2024 IEEE International Conference on Software Services Engineering (SSE) | 979-8-3503-6851-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/SSE62657.2024.00050

Characterizing, Detecting, and Correcting Comment Errors in Smart Contract Functions

Yutong Cheng*, Haowen Yang*, Zhengda Li†, Lei Tian†
*School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen, China
†School of Data Science, The Chinese University of Hong Kong, Shenzhen, China
Email: {yutongcheng1, haowenyang1, zhengdali1, leitian}@link.cuhk.edu.cn

***Abstract*—NatSpec comments play an essential role in smart contracts. Their clear and informative format helps users gain an accurate understanding of smart contract functions and diminish financial risk. However, widespread non-adherence to NatSpec standards currently causes confusion for both end-users and developers. Current research often neglects the importance of NatSpec formats or solely emphasizes user-centric comments in smart contract generation. This oversight can hinder contract trustworthiness, code reusability, maintenance efficiency, and ultimately, the development of the community ecosystem. To bridge this gap, this paper presents the first empirical study on 253 verified contracts encompassing 16,620 functions from Etherscan, uncovering that 87% of the smart contract functions have *Comment Errors (CE)* and pinpointing prevalent deviation patterns. Based on our findings, we propose *CETerminator*, an automated approach for detecting and rectifying *CE* in smart contract functions. Due to the scarcity of NatSpec-compliant comments for collected smart contract functions, *CETerminator* employs in-context learning on a large language model to generate NatSpec comments. The approach then compares the original and the generated comments, utilizing corpus-driven heuristic rules to identify and correct diverse error categories in the original comments. In our evaluation, *CETerminator* demonstrates a high token overlap rate for addressing missing comments. In addition, the average precision, recall, and F1-scores for handling inconsistency comments are 85.28%, 86.48%, and 85.85%, respectively, outperforming the baseline by 39.79%, 39.53%, and 39.84%.**

I. Introduction

In recent years, cryptocurrency and blockchain technology have gained significant attention [1], [2] in both industry and academia. As the core of Ethereum, smart contracts [3] have expanded into a variety of application scenarios, such as supply chain management [4], decentralized finance (DeFi) [5], and digital identity management [6]. These contracts involve end-users interacting with their functions through transactions that consume "gas" priced in Ethereum's native cryptocurrency [7], Ether [8]. Given the significant financial impact [9], a clear understanding of smart contract functionality is crucial for both end-users and developers, as it ensures seamless interaction and effective implementation across various applications.

The Ethereum Natural Language Specification Format (NatSpec) was introduced to facilitate a clear understanding of smart contracts. It employs specific tags to categorize comments for distinct audiences: for instance, the `@notice` tag is designated for user-oriented comments, while the `@dev` tag is used for developer-focused notes. As illustrated in Fig. 1, the function `getOwnedTokens` is designed to query the list of

```
/**
 * @notice Gets the complete list of token ids which belongs to
 * an address
 * @param eth_address The address you want to lookup owned tokens from
 * @return List of all owned by eth_address tokenIds
 */
function getOwnedTokens(address eth_address) public view returns (uint256[])
{
        return stables.getOwnedTokens(eth_address);
 }
```

Fig. 1. An example for NatSpec format comments.

token IDs owned by a specific Ethereum address. The function is well-documented with NatSpec comments, providing clear explanations for users and developers on the function purpose (`@notice`), parameter descriptions and return variables. The example provides an initial insight into the advantages of NatSpec comments. By utilizing specified tags for different purposes, NatSpec comments offer a format that is both clearer and more informative than traditional free-form comments. This, in turn, enhances the readability and explainability of smart contracts. Consequently, the use of NatSpec has been strongly recommended in the official documentation [10] of Solidity, the primary programming language specifically designed for developing Ethereum-based smart contracts.

However, the existence of comments non-compliant with NatSpec is prevalent (see Sec. III). Such non-compliance in smart contracts can lead to misplaced, indigestible, or empty items during document and transaction interface generation, negatively impacting developers' and users' understanding and potentially reducing the trustworthiness of smart contracts. In Fig. 2, the `get UnderlyingPrice` function includes user-oriented explanations, such as *"Get the underlying price of a cToken"*, as well as developer-focused technical details like *"Implements the PriceOracle interface for Compound v2."* Additionally, the function provides descriptions for its return values, stating *"Price in USD with 18 decimals"*, and for its parameters, specifying *"The cToken address."* Contrastingly, the `transferForm` function is inadequately documented, leaving both users and developers in the dark about its purpose and technical intricacies. This lack of information poses a risk, as it may lead to unawareness of potential security vulnerabilities, such as reentrancy attacks.

Despite numerous tools and plugins being developed to address the issues associated with comment errors, existing solutions suffer from certain limitations. The hardhat-output-

Fig. 2. An example of transaction interfaces generated from a function with NatSpec-compliant comments and a function without comment.

validator [11] plugin only detects missing comment errors without offering solutions to identified issues. The solidity-comments-core module [12], on the other hand, can generate tags and corresponding default comments but fails to provide a custom natural language description for each tag. These limitations underscore the need for more advanced tools to effectively identify and rectify comment errors in smart contracts.

To comprehend the prevalence and common characteristics of comment errors, we conducted an empirical study on 16,620 functions across 253 verified smart contracts crawled from Etherscan, a blockchain explorer providing real-time data of transactions, smart contracts, and token transfers. Our investigation was guided by the following research questions:

- **RQ1 (Prevalence & Impact):** *How prevalent are comment errors in smart contracts? What is the impact brought by comment errors?*
- **RQ2 (Patterns):** *What are the common patterns of comment errors?*

By examining the research questions, we discovered that non-compliance with NatSpec comments is a widespread issue. Specifically, 87% of all functions in smart contracts exhibit comment errors. Interestingly, we observe that smart contracts with a higher number of comment error issues tend to have fewer transactions. Regarding comment error patterns, we find that the complete absence of comments is the predominant issue, followed by partially missing and misassigned comments. In terms of partially missing comments, the most frequent type was the simultaneous lack of `@notice`, `@return`, and `@param` tags, accounting for 27.9% of all such errors. The second most common issue was the exclusive omission of `@return` tags, which occurred 1,003 times and made up 21.8% of the partially missing comment errors in the dataset. As for misassigned comments, the primary issue was the incorrect placement of `@notice` content under the `@dev` tag, constituting 88% of all identified misassigned errors.

Motivated and inspired by our findings, we further propose *CETerminator*, a novel approach that automatically detects and corrects comment errors of all kinds in a comprehensive way. Our approach consists of two main modules: (1) *Reference Comment Generator*. This module leverages in-context learning with an LLM through demonstration-based prompts, using demonstration-based prompts to take advantage of the model's code comprehension and natural language generation for generating comments. These comments then serve as a reference for correcting comment errors in the subsequent module. (2) *NatSpec Formatter*. This module detects and rectifies comment errors based on error categories and findings of the empirical study.

We evaluated our approach on two collected datasets: a refined set of $code, tagged\ comment$ pairs for comment completion evaluation, and a manually annotated set of $code, tagged\ comment, error\ type$ tuples for inconsistency error detection. Our approach achieves a median token overlap rate of 53 for completion and exhibits an average precision, recall, and F1-scores of 85.28%, 86.48%, and 85.85%, surpassing the baseline by 39.79%, 39.53%, and 39.84%. In addition, we perform a human evaluation to assess practitioners' views on the generated comments, yielding scores for similarity (3.10/5), informativeness (3.06/5), and naturalness (3.21/5). The results reveal that the performance of generated comments is comparable to properly written comments, justifying their utilization as reference answers in our approach.

The main contributions of this paper are as follows:

- **Study.** To the best of our knowledge, we are the first to study comment error issues in smart contracts, offering insights into comment error patterns and informing future research in this domain.
- **Dataset.** We release the first dataset[1] for comment error correction and detection in smart contracts, consisting of 16,620 comment functions extracted from 253 real-world contracts on Etherscan.
- **Technique.** We introduce *CETerminator*[1], an LLM-based innovative solution for addressing comment errors in smart contracts. The experimental findings indicate that *CETerminator* is capable of generating high-quality comments while accurately identifying and correcting comment errors in smart contracts.

II. PRELIMINARIES

A. NatSpec: Solidity Smart Contract Comment Standard

In smart contract development using Solidity, a high-level language for Ethereum blockchain, proper comments are vital for reliability and maintainability of contracts [13]. The NatSpec has improved the documentation process by offering a human-readable, machine-verifiable syntax. NatSpec employs tag-based comments, including `@dev`, `@notice`, `@param`,

[1] https://anonymous.4open.science/r/CETerminator-repo-5150/

and @return, to provide clear and concise descriptions of functions, parameters, and return values. The @dev tag is specifically designed for developer-oriented descriptions, offering detailed information like internal workings and code logic that assists in code understanding and maintenance. On the other hand, the @notice tag serves to provide user-facing explanations, aiming to clarify the function's purpose and behavior for end-users. In addition, the @param tag describes the purpose and constraints of each input parameter, while the @return tag outlines what the function returns and under what conditions. These tags promote seamless interaction and integration with user interfaces and development tools. [14].

Additionally, NatSpec comments serve other purposes that extend beyond code annotation. Firstly, they can be parsed to create comprehensive developer documentation [15]. For example, tools like Doxygen [16] or custom parsers can automatically generate documentation that includes function descriptions, parameter explanations, and return value details, all sourced from the NatSpec comments. This is particularly useful for complex smart contracts where understanding the codebase quickly is crucial for development and maintenance. Furthermore, NatSpec comments significantly enhance the user experience through understandable function descriptions and user-friendly interfaces [17]. For instance, a decentralized application interface could display a tooltip with the @notice description when a user hovers over a function, providing immediate context and reducing the barrier to entry for non-technical users.

While NatSpec offers numerous advantages in smart contract development, comment errors can pose challenges to the security and usability of decentralized applications. Missing comments, such as omitting @param or @return annotations can lead to misconceptions about the smart contract's functionality. This may cause developers to inadvertently introduce vulnerabilities, or end users to interact with the contract in a way that exposes them to potential risks. Furthermore, inconsistent comments, such as misassigned annotations where the content intended for @notice and @return is mistakenly placed under @dev, can contribute to confusion and misinterpretation of the smart contract's behaviour. Inadequate documentation hinders auditors and third-party developers in evaluating contract security and correctness, potentially hampering efforts to identify and address vulnerabilities pre-deployment. Given the benefits of adopting NatSpec in Solidity, it's imperative to use @notice, @dev, @param, and @return tags accurately. This ensures robustness, security, and usability in decentralized applications and mitigates the risks associated with poor documentation.

B. In-Context Learning

In-context learning [18] is a powerful approach employed by large language models (LLMs) to adapt to tasks without the need for extensive fine-tuning. By incorporating a limited set of examples within the input context, LLMs can effectively generalize to diverse tasks, drawing on their extensive pre-trained knowledge. [19] [20]. In-context learning [21] has been successfully applied to various generation tasks, such as summarization, translation, and code generation. For instance, when provided with a few examples of English-French translations within the input context, LLMs like GPT-3 can generate accurate translations for subsequent English sentences [22]. The task of generating well-formatted NatSpec comments for Solidity code is hindered by the lack of a substantial dataset. This scarcity of data, referred to as "NatSpec hungry," poses a challenge for traditional supervised learning approaches, which typically require extensive labeled data to perform well. Consequently, our approach (see IV) employs in-context learning to effectively generate proper NatSpec comments. This enables the model to transfer its pre-trained knowledge [23] to the specific task of generating NatSpec comments, overcoming the limitations posed by the scarcity of available data.

III. EMPIRICAL STUDY

A. Data Collection

In accordance with established data collection procedures in previous studies [24], [25], our dataset was prepared in two stages.

Step 1: Subjects Selection. Etherscan serves as a leading blockchain explorer, offering detailed analytics and information on Ethereum-based smart contracts, including transactions, addresses, and source codes. To analyze patterns and prevalence of comment errors, we crawled and randomly sampled 253 verified contracts with 16,620 functions, spanning four transaction volume regions: 0∼100, 100∼10k, 10k∼100k, and 100k+. This stratified sampling method guarantees diverse representation across varying transaction volumes, thereby facilitating a thorough understanding of commenting practices and their influence on smart contracts.

We chose to focus on *public* and *external* implemented functions in this study for the following reasons. From the visibility perspective, *internal* and *private* functions are not directly accessible by external parties, reducing their impact on the overall understanding and interaction with the smart contract. From the implementation perspective, abstract functions provide limited information and are generally easier for developers and users to comprehend. By focusing on implemented functions, we can better assess the challenges faced by developers in understanding and maintaining more complex smart contract code.

Step 2: Identify Comment Errors. To identify comment errors in the 253 projects, we employed a two-step approach. First, we examined whether functions lacked @param, @return, and @notice tags when the function body indicated that such comments should be present. This allowed us to extract functions with missing comments, which constitute one category of comment errors. Next, we manually analyzed all functions extracted from the first step. Our study consisted of three phases:

- Phase I: The two authors manually studied 16,620 randomly sampled functions to derive an initial list of comment error patterns. All disagreements were discussed until a consensus was reached.

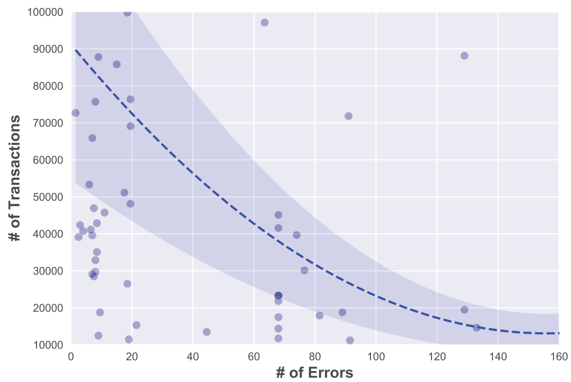

Fig. 3. The amount of transactions versus Error numbers of smart contracts.

- Phase II: The two authors independently categorized all functions according to the derived patterns from Phase I. No new patterns were discovered in this phase. The results of this phase yielded a Cohen's kappa of 0.806, indicating a substantial level of agreement.
- Phase III: The two authors discussed the categorization results obtained in Phase II. Any disagreements were resolved through discussion until a consensus was reached.

B. RQ1 (Prevalence & Impact)

*1) **Prevalence**:* Initially, we investigated comment error prevalence by analyzing 16,620 smart contract functions. We found that 14,470 (87%) functions had comment errors.

[colback=gray!10, colframe=black, arc=10pt, boxrule=2pt] **Finding 1:** 87% of all functions in smart contracts have comment errors, motivating dedicated approaches for detecting and correcting comment errors.

*2) **Impact**:* Standardized and comprehensive NatSpec comments could enhance readability and function quality, potentially increasing user engagement. To explore this hypothesis, we analyzed transaction data and investigated the correlation between comment quality and transaction volumes. Fig. 3 demonstrates that contracts with a higher number of errors typically exhibit lower transaction volumes in our randomly-selected dataset of contracts with 10k-100k transaction volumes. Consequently, users tend to use contracts with high-quality, NatSpec-compliant comments to conduct transactions. It implies that the absence of NatSpec-compliant comments affects the popularity of a smart contract.

[colback=gray!10, colframe=black, arc=10pt, boxrule=2pt] **Finding 2:** There exists a negative correlation between the frequency of errors and the associated transaction volume.

C. RQ2 (Manifestation Patterns)

We classify the comment errors investigated in our study into two categories: *missing comments* and *inconsistent comments*. In the subsequent sections, we will delve into a detailed exploration of these manifestation patterns and provide illustrative examples.

```
/**
 * @dev This replicates the behavior of the
 * https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_sign[`eth_sign`]
 * JSON-RPC method.
 * See {recover}.
 */
function getOwnedTokens(address eth_address)
public view returns (uint256[]) {
        return stables.getOwnedTokens(eth_address);
}
```

Fig. 4. `@dev` comments with information unrelated with function.

1) Pattern A: missing comments: In this paper, we define *missing comments* as the absence of tagged comments when the associated code suggests their necessity. For instance, in Fig. 5, the code should have included NatSpec comments such as `@notice`, `@dev`, `@param` (*oracle*) and `@param` (*newOracle*). Several factors, including time constraints, inconsistent commenting practices, overconfidence in code's self-explanatory nature, and failure to update comments, contribute to missing comments in smart contracts, potentially resulting in adverse outcomes. For developers, this leads to reduced code readability, longer onboarding times, and a greater chance of errors. For users, it elevates the risk of scams and fraud, complicates unauthorized transaction detection, and diminishes trust and satisfaction.

Missing comment issues can be further classified based on our finding on the extent of missing: partial or complete. Additionally, considering the comment tags we studied (i.e., `@dev`, `@notice`, `@param`, and `@return`), the category of partially missing comments can be further subdivided into three types. We exclude the `@dev` tag due to its inherently high subjectivity. Developers tend to embed a wide range of information within this tag, which may not be directly related to the function. As shown in Fig. 4, the internal pure function `toEthSignedMessageHash` takes a bytes32 input named hash and returns a bytes32 value. The `@dev` tag in the NatSpec comment explains that the function replicates the behavior of the `eth_sign JSON-RPC` method referenced in the Ethereum wiki. This additional context establishes a connection between the implemented function and the widely recognized Ethereum `JSON-RPC` method, a connection that may not be readily apparent from the code alone.

a) Completely missing comments: Consider a scenario, as shown in Fig. 5, where a blockchain-based on our finding voting system relies on a smart contract, and this contract includes a function to set a new oracle address. The function `setOracle(address newOracle)` is a public function that takes an address parameter `newOracle` and returns the updated oracle address. The code first checks if the sender of the transaction is the owner of the `AORISEATSADD` contract using the require `(msg.sender`

```
function setOracle (address newOracle) public returns (address) {
        require(msg.sender == AORISEATSADD.owner());
                oracle = newOracle
                return oracle;
}
```

Fig. 5. Code example for completely missing comment.

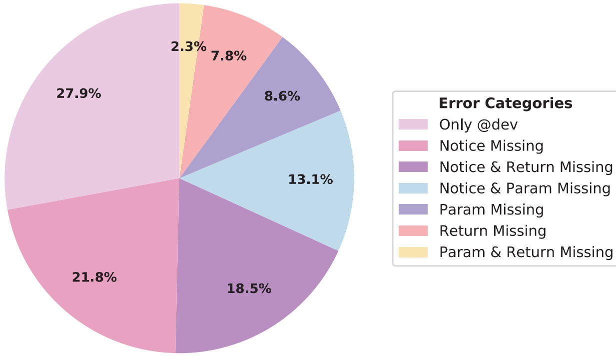

Fig. 6. Distribution of partially missing comments categories.

`== AORISEATSADD.owner())` statement. If the sender is indeed the owner, the oracle address is updated with the new address using `oracle = newOracle`. Finally, the updated oracle address is returned with return oracle.

The absence of comments in this smart contract function poses risks for both developers and users. Developers may find it difficult to understand the function's purpose and logic, hindering maintenance and teamwork. Users may be unclear about the function's objectives, such as updating the oracle address and limiting access to the contract owner, thereby reducing trust and usage. Moreover, the lack of comments could lead to misunderstandings about constraints and intended use, exposing the contract to security vulnerabilities and incorrect usage, ultimately compromising the system's functionality and integrity.

[colback=gray!10, colframe=black, arc=10pt, boxrule=2pt] **Finding 3:** 9,782 functions have completely missing comments, accounting for 58.9% of all functions and 68.0% of missing functions.

b) Partially missing comments: As illustrated in Fig. 6, partially missing comments can be classified into several categories. The most frequently observed case involves the simultaneous absence of `@notice`, `@param`, and `@return` tags, accounting for 27.9% of all partially missing comments. The second most prevalent scenario entails the standalone omission of the `@notice` tag, appearing 1,003 times in the dataset and comprising 21.8% of all partially missing comments. In 19.4% of instances, both the "`@notice`" and "`@return`" tags are jointly absent.

In the example of pull request #93 [26] where only `@notice` exists while others are missing, the changes made to the `onlyFactory` function include adding `@dev` and `@param` tags to provide additional documentation for the initialize function. This function initializes the market and its risk parameters and is called only once by the factory on deployment. The added comments clarify that the function is called upon deployment by the factory contract to set up the market configuration and describe the input parameter `_params` as an array of market parameters documented in the Risk library. These modifications enhance the understandability of the function and its intended use, benefiting both developers and users working with the contract.

[colback=gray!10, colframe=black, arc=10pt, boxrule=2pt]

```
/**
 * @dev Transfers ownership of the contract to a newaccount('newowner').
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner
{
require(newOwner != address (0), "new owner is the zero address");
 transferOwnership(newOwner);
}
```

Fig. 7. Code example for misassigned comments.

Finding 4: We found that the predominant partially missing comment type was the concurrent absence of `@notice`, `@return`, and `@param` tags, constituting 27.9% of all partially missing comment errors. The second most common issue involved the sole omission of `@return` tags, occurring 1,003 times and representing 21.8% of such errors in the dataset.

2) Pattern B: inconsistency comments: Inconsistencies in smart contract comments can be divided into misassigned and low-quality comments. Misassigned tagged comments occur when content intended for one tag, e.g., `@return`, is mistakenly placed under another tag, such as `@dev`. Similarly, low-quality comments may arise when the provided description for a function or a tag does not accurately represent the actual implementation or purpose of the code. Inconsistent comments in smart contracts pose hazards for both developers and users by causing misunderstandings of the contract's functionality, hindering collaboration, reducing trust, and potentially leading to security vulnerabilities or incorrect usage.

a) Misassigned comments: Misassigned comments predominantly occur when `@notice` content is erroneously placed under the `@dev` tag, accounting for 88% of all misassigned comments errors. This highlights the need for an automated comments error correction approach to assist developers with such dilemmas. The provided code snippet in Fig. 7 demonstrates a misassigned comment under the `@dev` tag, which should have been placed under the `@notice` tag. Furthermore, misassigned comments may lead to disorganized documentation when generated, such as content intended for users under the `@notice` tag being misplaced under the `@dev` tag for developers, making it difficult for users to understand the intended information.

[colback=gray!10, colframe=black, arc=10pt, boxrule=2pt] **Finding 5:** We observed that the most prevalent type of misassigned comments is the `@notice` content mistakenly placed under the `@dev` tag, constituting 88% of all identified misassigned errors.

b) Low-quality comments: In addition to misassigned comments, there are also low-quality comments. Despite the relatively lower number, these comment errors still have a strong impact, as users and developers may be misled by incorrect and ambiguous information.

In Fig. 8, the issue of low-quality commenting is evident in the `@notice` tag, which ambiguously states *"Dry run onJoinPool"*, offering little clarity on the function's purpose. The function seems to simulate a join action in a pool, yet the comment fails to elaborate on the meaning of *"Dry run"* or the

```
/**
 * @notice "Dry run" `onJoinPool`.
 */
function queryJoin(address sender, address _onJoinPool, uint256[] memory
balances, bytes memory userData) external override returns
(uint256 bptOut, uint256[] memory amountsIn) {
_queryAction(sender, balances, userData, _onJoinPool);
return (bptOut, amountsIn);
}
```

Fig. 8. Code example for low-quality comments.

roles of userData and balances. A more descriptive `@notice` comment, such as *"Simulates a join action in the pool"* would be more informative. The function appears to simulate a join action in a pool, but the comment doesn't explain what "Dry run" means in this specific context. A more accurate and informative `@notice` comment could be *"Query the balance of the senders in the Joinpool"*. In the given example, the vague `@notice` tag *"Dry run onJoinPool"* could mislead a user into assuming the function is a test run for joining a pool. Unaware of the function's actual dependencies on *userData* and *_onJoinPool*, the user receives inaccurate output values for *bptOut* and *amountsIn*. Relying on these misleading outputs, he proceeds with the actual pool join, only to receive fewer pool tokens than anticipated. This discrepancy poses a financial risk for the user and erodes trust in the smart contract, potentially harming its reputation.

IV. *CETerminator*: Comment Error Detection and Correction

A. Framework Overview

Our empirical study reveals the prevalence of comment errors in smart contracts, which motivates us to develop an approach, *CETerminator*, to automatically detect and correct comment errors. Fig. 9 illustrates our approach's framework, consisting of two primary components: (1) reference comment generator and (2) NatSpec formatter. Initially, the reference generator processes smart contracts and utilizes an LLM with in-context learning to generate concise reference comments for error detection and correction. These results are passed to the NatSpec formatter, where consistency is evaluated. In case of inconsistencies, the formatter corrects them using generated comments and identifies missing elements. When missing elements are found, the reference comments are used for completion, producing the final output. Otherwise, the initial results serve as the corrected comments and are outputted.

Our approach is designed to be compatible with any generative LLM. In this work, we utilize GPT-3.5 [27] due to several reasons. Firstly, GPT-4's API [28], while potentially more powerful, is not currently accessible. Secondly, a study [29] revealed that GPT-3.5's exceptional zero-shot performance on multiple NLP tasks, such as code understanding and text generation, surpasses previous state-of-the-art zero-shot models on several evaluation datasets. This finding showcases GPT-3.5's strong multi-task ability without any fine-tuning or training on specific tasks, making it an ideal choice for our task.

B. Reference Comment Generation

1) Static program analysis: As shown in Fig. 9 and 10, we conduct static program analysis on the input contracts to provide questions in dynamic expression. Solidity-parser-antlr [30], a widely-used parser, is employed to extract the Abstract Syntax Tree (AST) from the input contracts. Upon obtaining the AST, we extract the function name, body, visibility, input parameter list, return variable list, and comments. A comment template is created based on these elements' information, as depicted in Fig. 10. The generated question part comprises a comment template and function body.

2) Prompt generation: Following static program analysis, the AST was augmented with well-commented examples to create structured prompts in reference comments generator as illustrated in Fig. 9. In addition to parsing, we manually select examples that meet the following requirements: 1) functions with complete number of tags, 2) functions with consistent content following each tag. Utilizing these examples, we crafted a multi-round conversational dataset, wherein each round encompasses a question and a corresponding answer, as shown in Fig. 10. Combining instruction, examples and question parts, we generate prompts for the LLM by incorporating a set of rules to facilitate in-context learning. This encompasses conversational context, guided formatting, and example-driven learning, which collectively provide a clear context for the LLM to generate relevant and accurate comments adhering to the desired output format.

Guided formatting is implemented using a template-based approach, defining the desired output format with placeholders like $text$. This directs the LLM to fill in appropriate information while adhering to the specified format. Additionally, example-driven learning is incorporated by providing a correctly formatted NatSpec comment, serving as a reference point for the LLM to generate comments for other functions. The question prompts the model to fill in multiple placeholders present within a comment given a smart contract function, while the answer supplies the necessary content for these placeholders. This tailored format empowers the large language model to efficiently grasp the tasks it needs to accomplish, along with comprehending the appropriate format for the responses.

3) Few-shots learning based on LLM: Our study indicates that a small percentage of smart contracts have comments fully compliant with NatSpec, and crafting such comments manually demands significant time and expertise, making acquiring well-annotated smart contract datasets difficult.

To address this challenge and generate reference comments automatically, we propose a self-training approach. Self-training [31], [32] is a technique where a model iteratively refines its understanding and performance by learning from its own generated output. Our method resembles self-training as it leverages minimal human involvement in providing a few examples as initial assistance. In this task, we utilize GPT-3.5, an advanced large-scale language model, which is designed and trained using causal language modelling (CLM) techniques. In CLM, the primary objective of the model is to predict the next token in a sequence, given the context of all preceding tokens. This methodology enables the model to

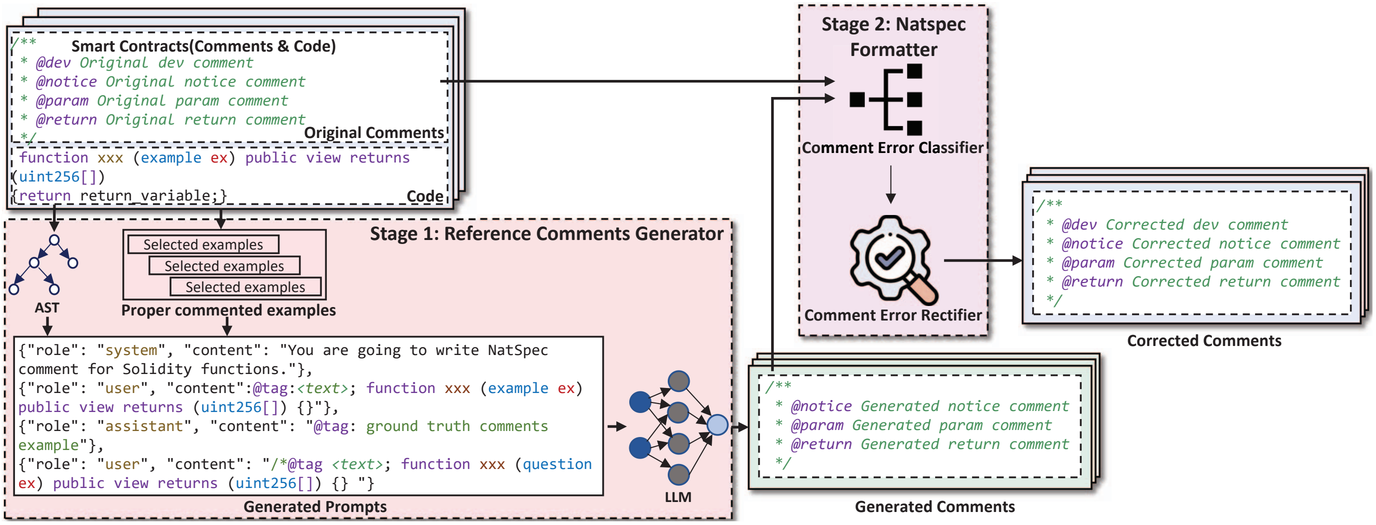

Fig. 9. The overall architecture of *CETerminator*.

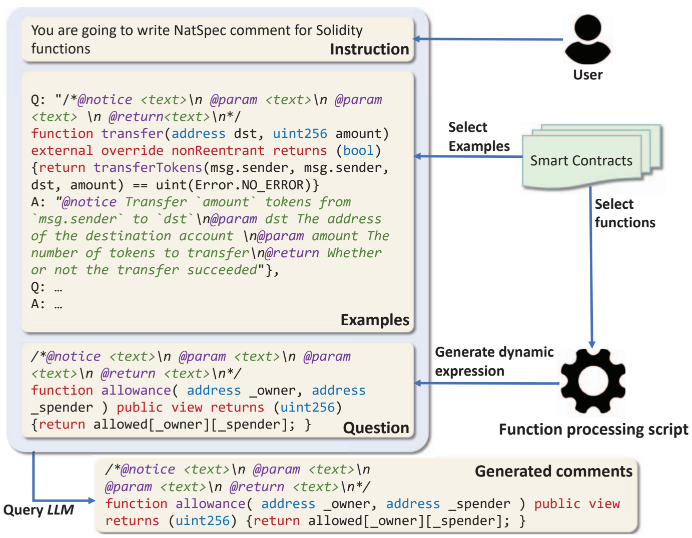

Fig. 10. A prompt example consisting of the instruction, examples and the question.

learn contextual relationships and syntactic structures, facilitating the generation of coherent and contextually appropriate responses.

We feed a few-shots prompt (shown in Fig. 10) to the LLM, using the model completion as the reference comments. Subsequently, we query the LLM to obtain the corresponding comments used as references. The user notice generated by this approach supports the dynamic expression mechanism which is used to dynamically replace corresponding values when end users interact with the contract. In contrast to traditional code comments in other programming languages, the Solidity compiler [33] dynamically constructs user notices from the source code. This dynamic expression mechanism necessitates the alignment of specific terms in the user notice with corresponding tokens in the source code.

C. *NatSpec formatter*

The second part of our framework performed the detection and correction of comment errors (i.e. NatSpec violations). For `@return`, `@notice`, and `@param`, we perform a comprehensive inspection. However, for `@dev`, we only examine content misassignment using `@notice`, according to our study's findings.

Algorithm 1 Inconsistency Error Classifier

$Orig_d, Orig_n, Orig_p, Orig_r, Gen_n, Gen_p, Gen_r$ Err $Err \leftarrow Null$ $Orig_d$ $compare(Orig_d, Gen_n) > \tau$ $Err \leftarrow$ “@notice under @dev” $Orig_p$ $compare(Orig_p, Gen_p) < \tau$ $Err \leftarrow$ “low quality @param”; $Orig_r$ $compare(Orig_r, Gen_r) < \tau$ $Err \leftarrow$ “low quality @return”; $Orig_n$ $compare(Orig_n, Gen_n) < \tau$ $Err \leftarrow$ “low quality @notice”;

1) Comment Error Classifier: The Comment Error Classifier is composed of two components: the first one identifies inconsistency types of comment errors in smart contract comments, while the second one detects missing types of comment errors.

Algorithm 1 illustrates the first part of the Comment Error Classifier. The algorithm takes seven input parameters: $Orig_d, Orig_n, Orig_p, Orig_r, Gen_n, Gen_p$ and Gen_r. These parameters represent the comments under `@dev`, `@notice`, `@param`, and `@return`, as well as respective generated comments of the last three tags. It produces an error type as output, which indicates the category of the comment error.

Initially, the algorithm assigns a $Null$ value to Err (line 1). It then checks if an `@dev` tag is present in the original comment (line 2). If so, it compares the content of the `@dev` tag with the generated `@notice` comment using the Token Overlap Rate (TOR) below.

$$TOR = \frac{W_{both}}{W_{original}}, \tag{1}$$

where W_{both} is the count of words that appear in both the original and generated comments, and $W_{original}$ represents the total number of words in the original comments. If the token overlap rate is larger than the given threshold, τ, it means the notice is misplaced under the `@dev` tag, resulting in an error being reported as *"`@notice` under `@dev`"* (line 4) and updates the error output accordingly. Next, the algorithm examines the existence of `@param`, `@return`, or `@notice` tags in the original comment (line 6-14), respectively. Depending on the tag present, it compares the content of the respective tag with the corresponding generated reference comment. If the token overlap rate is smaller than τ, the algorithm classifies the error as *"low quality `@param`/`@return`/`@notice`"* respectively and updates the error output (line 8-14).

The second component of the Comment Error Classifier is responsible for detecting missing types in smart contract comments. We employ Solidity-parser-antlr, a widely-used parser, to extract the Abstract Syntax Tree (AST) [34] from the contracts. From the AST, we extract the function name, body, visibility, input parameter list, return variable list, and comments. We then utilize regular expressions to verify if the number of comment tags match the information implied by the function.

2) Comment Error Rectifier: Upon identifying error categories, comments are passed to the Comment Error Rectifier module for correction or completion, in line with the findings from our empirical study. The rectification process starts by handling misplaced notices under the `@dev` tag. If detected, the rectifier transfers comments from $Orig_d$ to $Orig_n$ and subsequently removes the $Orig_d$ comments. The module addresses low-quality notice errors by removing comments in $Orig_n$ and handles low-quality param/return errors via eliminating the corresponding comments in $Orig_{p/r}$. Following the previous steps, if any comment tags are missing in the processed comments, the rectifier fills the corresponding comment sections ($Orig_{n/d/p/r}$) with the respective generated comments ($Gen_{n/d/p/r}$).

V. EVALUATION

Due to the 100% accuracy of simple scripts in rectifying inconsistency-related comment errors and detecting missing comment errors, we focus on evaluating our approach's ability to correct missing comment errors in RQ3 and detect inconsistency-related comment errors in RQ4 as follows:

- RQ3 (Missing Correction): What is the performance of our model to correct comment errors of missing?
- RQ4 (Inconsistency Detection): What is the performance of our approach in detecting inconsistency-related comment errors?

We use a 128-core workstation with 282 GB RAM and run Ubuntu 20.04.5 LTS with 7 NVIDIA RTX 3090 GPUs.

A. RQ3: Missing Correction

1) Data collection: To evaluate our approach's ability to detect missing-related comment errors, we extracted 935 $code, tagged\ comment$ pairs, where the comments are in line with NatSpec format, from verified Ethereum smart contracts. We utilized Solidity-parser [35] to parse smart contracts and extract functions, following existing study recommendations.

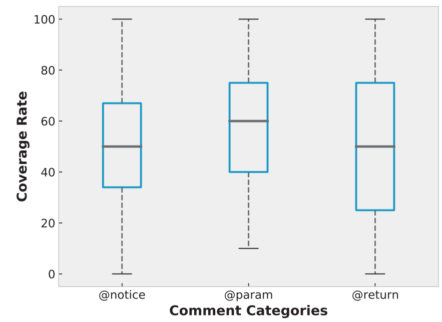

Fig. 11. Token Overlap Rate distribution of the three comment categories.

Regular expressions were employed to supplement Solidity-parser for NatSpec comment extraction.

We employ specific regular expressions to identify functions with complete NatSpec comments, filtering out those with incomplete NatSpec annotations. Upon collecting fully annotated data, we performed a manual analysis to confirm the semantic alignment between each sentence and its corresponding tag. Consequently, we obtained a dataset comprising functions with complete and semantically consistent comments.

In the complete and consistent dataset, we manually excluded instances with poorly named functions and variables. This led to a final set of 312 $code, tagged\ comment$ pairs. Subsequently, we conducted several preprocessing steps to refine the comments.

First, we removed redundant, non-informative phrases like *"Allow to..."* and *"Function to..."* to emphasize the semantic information within a sentence. This step is crucial in minimizing data noise and enhancing text analysis efficiency. Next, we removed all non-letter characters, which lack semantic information, to further cleanse the comments and streamline input for subsequent analysis. Then we removed stop words, which hold minimal semantic meaning, such as *the*, *from*, *to*, *a*, and *is*. Finally, we applied stemming to reduce words to their root or base form, facilitating more accurate text analysis by eliminating variations of the same word (e.g., converting "running," "runs," and "ran" to "run"). This step unifies similar terms and enhances our method's performance.

2) Evaluation Metrics: In this study, we opted for the Token Overlap Rate (TOR) mentioned in Eq. 1 instead of the Bilingual Evaluation Understudy (BLEU) to evaluate our performance of smart contracts comment generation. The primary reasons for this choice include the unsuitability of BLEU for our task as it relies on fixed n-gram lengths, while generated comments can have varying lengths. In contrast, TOR does not impose any length restrictions, allowing for a more accurate evaluation.

3) Result Analysis: TOR primarily assesses the presence of original comment words in the generated comments. Since the prompt restricts generated comments to a single sentence, there is a lower likelihood of redundancy. Our manual checks showed that only 4% of the generated results had a redundancy level above 60%, and these redundancies were closely related to the function content. Thus, they did not significantly impact the semantic clarity or developers' and users' understanding of the function. Furthermore, while BLEU measures n-gram precision, it may not capture the essence of the comments' meaning and can be more sensitive to word order and phrasing differences.

The evaluation results for token overlap rate distribution across the three comment categories—`@notice`, `@param`, and `@return`—demonstrate the effectiveness of our proposed method in generating high-quality comments similar to the original ones for completing missing comments.

Fig. 11 displays the token overlap rates for three comment categories. The `@param` category exhibits the highest median token overlap rate (approx. 60%) and an interquartile range (IQR) of 40%-75%, indicating close alignment with original comments and reflecting the high quality of the generated comments. The `@notice` category displays a median token overlap rate of approximately 50% with an IQR of 34%-67%. The lower median and wider distribution compared to the `@param` category are attributed to the complexity and diversity of `@notice` comments. The `@return` category exhibits an IQR of 25%-75%, highlighting a considerable portion of generated comments being reasonably similar to the original comments, demonstrating our method's effectiveness across all categories.

4) Human Study: Although the token overlap rate as an automatic metric can assess the difference between generated and human-authored ground truth comments, it may not accurately represent human perceptions of the generated comments. We adopt the approach of Hu et al. for a human evaluation, focusing on three aspects: *Similarity* between the generated user documentation and reference documentation, *Naturalness* and *Informativeness* (the extent to which generated comments can explain clearly the purpose or functionality of the code or parameter to audience). Scores are assigned on a 0 to 5 scale, with higher values indicating better results.

We recruited four volunteers with 1-3 years of experience in blockchain or smart contracts development and strong English proficiency to evaluate the generated comments. They assessed a random selection of 100 smart contract functions, accompanied by human-written reference documentation. Participants scored each sample for similarity, naturalness, and informativeness, using integer values from 0 to 5. During the annotation process, participants were allowed to search the Internet for relevant information and unfamiliar concepts.

Fig. 12 presents the distribution of similarity, naturalness, and informativeness scores for the generated comments. In terms of similarity, 38.5% (154) of comments score a 4, indicating a strong resemblance to human-written comments, while 42.25% (169) score a 3, suggesting significant similarity. No comments score a perfect 5, likely due to minor variations in wording or structure. Overall, 80.75% (323) of generated comments are deemed similar to human-written ones.

For naturalness, 34.5% (138) of comments score a 4, reflecting a high level of natural language use. Additionally, 36% (144) score a 3, indicating moderately natural language. Only 3% (12) achieve a perfect score of 5, demonstrating indistinguishable language from human-written comments. Consequently, 73.5% (294) of generated comments exhibit naturalness comparable to human-written ones.

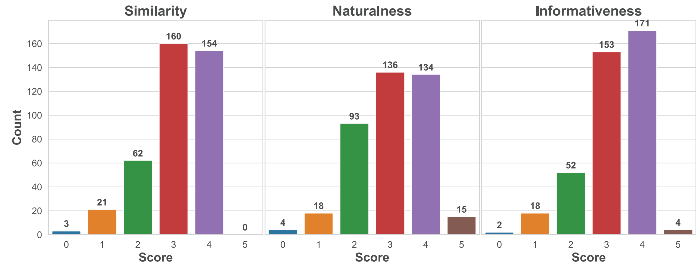

Fig. 12. The count of Similarity, Naturalness and Informativeness scores of the generated comments

Regarding informativeness, 43.5% (174) of comments score a 4, implying nearly equal information content to human-written comments. Furthermore, 35% (140) score a 3, denoting slightly lower information levels. Notably, 1.75% (7) achieve a perfect score of 5, potentially surpassing human-written comments in information content. In total, 80.25% (321) of generated comments are considered informative and comparable to human-written ones.

B. RQ4: Inconsistency Detection

1) Data collection: For the dataset employed in Research Question 4, we adopted a process akin to the one previously described, with a few distinctions. Instead of extracting a specific number of $code, comment$ pairs, we randomly selected 268 functions from Ethereum smart contracts. Based on our empirical study results, we manually annotated the dataset by categorizing comments under various tags into different labels, such as `@notice` (True or Low quality), `@dev` (True or Inconsistent), `@param` (True or Low quality), and `@return` (True or Low quality). Ultimately, we acquired a dataset comprising 1065 $code, tagged\ comment$ tuples, which we employed to address Research Question 4.

2) Evaluation Metrics: To gauge the approach's capability in detecting comment errors of inconsistency, we rely on *precision*, *recall*, and *F1-score*. Considering the discrepancies between the labels of tags in original comments and their corresponding tags in generated comments, we computed the true positives *(tp)*, false positives *(fp)*, and false negatives *(fn)* for the low-quality or inconsistent comments across various tags. Subsequently, we determined the precision *(P)*, recall *(R)*, and F1-score *(F1)*.

3) Baseline: The baseline approach utilizes GPT-3.5, excluding Chain of Thought and omitting multi-round conversational QA examples from the prompt, concentrating solely on comment consistency and quality testing.

TABLE I
EVALUATION RESULTS OF DETECTING INCONSISTENCY-RELATED COMMENT ERRORS

2***Tags**	***CETerminator***			***GPT-3.5***		
(r)2-4 (r)5-7	P	R	F1	P	R	F1
TranNo1	84.85%	82.35%	83.58%	51.11%	47.92%	49.46%
TranNo2	81.01%	83.12%	82.05%	37.14%	46.43%	41.27%
TranNo3	85.71%	90.91%	88.23%	42.55%	47.62%	44.94%
TranNo4	89.53%	89.53%	89.53%	51.16%	45.83%	48.35%
Average	85.28%	86.48%	85.85%	45.49%	46.95%	46.01%

4) Result analysis: Table I presents a comparative analysis of the *CETerminator* and baseline models for transaction volume classification across four categories (0∼100, 100∼10k, 10k∼100k, and 100k+), utilizing performance metrics such as precision, recall, and F1-score. The *CETerminator* model consistently outperforms the baseline across all transaction volume ranges, peaking in the 100k+ category with a precision of 88.51%, recall of 89.53%, and F1-score of 89.02%. In contrast, the baseline model exhibits lower and more varied scores, with its best performance in the 100k+ range at 51.16% precision, 45.83% recall, and 48.35% F1-score. On average, the precision, recall, and F1-scores for addressing inconsistency comment errors are 85.28%, 86.48%, and 85.85%, respectively, exceeding the baseline by 39.79%, 39.53%, and 39.84%. The data suggests that the *CETerminator* model's superior performance over the baseline is due to the application of heuristic rules in its methodology, which is evidenced by the consistently higher precision, recall, and F1-scores across all transaction volume categories.

VI. Discussions

Threats to validity. Potential threats may impact the validity of our study. One threat is the representativeness of our dataset used in the empirical study. To reduce this threat, we employ stratified sampling based on the transaction volumes of the contracts. Another threat lies in the effectiveness of our automatic evaluation metric, TOR, in gauging the quality of generated comments. To mitigate this threat, we conducted a supplementary human evaluation, in which four co-authors independently assessed the generated comments based on three criteria: *Similarity*, *Naturalness*, and *Informativeness*.

Limitations. Our work presents two limitations. Firstly, our focus is limited to comment errors at the function granularity, despite NatSpec's applicability to events, interfaces, and variables, where comment errors may also arise. Secondly, our approach targets only error comments deviating from NatSpec in Ethereum's Solidity language. However, other blockchain platforms (e.g., Cardano with Plutus, Solana with Rust) utilize alternative languages, and our technique does not address comment errors in these contexts. We aim to address these limitations in future work.

VII. Related works

Smart Contract. There are several studies on comments or api documentation in smart contracts [36]–[39]. Hu *et al.* proposes an approach, called SMARTDOC, for user notice generation [40]–[43], a natural language description of smart contract functions, using a neural machine translation model with an attention mechanism. Yang *et al.* [44] presents a Multi-Modal Transformer-based code summarization approach [45] for smart contracts, which leverages multiple modalities to learn from both the source code and natural language descriptions, resulting in code summaries for developers' understanding. Both of their studies found that comments are crucial for user's and developers' understanding of smart contracts. Clear and informative comments can improve the transparency and trustworthiness of smart contract transactions. Zhu *et al.* [46] proposes a tool called DocCon that identifies inconsistencies between Solidity smart contract libraries and their API documentation. The tool uses a fact-based approach to query precomputed facts about the API code and documentation to detect inconsistencies of different severity levels. Different from prior studies, in this paper, we focus on manually studying the prevalence, categories and severity of comment errors, ie. NatSpec violation, and propose an LLM-based technique to detect and correct them automatically.

Large Language Models Deng *et al.* proposes FuzzGPT, a novel technique that leverages large language models to prime LLMs [47] to synthesize unusual programs for fuzzing, with the aim of finding more bugs in DL libraries. Wei *et al.* [48] presents ChatIE, a prompt-based [49] framework that transforms the zero-shot [50] information extraction task into a multi-turn [51] question-answering problem, using ChatGPT to extract structured data from unannotated text. Deng *et al.* [52] proposes a new approach using large language models for fuzzing deep learning libraries [53], which can generate high-quality seed inputs and guide the generation towards a higher number of unique library API usages [54] and valid/diverse DL programs. Our method, *CETerminator*, distinguishes itself from these approaches by comparing the comments generated by LLM through in-context learning [55] with original comments to detect and correct comments not compliant with NatSpec in smart contracts.

VIII. Conclusion

This paper is the first to systematically investigate comment errors in smart contracts, which cause confusion to both developers and end-users. We reveal the pervasiveness of the comment errors in smart contracts and their common patterns through an empirical study of 253 verified contracts with 16,620 functions, which are released as the first dataset for comment error correction and detection in smart contracts to encourage further research. Based on our findings, we introduce *CETerminator*, an innovative automated technique that detects and rectifies comment by comparing the comments generated by a large language model through in-context learning and the original comments. Our evaluation demonstrates the effectiveness of *CETerminator* in addressing comment errors.

References

[1] T. Hu, B. Li, Z. Pan, and C. Qian, "Detect defects of solidity smart contract based on the knowledge graph," *IEEE Transactions on Reliability*, 2023.

[2] S. Ji, S. Zhu, P. Zhang, H. Dong, and J. Yu, "Test-case generation for data flow testing of smart contracts based on improved genetic algorithm," *IEEE Transactions on Reliability*, vol. 72, no. 1, pp. 358–371, 2022.

[3] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[4] S. A. Abeyratne and R. P. Monfared, "Blockchain ready manufacturing supply chain using distributed ledger," *International journal of research in engineering and technology*, vol. 5, no. 9, pp. 1–10, 2016.

[5] F. Schär, "Decentralized finance: On blockchain-and smart contract-based financial markets," *FRB of St. Louis Review*, 2021.

[6] P. Dunphy and F. A. Petitcolas, "A first look at identity management schemes on the blockchain," *IEEE security & privacy*, vol. 16, no. 4, pp. 20–29, 2018.

[7] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[8] , https://etherscan.io/chart/etherprice, accessed: April 23, 2023.

[9] M. Alharby and A. Van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

[10] E. Foundation, "Solidity Documentation: NatSpec Format." [Online]. Available: https://solidity.readthedocs.io/en/v0.5.10/natspec-format.html

[11] Hardhat, "Hardhat runner plugins," 2021, available at: https://hardhat.org/hardhat-runner/plugins. [Online]. Available: https://hardhat.org/hardhat-runner/plugins

[12] NodeFactoryIo, "Solidity comments core," https://github.com/NodeFactoryIo/solidity-comments-core, 2021, accessed: April 23, 2023.

[13] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Financial Cryptography and Data Security: FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers 20*. Springer, 2016, pp. 79–94.

[14] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[15] P. Gaži, A. Kiayias, and D. Zindros, "Proof-of-stake sidechains," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 139–156.

[16] Doxygen, "Doxygen:Doxygen." [Online]. Available: https://www.doxygen.nl/

[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[18] Y. Chen, R. Zhong, S. Zha, G. Karypis, and H. He, "Meta-learning via language model in-context tuning," *arXiv preprint arXiv:2110.07814*, 2021.

[19] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling bert for natural language understanding," *arXiv preprint arXiv:1909.10351*, 2019.

[20] G. Lample and A. Conneau, "Cross-lingual language model pretraining," *arXiv preprint arXiv:1901.07291*, 2019.

[21] T. Linzen, "How can we accelerate progress towards human-like linguistic generalization?" *arXiv preprint arXiv:2005.00955*, 2020.

[22] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[23] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning," in *International conference on learning representations*, 2017.

[24] J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, "A tale of two cities: How webview induces bugs to android applications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 702–713.

[25] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 125–135.

[26] "Issue #93 of v1-core," https://github.com/overlay-market/v1-core/pull/93/files Accessed April 12, 2023.

[27] OpenAI. Models - OpenAI API. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5

[28] ——. GPT-4 API waitlist. [Online]. Available: https://openai.com/waitlist/gpt-4-api

[29] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung *et al.*, "A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity," *arXiv preprint arXiv:2302.04023*, 2023.

[30] , https://github.com/solidity-parser/antlr, accessed: April 23, 2023.

[31] B. Zoph, G. Ghiasi, T.-Y. Lin, Y. Cui, H. Liu, E. D. Cubuk, and Q. Le, "Rethinking pre-training and self-training," *Advances in neural information processing systems*, vol. 33, pp. 3833–3845, 2020.

[32] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," *arXiv preprint arXiv:2304.02014*, 2023.

[33] Learn Solidity: What is the Solidity compiler? [Online]. Available: https://www.alchemy.com/overviews/solidity-compiler

[34] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.

[35] , https://pypi.org/project/solidity-parser/., accessed: April 23, 2023.

[36] X. Hu, Z. Gao, X. Xia, D. Lo, and X. Yang, "Automating user notice generation for smart contract functions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 5–17.

[37] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2019.

[38] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintaining smart contracts on ethereum: Issues, techniques, and future challenges," *arXiv preprint arXiv:2007.00286*, 2020.

[39] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2874–2891, 2020.

[40] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

[41] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[42] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 349–360.

[43] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[44] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 1–12.

[45] S. Appalaraju, B. Jasani, B. U. Kota, Y. Xie, and R. Manmatha, "Docformer: End-to-end transformer for document understanding," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 993–1003.

[46] C. Zhu, Y. Liu, X. Wu, and Y. Li, "Identifying solidity smart contract api documentation errors," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[47] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for gpt-3?" *arXiv preprint arXiv:2101.06804*, 2021.

[48] "Zero-shot information extraction via chatting with chatgpt," 2023.

[49] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," *arXiv preprint arXiv:2104.08786*, 2021.

[50] R. L. Logan IV, I. Balažević, E. Wallace, F. Petroni, S. Singh, and S. Riedel, "Cutting down on prompts and parameters: Simple few-shot learning with language models," *arXiv preprint arXiv:2106.13353*, 2021.

[51] Z. Zhang, J. Li, P. Zhu, H. Zhao, and G. Liu, "Modeling multi-turn conversation with deep utterance aggregation," *arXiv preprint arXiv:1806.09102*, 2018.

[52] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," 2023.

[53] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 995–1007.

[54] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 631–642.

[55] S. Min, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Noisy channel language model prompting for few-shot text classification," *arXiv preprint arXiv:2108.04106*, 2021.