Number	Percentage	*p-value*
ChatGPT	21	70%	$p = 0.3711$
ChainGPT	20	67%	$p = 0.2429$
Google Gemini	3	10%	$p < 0.0001^*$
Original	24	80%	—

* denotes statistically significant results (**p-value < 0.05**)

RQ1 Findings

Blockchain-based software developers rarely use LLMs to develop smart contracts, and—despite potential challenges—they desire novel mechanisms to incorporate generative AI tools into their development environment and workflows.

4.1.4 Opportunities. We were also interested in gathering developers' perspectives on opportunities for LLMs in smart contract development processes. In response to a question about how developers envisioned integrating LLM-generated contracts into existing development workflows: participants responded with desiring automated testing frameworks to validate functionality and security ($n = 29$; 25.44%), integrating AI tools into development environments and version control systems ($n = 26$; 22.81%), and providing documentation or training to educate developers ($n = 15$; 13.16%). Participants also emphasized various desired tools and features to support the integration of AI tools into development workflows. For instance, interactive debugging environments powered by LLMs ($n = 28$; 24.56%), collaborative platforms to share AI-generated smart contract code and obtain feedback ($n = 24$; 21.05%), standardized APIs ($n = 17$; 14.91%), and mechanisms to make sure output is "reliable, consistent, secure" (P46, $n = 20$; 17.54%).

Additional responses highlighted specific situations where AI-generated smart contracts would be most beneficial. For instance, one developer mentioned that LLMs could help generate code frameworks for standard smart contracts like ERC20 tokens, simplifying basic structures and saving development time (P67). Another respondent emphasized their usefulness for beginners or individuals without coding experience who want to develop personal Web3 projects, potentially bypassing the steep learning curve (P104). Other use cases include aiding in debugging, generating simple prototypes for design validation, and creating simple contracts for everyday use, such as tenant or loan agreements (P10, P15, P107).

These findings provided valuable insights into the perceptions, attitudes, and concerns of blockchain developers regarding the utilization of LLMs for smart contract development, providing insight into how generative AI techniques can be used to assist in the engineering and understanding of future smart contracts.

4.2 RQ2: Effectiveness of Generative AI Tools

The evaluation of the three generative AI tools, ChatGPT, Google Gemini, and ChainGPT, along with the original human-written smart contracts from our dataset yielded the following key results:

Table 5: Effectiveness Testing Results

Type	Passed	Failed	Test %	*p-value*
Manual				
ChatGPT	63	27	70%	$p = 0.043^*$
ChainGPT	53	19	74%	$p = 0.1661$
Gemini	7	5	58%	$p = 0.0252^*$
Original	88	19	82%	—
Automated				
ChatGPT	240	60	80%	$p = 1$
ChainGPT	124	31	80%	$p = 1$
Original	232	58	80%	—

* denotes statistically significant results (**p-value < 0.05**)

4.2.1 Compilation. The findings for the compilation of smart contract code are presented in Table 4. The original human-written smart contracts had the highest successful compilation rate, with 80% ($n = 24$) of smart contracts compiling successfully. However out of these, three required human intervention to add the required imports for the code to successfully compile. Among the smart contracts generated by ChatGPT, 70% ($n = 21$) passed, while 9 failed and two needed manual fixes. In contrast, ChainGPT produced 20 successfully compiled smart contracts and encountered 10 failures. Despite being trained to support blockchain-related tasks, such as smart contract development, ChainGPT had a worse compilation rate than ChatGPT and the original human written smart contracts. However, Google Gemini had the worst performance by far—producing only three valid smart contracts. This difference was also statistically significant ($\chi^2 = 29.687$, $p < 0.0001$), indicating an inability of Gemini to generate smart contracts effectively.

4.2.2 Testing. We used manual and automated testing to observe the quality of LLM-generated smart contracts. These results are depicted in Table 5. We found the majority of smart contracts generated by Google Gemini did not compile, and exclude them from this analysis.

Manual. For manual testing, we found the original human-written smart contracts that compiled had the highest percentage of passing test cases (82%, $n = 88/107$). Meanwhile, the compilable LLM-generated programs had lower passing rates for ChainGPT (74%) and ChatGPT (72%). Comparatively, there was a major drop off in the effectiveness of Google Gemini (58%). Further analysis shows the differences in testing results between the original smart contracts and those generated by ChatGPT ($\chi^2 = 4.0939$, $p = 0.043$) and Google Gemini ($\chi^2 = 5.0124$, $p = 0.0252$) are statistically significant.

Automated. For the automated testing, we found similar results across the LLM-generated and original human-written smart contracts, except for those generated by Google Gemini which was unable to support the automated testing. This could indicate limitations in automatic test case generation tools, which may aim to meet code coverage requirements and lack relevance to real-world code [17].

Table 6: Static Analysis Results

AI Tool	Total Issues	Average	*p-value*
Remix IDE			
ChatGPT	497	16.57	$p = 0.0029^*$
ChainGPT	125	6.94	—
Solhint			
ChatGPT	168	5.6	$p = 0.1835$
ChainGPT	69	3.83	—

* denotes statistically significant results (**p-value** < **0.05**)

RQ2 Findings

With the exception of Google Gemini, generative AI tools can generate compilable code for smart contracts. However, smart contracts generated by ChainGPT and ChatGPT have significantly worse quality with regard to manual unit testing efforts.

4.3 RQ3: Customized Training Data Comparison

4.3.1 Static Analysis. We conducted static analysis of the smart contracts generated by generative AI tools using Remix IDE solidity analyzer and Solhint. Our analysis involved the examination of contracts to identify potential issues and ensure code quality between. These results are presented in Table 6. In general, we found lower reported issues on average for successful smart contracts generated by ChainGPT compared to those generated by ChatGPT. Further, for warnings provided by the Remix static analyzer we observed significantly higher issues reported for ChatGPT-generated smart contracts ($U = 129.5$, $p = 0.0029$). This indicates the specialization of ChainGPT can produce higher quality code for generated contracts, with respect to the Remix IDE.

4.3.2 Compilation. In Table 4, our results show that ChatGPT had a higher success rate for smart contracts that can compile with minimal effort. Extending these results to the entire dataset, we found that among the smart contracts generated by ChatGPT, 79 (77%) passed. In contrast, ChainGPT produced 46 (45%) successfully compiled smart contracts. This result is also statistically significant ($\chi^2 = 24.97$, $p < 0.00014$), indicating that——despite ChainGPT being trained on data to support blockchain-related tasks, such as smart contract development—the general LLM ChatGPT produces compilable code more frequently. This comparison sheds light on the differences in the compilation success rates between the two AI models.

4.3.3 Testing. We also used automated and manual testing to observe the quality of smart contracts generated by general and specific generative AI tools. Using the findings from Table 5, we observed higher percentages of test cases passed for the manual tests, while automated test case percentage was the same across ChainGPT and ChatGPT. However, the difference for the manual testing results was not statistically significant ($\chi^2 = 0.2565$, $p = 0.6125$).

4.3.4 Efficiency. We also investigated the efficiency of ChainGPT and ChatGPT for generating smart contracts. On average, we found that both ChatGPT (0.085s) and ChainGPT (0.09s) were able to generate source code for smart contracts quickly with comparable results.

RQ3 Findings

ChainGPT, which uses customized training data for smart contract development, is significantly better for producing quality code with less reported static analysis issues—yet significantly worse for producing compilable code with comparable efficiency to ChatGPT.

5 DISCUSSION

The results of our study provide valuable insights into the potential of generative AI tools for efficient smart contract development, as well as the perceptions and attitudes of blockchain developers towards the adoption of these tools. We explored the capabilities of LLM-based tools, in particular ChatGPT, Google Gemini, and ChainGPT, for generating smart contracts based on natural language prompts. We compare AI-generated code with human-written code for smart contracts, and show that human effort is needed to produce effective code with regard to compileability and testing. Further, we show that customizing LLMs to fit certain domain-specific tasks can provide advantages and disadvantages. We found ChainGPT produce significantly invalid code that could not compile—yet when the code did compile, it was produced at a significantly higher quality. In this section, we discuss the key findings, implications, and future directions.

5.1 Perceptions and Attitudes of Blockchain Developers

The survey results revealed a mix of optimism and caution among blockchain developers regarding the utilization of large language models (LLMs) for smart contract development. This finding aligns with prior work investigating developer perceptions of LLMs in software development [51]. For instance, P1 noted *"As with any coding, I believe AI can help initiate the code but the specifics need to be done by a developer"*. We observed developers reported specific benefits and hesitation with adopting generative AI tools due to various aspects inherent in smart contract development. For instance, P12 replied *"AI-generated smart contracts would be most useful in complex, high-frequency trading environments where rapid, accurate, and transparent execution of financial transactions is crucial, such as in cryptocurrency exchanges or automated investment platforms"*.

We also observed survey participants highlight smart contract-specific concerns, such as security, trust, etc. For instance, P17 responded *"While I most certainly can see the multitude of benefits to efficiency and speed of workflow, widespread implementation of AI-generated smart contracts would concern me in the context of security and legality. Smart contracts are not bound by borders, and whatever AI system that would be handling their generation would have to ensure compliance with laws of varying jurisdictions. Additionally, if the AI generating smart contracts is not robust or advanced enough, poorly written and insecure code in relation to blockchain uses could easily be exploited by malicious actors"*. Despite these concerns, the majority still expressed a desire to use them. This

indicates developers overall perceive LLMs can support smart contract development—in particular to *"omit some complex details and save development time"* (P22).

5.2 Capabilities and Limitations of Generative AI Tools

Our evaluation of ChatGPT, Google Bard, and ChainGPT revealed strengths and limitations in their capabilities for smart contract development.

5.2.1 ChatGPT. ChatGPT emerged as the most effective tool for smart contract development, compared to ChainGPT and Gemini. Prior work shows this LLM is effective [54] and widely used [53] for code generation tasks in practice. We observed ChatGPT was able to generate more compilable smart contract code in our analysis. This demonstrates its potential utility in the smart contract development process. Moreover, prior work demonstrates ChatGPT is proficient in vulnerability detection [14], correction [27, 38], and auditing [65] smart contracts, highlighting its capabilities to generate as well as analyze and evolve smart contracts.

However, while ChatGPT exhibits strengths in certain aspects, it also has its limitations. For instance, it may encounter challenges in handling complex and nuanced requirements, leading to potential inaccuracies or inconsistencies in generated smart contracts. Additionally, the reliance on chat-based prompts may introduce ambiguity and variability in the generated outputs [33], necessitating careful refinement.

5.2.2 Google Gemini. Google Gemini was unable to demonstrate competency in generating compilable code. Its integration with Google's training data enables it to effectively generate code [46]. However, Google Gemini may lack the depth and specificity required for complex smart contract scenarios, limiting its applicability in certain use cases. Similarly, Google Gemini struggled to produce correct and compilable smart contracts based on human-written unit tests, with only 2 out of 30 initial DApps generating accurate outputs. Consequently, we decided to exclude Google Gemini from further analysis due to its lack of effectiveness.

5.2.3 ChainGPT. On the other hand, ChainGPT showcased proficiency in understanding and generating smart contract logic, leveraging its extensive training on blockchain-related topics. Its ability to handle technical complexities and industry-specific terminology makes it a valuable tool for developers. For example, ChainGPT correctly imported OpenZeppelin,[15] open libraries for secure smart contract development, demonstrating its reliability in generating robust and standards-compliant smart contracts.

These findings suggest, while generative AI tools can enhance the efficiency and productivity of smart contract development, they should be used in conjunction with human expertise and domain knowledge. For instance, participants noted LLMs can be particularly useful for generating smart contract code templates and modules (P7), automating repetitive tasks (P27), and providing solutions to common challenges (P16), making the development process faster and more accessible. Prior work suggests LLMs excel at simple coding tasks, but lack the ability to support more complex development activities [61]. Our results suggest the role of developers remains essential in refining, optimizing, and validating the generated output for smart contract code.

[15] https://github.com/OpenZeppelin/openzeppelin-contracts

5.3 Future Directions

Our findings motivate several directions for future work. First, participants highlighted the need for robust mechanisms to ensure that AI-generated code is reliable, consistent, and secure. For instance, one participant (P24) suggested integrating formal verification rules and automated testing frameworks directly into the generative process. Prior work suggests leveraging formal verification to enhance LLM-generated code [68] and smart contract development [22]. Building on this, future efforts should focus on incorporating testing, auditing, and formal verification frameworks into LLMs that specialize in smart contract generation. These frameworks could include vulnerability detection, adherence to security best practices, and formal correctness checks to ensure generated smart contracts meet industry standards.

Another future area is domain-specific fine-tuning of LLMs for smart contract development. While general-purpose models like ChatGPT demonstrate strong capabilities, their performance can be enhanced through targeted training using domain-specific datasets [23]. This would enable models to better understand the nuances of smart contracts, such as token standards (e.g., ERC20, BEP20) and secure contract interactions. Domain-specific fine-tuning can improve the accuracy and contextual relevance of AI-generated code [63], making it more immediately usable by developers.

As the capabilities of generative AI tools continue to evolve, it is essential to explore effective strategies for integrating these tools into existing smart contract development workflows. For example, future studies can explore the potential of generative AI tools for reducing development time and effort required for smart contract development compared to manual coding methods. This may involve developing specialized plugins, APIs, or integrated development environments (IDEs) that seamlessly incorporate AI-assisted code generation, documentation, and testing capabilities. Seamless integration into existing workflows can accelerate the adoption and practical utilization of these tools by developers [32].

6 THREATS TO VALIDITY

External Validity. There are several threats to the generalizability of our research findings. First, our survey sample may not generalize to all smart contract developers' perceptions of LLMs. This may not fully capture the perspectives of more experienced smart contract engineers. This limitation reflects broader recruitment challenges encountered in human-centric computing research involving specialized technical communities [29]. Future work can conduct a larger scale qualitative analysis, employing surveys or interviews to gain further insights from blockchain developers. Additionally, our evaluation only examines ChatGPT, Google Gemini, and ChainGPT to explore LLM-generated smart contracts. To mitigate this, we attempted to leverage varying models trained on different data. Future studies can leverage other AI assistants or LLM-based programming assistant, such as GitHub Copilot, to further explore the capabilities of LLMs for supporting smart contract development. Finally, the dataset for our evaluation consists of a limited number

of Ethereum-based smart contracts written in Solidity. However, smart contracts can be implemented on different blockchains (e.g., Binance Smart chain[16]) and different programming languages (i.e., TypeScript[17] [49]).

Internal Validity. There are also several limitations to our study design. Several questions in the survey were not required, and thus some participants failed to provide responses for specific questions. In addition, our recruitment approach faced limitations—incurring a large number of span responses. Our filtering process may have removed valid responses or incorporated invalid ones. For phase 2, we rely on specific static analysis and testing tools, Solhint and the Remix Solidity analyzer, to evaluate LLM-generated smart contracts. While these tools are widely adopted in practice, using different analysis systems—such as Slither [20]—could impact our results. Our study also involves human analysis of generated smart contracts, which could incorporate bias.

7 CONCLUSION

This research investigated the potential of leveraging generative AI tools powered by large language models (LLMs) for efficient and secure smart contract development. Through a developer survey and evaluation of ChatGPT, Google Bard, and ChainGPT, our study yielded valuable insights. The survey revealed a mix of optimism and caution among blockchain developers regarding LLM adoption, highlighting concerns about security risks and integration challenges. The evaluation demonstrated the tools' capabilities in facilitating smart contract understanding and generating code, with ChatGPT outperforming Google Gemini and ChainGPT. Notably, the use of these tools showed promising potential in reducing development time and effort. However, thorough verification and validation by experienced developers remain crucial. Based on our findings, we proposed future directions, including developing robust testing frameworks, exploring domain-specific fine-tuning of LLMs, developing interpretable AI models, integrating AI tools into workflows, and conducting longitudinal studies.

REFERENCES

[1] AI | 2024 Stack Overflow Developer Survey — survey.stackoverflow.co. https://survey.stackoverflow.co/2024/ai#sentiment-and-usage-ai-select. [Accessed 02-08-2024].

[2] Chaingpt. https://www.chaingpt.org/.

[3] Chatgpt. https://chat.openai.com.

[4] Gemini. https://gemini.google.com.

[5] Unleashing the power of ai and blockchain technology: An overview of the chaingpt model. Technical report. https://assets-global.website-files.com/63d0e411b048e60e70c275df/63d6498bd4b43923151548ac_research.pdf.

[6] Smart contract development: Real-world use cases ... - LinkedIn. https://www.linkedin.com/pulse/smart-contract-development-real-world-use-cases-interesting/, 2024. Accessed: 2024-04-18.

[7] Aakash Ahmad, Muhammad Waseem, Peng Liang, Mahdi Fahmideh, Mst Shamima Aktar, and Tommi Mikkonen. Towards human-bot collaborative software architecting with chatgpt. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 279–285, 2023.

[8] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations. *arXiv preprint arXiv:2402.06013*, 2024.

[9] Morena Barboni, Andrea Morichetta, and Andrea Polini. Smart contract testing: Challenges and opportunities. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 21–24, 2022.

[10] Cristian Bartoletti and Barbara Pes. An empirical analysis of smart contracts: Platforms, applications, and design patterns. *arXiv preprint arXiv:1703.06322*, 2017.

[11] Lenz Belzner, Thomas Gabor, and Martin Wirsing. Large language model assisted software engineering: prospects, challenges, and a case study. In *International Conference on Bridging the Gap between AI and Reality*, pages 355–374. Springer, 2023.

[12] J. Benson. Uniswap trading volume exploded by 450 https://decrypt.co/63280/uniswap-trading-volume-exploded-7-billion-heres-why, 2024. Accessed: 2024-10-09.

[13] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Ethereum Whitepaper*, 2014.

[14] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. When chatgpt meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520*, 2023.

[15] X. Chen. Ai-powered code analysis and testing tools. *Computers & Security*, 58:45–55, 2021.

[16] Victoria Clarke and Virginia Braun. Thematic analysis. *The journal of positive psychology*, 12(3):297–298, 2017.

[17] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67, 2017.

[18] Gabriele De Vito. Assessing healthcare software built using iot and llm technologies. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 476–481, 2024.

[19] Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. Exploring chatgpt's code refactoring capabilities: An empirical study. *Expert Systems with Applications*, 249:123602, 2024.

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

[21] Edwin Frank and Olaoye Godwin. Enhancing developer productivity: A study on github copilot's code completion capabilities. 2024.

[22] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. A survey on formal verification for solidity smart contracts. In *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, pages 1–10, 2021.

[23] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22, 2025.

[24] Vitor Guilherme and Auri Vincenzi. An initial investigation of chatgpt unit test generation capability. In *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, pages 15–24, 2023.

[25] Wenpin Hou and Zhicheng Ji. A systematic evaluation of large language models for generating programming code. *arXiv preprint arXiv:2403.00894*, 2024.

[26] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 297–306. IEEE, 2023.

[27] Giacomo Ibba, Marco Ortu, Roberto Tonelli, and Giuseppe Destefanis. Leveraging chatgpt for automated smart contract repair: A preliminary exploration of gpt-3-based approaches. *Available at SSRN 4474678*.

[28] Shawal Khalid and Chris Brown. Software engineering approaches adopted by blockchain developers. In *2023 Tenth International Conference on Software Defined Systems (SDS)*, pages 1–6, 2023.

[29] Shawal Khalid and Chris Brown. Exploring stakeholder challenges in recruitment for human-centric computing research. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 432–438, 2024.

[30] Shawal Khalid, Huayu Liang, and Chris Brown. Exploring user perceptions of crypto signals: An empirical study from social media. In *2024 IEEE International Conference on Blockchain (Blockchain)*, pages 483–490, 2024.

[31] Shawal Khalid, Huayu Liang, and Chris Brown. Decoding the influence: Analyzing the impact of crypto signals on software repositories. https://www.researchgate.net/publication/388194415_Decoding_the_Influence_Analyzing_the_Impact_of_Crypto_Signals_on_Software_Repositories, 2025. Accepted at IWBOSE 2025, co-located with SANER 2025. Preprint available on ResearchGate.

[32] Jenny T Liang, Chenyang Yang, and Brad A Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13, 2024.

[33] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. Guiding chatgpt for better code generation: An empirical study.

[16] https://www.binance.org/en/smartChain

[17] https://www.typescriptlang.org/

In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 102–113. IEEE, 2024.

[34] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580*, 2024.

[35] Yu Luo, Weifeng Xu, Karl Andersson, Mohammad Shahadat Hossain, and Dianxiang Xu. Fellmvp: An ensemble llm framework for classifying smart contract vulnerabilities. In *2024 IEEE International Conference on Blockchain (Blockchain)*, pages 89–96. IEEE, 2024.

[36] Muhammad Izhar Mehar, Charles Louis Shier, Alan Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Banking & Insurance eJournal*, 2017.

[37] Viraaji Mothukuri, Reza M Parizi, and James L Massa. Llmsmartsec: Smart contract security auditing with llm and annotated control flow graph. In *2024 IEEE International Conference on Blockchain (Blockchain)*, pages 434–441. IEEE, 2024.

[38] Emanuele Antonio Napoli and Valentina Gatteschi. Evaluating chatgpt for smart contracts vulnerability correction. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1828–1833. IEEE, 2023.

[39] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, 2022.

[40] Anh Nguyen-Duc, Beatriz Cabrero-Daniel, Adam Przybylek, Chetan Arora, Dron Khanna, Tomas Herda, Usman Rafiq, Jorge Melegati, Eduardo Guerra, Kai-Kristian Kemell, Mika Saari, Zheying Zhang, Huy Le, Tho Quan, and Pekka Abrahamsson. Generative artificial intelligence for software engineering – a research agenda, 2023.

[41] Chao Ni, Cong Tian, Kaiwen Yang, David Lo, Jiachi Chen, and Xiaohu Yang. Automatic identification of crash-inducing smart contracts. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 108–119. IEEE, 2023.

[42] S. Nijssen. Generative artificial intelligence for software engineering. *arXiv preprint arXiv:2310.18648*, 2023.

[43] Author(s) of the Dataset. Dapps quality character- istics datase. https://zenodo.org/records/3382127. Accessed: Insert date accessed.

[44] Mayowa Olatunji. Solidity tutorial: How to use remix ide for solidity smart contract development. *Medium*, 2023. https://medium.com/coinmonks/solidity-tutorial-how-to-use-remix-ide-for-solidity-smart-contract-development-d0d2ce6da051.

[45] Pierluigi Paganini. Ethereum parity wallet freeze that locked up $280 million in ether was a hack, claims cappasity. November 2017. Accessed: 2025-03-15.

[46] Hetvi Patel, Kevin Amit Shah, and Shouvick Mondal. Do large language models generate similar codes from mutated prompts? a case study of gemini pro. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 671–672, 2024.

[47] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 2023.

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

[49] QANplatform. Write smart contracts in typescript. *Medium*, 2023. https://medium.com/qanplatform/write-smart-contracts-in-typescript-powered-by-qanplatform-ca6bac09dca4.

[50] et al. Qin. Machine learning-based approach for detecting vulnerabilities in smart contracts. *Journal of Blockchain Research*, 2(1):1–15, 2021.

[51] Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. An empirical study on usage and perceptions of llms in a software engineering project. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 111–118, 2024.

[52] Jorge Feliú Rey. 3. smart contract: conceito, ecossistema e principais questões de direito privado. *Redes*, 2019.

[53] Daniel Russo. Navigating the complexity of generative ai adoption in software engineering. *ACM Transactions on Software Engineering and Methodology*, 2024.

[54] Fardin Ahsan Sakib, Saadat Hasan Khan, and AHM Rezaul Karim. Extending the frontier of chatgpt: Code generation and debugging. In *2024 International Conference on Electrical, Computer and Energy Technologies (ICECET*, pages 1–6. IEEE, 2024.

[55] Jaibir Singh, Suman Rani, and Parveen Kumar. Blockchain and smart contracts: Evolution, challenges, and future directions. In *2024 International Conference on Knowledge Engineering and Communication Systems (ICKECS)*, volume 1, pages 1–5, 2024.

[56] Margaret-Anne Storey, Neil A Ernst, Courtney Williams, and Eirini Kalliamvakou. The who, what, how of software engineering research: a socio-technical framework. *Empirical Software Engineering*, 25:4097–4129, 2020.

[57] Nick Szabo. Smart contracts. *First Monday*, 1(5), 1994.

[58] Nick Szabo. The idea of smart contracts. *Nick Szabo's papers and concise tutorials*, 6(1):199, 1997.

[59] et al. Tann. Automatically generating smart contract specifications from natural language requirements. *Journal of Software Engineering Research and Development*, 6(1):1–18, 2018.

[60] Che Wang, Jiashuo Zhang, Jianbo Gao, Libin Xia, Zhi Guan, and Zhong Chen. Contracttinker: Llm-empowered vulnerability repair for real-world smart contracts. *arXiv preprint arXiv:2409.09661*, 2024.

[61] Wei Wang, Huilong Ning, et al. Rocks coding, not development–a human-centric, experimental evaluation of llm-supported se tasks. *arXiv preprint arXiv:2402.05650*, 2024.

[62] Joko Waru. Chaingpt: A cryptocurrency and blockchain ai language model. *Linkedin Pulse*, 2023. https://www.linkedin.com/pulse/chaingpt-cryptocurrency-blockchain-ai-language-model-joko-waru.

[63] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 2023.

[64] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative ai for effective software development*, pages 71–108. Springer, 2024.

[65] Shihao Xia, Shuai Shao, Mengting He, Tingting Yu, Linhai Song, and Yiying Zhang. Auditgpt: Auditing smart contracts with chatgpt. *arXiv preprint arXiv:2404.04306*, 2024.

[66] Q. Xu. Regulating blockchain: Techno-social and legal challenges. *Computers & Security*, 58:45–55, 2017.

[67] Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. Domain knowledge is all you need: A field deployment of llm-powered test case generation in fintech domain. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 314–315, 2024.

[68] Sichao Yang and Ye Yang. Formaleval: A method for automatic evaluation of code generation via large language models. In *2024 2nd International Symposium of Electronics Design Automation (ISEDA)*, pages 660–665. IEEE, 2024.

[69] Wenxuan Zhao, Wei Mi, and Xiaodan Zhang. The security paradox of smart contracts: Blind spots and prospects of current detection strategies. In *2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 1546–1551, 2024.

[70] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4):352–375, 2018.