



Why and how developers maintain smart contracts

Giovanni Rosa¹ · Simone Scalabrino¹ · Stefano Mastrostefano^{1,2} · Rocco Oliveto¹

Accepted: 7 March 2025 / Published online: 24 March 2025
© The Author(s) 2025

Abstract

Smart contracts, *i.e.*, self-executing contracts written in code, have gained popularity in recent years due to the introduction of blockchain technology. These contracts are executed automatically when certain conditions are met, and, once deployed, they can not be modified. This presents issues when errors are found or updates are needed. Previous research has mainly focused on introducing approaches and tools for detecting bugs or vulnerabilities in smart contracts. However, it is unclear if these are the only maintenance-related operations developers perform. In this paper, we aim to understand *why* and *how* developers maintain smart contracts. We run a qualitative analysis on 590 commits from 14 open-source smart contract repositories written in Solidity, the most popular programming language for smart contracts. We analyze the commit messages, related issues, and the changes made to understand what triggered changes. Then, we examine how developers changed the source code. As a result, we define two taxonomies: one reporting the reasons for the maintenance and one regarding the patterns of modifications. Our findings suggest that smart contract maintenance is often focused on improving the internal quality of the scripts (40% of the cases), and that many changes aim to fix bugs despite the several approaches available for detecting them beforehand.

Keywords Empirical study · Smart contract · Solidity

Communicated by: Grischa Liebel

✉ Simone Scalabrino
simone.scalabrino@unimol.it

Giovanni Rosa
giovanni.rosa@unimol.it

Stefano Mastrostefano
stefano.mastrostefano@enea.it

Rocco Oliveto
rocco.oliveto@unimol.it

¹ University of Molise, Pesche, Italy

² Nuclear Department (NUC), ENEA C.R., Frascati, Italy

1 Introduction

Smart contracts are self-executing contracts in which the terms of the agreement among the parties are directly written into lines of code. Such programs can be conceptually developed and deployed in several contexts. However, they have recently proliferated thanks to the introduction of blockchains and, specifically, of some blockchain platforms, such as Ethereum (Wood 2014). Such platforms allow developers to easily deploy smart contracts, which almost immediately become available to all their users. Another feature that makes smart contracts appealing on blockchain is the decentralization nature of the latter: There is no need of a centralized authority that manages the process at hand and that has the responsibility of checking that the terms of the contract are fulfilled. Blockchains, indeed, provide a certain degree of guarantee that the distributed ledger can not be tampered with and, specifically, the blockchain platforms that allow the deployment of smart contracts provide guarantees that code is executed exactly as is.

Given their nature of contracts, smart contracts should not be changed without an agreement among the parties and, in fact, they can not be modified once they get deployed. If a smart contract with a bug or vulnerability is deployed, it could financially harm the end-users. The most notorious example is the one that brought to the DAO attack in the Ethereum blockchain, used to steal 3.6M ETHs (~\$70 million) (Falkon 2017). For this reason, the literature mainly focused on introducing approaches and tools for detecting errors in smart contracts before they are deployed. Between 2015 and 2020, 68 papers introduced approaches for detecting functional and security-related issues in smart contracts, most of which rely on static code analysis (Piantadosi et al. 2023). For example, Bai et al. (2018) used formal methods to detect functional issues, while Kolluri et al. (2019) introduced an approach based on dynamic symbolic execution to detect event-ordering bugs (a class of vulnerabilities).

Despite their nature of contracts, smart contracts are still *software* and, thus, they intrinsically need to evolve. For example, the programming language or the blockchain platform might introduce new features, or, again, developers might spot bugs or vulnerabilities only *after* the smart contract has been deployed in some instances and they want to avoid that the erroneous version is deployed in other contexts. This calls for the question: Why and how do developers maintain smart contracts? Recently, Chen et al. (2021) tried to answer this question by performing a Systematic Literature Review based on 131 papers on smart contract maintenance, complemented by a survey with developers. However, little is known about what developers do *in practice* while maintaining smart contracts. Answering this question would allow researchers to have a more complete view of what developers need, so that they can define new approaches aimed at *globally* supporting the maintenance and evolution of such software products.

In this paper, we run an empirical study in which we aim at filling this gap. We analyze the evolution history of 14 open-source repositories containing smart contracts written in Solidity, the most popular programming language for smart contracts. We select all the changes that are focused on Solidity source files and we extract information regarding (i) what triggered the change, by looking at the commit messages and at possible related issues, and (ii) how developers performed the change. In total, we manually analyzed 590 commits of such repositories, representing a statistically significant sample. Then, we use a procedure inspired by card-sorting to define a taxonomy of reasons triggering the changes and a taxonomy of patterns of modifications made.

We found that smart contract maintenance is often aimed at improving the internal quality of the scripts: ~40% of the commits analyzed contained refactoring operations, mostly aimed

at improving the aesthetic of the code (*e.g.*, by introducing comments). This suggests that refactoring and readability improvement and detection tools tailored for smart contracts would benefit developers. We also observed that several changes ($\sim 8\%$) aimed at improving testing, mainly by introducing test cases, while only a minority of approaches are available for supporting developers in this activity (Piantadosi et al. 2023).

Finally, a large amount of commits ($\sim 36\%$) aim at correcting errors, showing either that currently available approaches are not used in practice yet or that they are not able to spot a relevant class of issues.

The remainder of this paper is organized as follows. In Section 2, we introduce some concepts related to the blockchains and smart contracts used throughout the paper. In Section 3, we present the empirical study design. In Section 4, we report the results and present the categories of changes found in our analysis by also reporting examples. In Section 5 we discuss the results and provide possible future directions. We discuss the threats to validity in Section 6 and the related work in Section 7. Finally, we conclude our paper with Section 8.

2 Background

In this section we report some background information (such as the definitions of “blockchain” and “smart contract”) to facilitate the reading of the remainder of this paper.

2.1 Blockchains

Blockchains are decentralized (distributed) digital ledgers that record transactions across a network of computers, also called “nodes”. Such systems are used to create and maintain a continuously growing list of records, called *blocks*. Each block is securely linked to the previous one through cryptography. In public (or “permissionless”) blockchains, all the nodes have the right to write blocks. Thus, there is a risk that two or more nodes write blocks at the same time, thus generating so-called “forks”, *i.e.*, divergent versions of the ledger. To reduce such a risk, blockchains rely on consensus protocols. For example, the “proof-of-work” consensus protocol requires nodes who want to write to provide the proof of having solved a computationally intensive problem easy to verify. By design, a blockchain is resistant to tampering, making it useful for recording events, such as financial transactions, in a transparent and secure way.

2.2 Smart contracts on Blockchains

A smart contract is a computer program that automatically executes the terms of a contract when certain conditions are met. On a blockchain, a smart contract is stored on the distributed ledger and can be programmed to trigger actions, such as the transfer of digital assets, when predefined conditions are met. This allows for the automation of processes and the ability to enforce the terms of a contract without the need for intermediaries. An example¹ of smart contract designed to regulate an auction is depicted in Fig. 1: It collects money (*i.e.*, ETH, in this case) when the bidders make an offer by calling the function `bid`. Then, when the auction terminates, the winner can call the function `auctionEnd` to get the best offer. Other bidders can get a refund for their offers by calling the function `withdraw`.

¹ Adapted from the official documentation, see <https://bit.ly/3iM4616>

```

pragma solidity ^0.8.4;
contract SimpleAuction {
    address payable public beneficiary;
    uint public auctionEndTime;
    address public bestBidder;
    uint public bestBid;
    mapping(address => uint) pendingReturns;
    bool ended;

    event HighestBidIncreased(address bidder, uint amt);
    event AuctionEnded(address winner, uint amt);

    // Creates a new auction
    constructor(uint t, address payable b) {
        beneficiary = b;
        auctionEndTime = block.timestamp + t;
    }

    // Users bid (called with money attached)
    function bid() external payable {
        if (block.timestamp > auctionEndTime)
            revert AuctionAlreadyEnded();

        if (msg.value <= bestBid)
            revert BidNotHighEnough(bestBid);

        if (bestBid != 0) {
            pendingReturns[bestBidder] = bestBid;
        }
        bestBidder = msg.sender;
        bestBid = msg.value;
        pendingReturns[bestBidder] = 0;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    // Users get back their bid (e.g., called by users
    // that have not won the auction to get back their
    // money).
    function withdraw() external returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            pendingReturns[msg.sender] = 0;
            if (!payable(msg.sender).send(amount)) {
                pendingReturns[msg.sender] = amount;
                return false;
            }
            return true;
        }
    }

    // The beneficiary gets their money.
    function auctionEnd() external {
        if (block.timestamp < auctionEndTime)
            revert AuctionNotYetEnded();
        if (ended)
            revert AuctionEndAlreadyCalled();

        ended = true;
        emit AuctionEnded(bestBidder, bestBid);

        beneficiary.transfer(bestBid);
    }
}

```

Fig. 1 Example of smart contract for regulating an auction

2.3 Ethereum and Solidity

Ethereum (Buterin et al. 2014; Wood 2014) is, currently, one of the most popular blockchain platforms for the creation of smart contracts (Serebryantseva 2022). Differently from other popular blockchain platforms (such as Bitcoin), the Ethereum Virtual Machine (EVM), used to run smart contracts in Ethereum, supports a Turing-complete instruction set. At a low level, the programming language that the EVM understands is a stack-based bytecode language. However, several high-level programming languages have been introduced to write smart contracts, the most popular of which is Solidity.

Solidity is a statically typed contract-oriented programming language that provides tools and features that allow developers to define the logic and rules of the contract, as well as specify the conditions under which it can be executed. In addition, Solidity also provides a way to define the state of the contract, the data that it holds, and the functions that can be performed on the data. The previously shown example (Fig. 1) is, indeed, in Solidity.

To discourage the deployment of malicious smart contracts (*e.g.*, for spam) and to reduce the risk of denial-of-service (DoS) attacks, the execution of Ethereum smart contracts entails the consumption of “gas.” Gas refers to the amount of computational effort required to execute a particular operation or set of operations in a smart contract, and it is measured in units called “wei”. Each operation in a smart contract has a gas cost associated with it, and the total gas cost of executing a contract is the sum of the gas costs of all the operations it performs. When a user sends a transaction to a smart contract or, in general, executes a function, they must also include a gas price and a gas limit, which determine the maximum amount of gas that the contract can consume in order to execute the transaction. Gas is also used to prevent infinite loops and other forms of unintended computation.

3 Study Design

The *goal* of our study is to identify the reasons behind changes in smart contracts and the patterns of such modifications. The *context* consists of 590 commits extracted from 14 open-source Solidity repositories.

Our study is steered from the following research questions (RQs):

RQ₁: *Why do developers maintain smart contracts?* This first research question focuses on the self-declared reason *why* developers perform maintenance activities in smart contracts (*e.g.*, to fix a bug).

RQ₂: *How do developers modify smart contracts?* The second research question aims to analyze *how* developers modify the smart contracts to achieve their maintenance goal. Such an RQ is focused on finding common ways in which the intention is implemented as code improvement.

3.1 Study Context

To select an appropriate sample of smart contracts and, most importantly, their revision history, we relied on GitHub. More specifically, we used the GitHub search feature to get a list of repositories that contained files written in Solidity. We filtered out repositories with less than 100 stars to avoid toy projects or personal repositories. With the same aim, we also excluded archived repositories. Then, we randomly selected a sample of repositories, using a uniform distribution, for which we extracted all the commits involving Solidity files (*i.e.*, with

“.sol” extension). In particular, only the *main* or *master* branches were considered. Since we aim to manually investigate the entire commit history of each repository manually, we excluded from the sample the repositories having more than 200 commits. We manually inspected each repository to check if it was mainly aimed for smart contract development. We discarded, at this stage, the repositories that only collaterally contained smart contracts. For example, we discarded *TheAlgorithms/Solidity* since it contains examples of algorithms and data structures implemented in Solidity. In the end, we selected 14, for a total of 590 commits. We reported them in Table 1, along with the number of commits selected.

As for sample significance, our aim was to obtain a sufficiently large set of commits that allowed at to have most a $\pm 5\%$ margin of error with 95% confidence level. To find out the minimum number of commits to analyze to meet this requirement, we used the following formula, which allows to compute the sample size for an unknown population (Cochran 1977):

$$S = 0.25 \frac{Z_{\alpha}^2}{E^2}$$

where Z_{α} is the value of the Z distribution for the confidence level (0.95, in our case, which leads to $Z_{\alpha} = 1.96$), and E is the margin of error (0.05). Thus, we needed to include at least 384 commits. We obtained a significant sample size (590 commits) by selecting 14 repositories.

3.2 Experimental Procedure

For both the research questions, two authors (annotators) independently assigned a set of tags to each commit. Since assigning tags requires some degree of understanding of how each repository works, we analyzed one repository at a time (*i.e.*, we did not shuffle the commits).

As for RQ₁, the annotators read both the commit messages and the possibly linked GitHub issues or pull requests. Examples of tags assigned are “refactoring” and “optimize gas.” Note that, in this process, we almost exclusively relied on the explicit information provided by the developers: If the commit message was not clear and the commit was not linked to any issue or pull request we did not try to guess the reason of the change from the code change itself. The only exception is constituted by unambiguous changes: For example, a change in which a developer only adds a comment is clearly aimed at improving the maintainability of the smart contract. In the end, we were able to identify a reason for all the commits, except for few cases (*e.g.*, commit `aa237f` of BSC Genesis Contract: They are probably part of a bigger change but they are not explicitly referencing any issue/pull request clearly expressing the intentions, and thus we preferred not to guess the category. We excluded those commits also for RQ₂. The same is for merge commits (*i.e.*, 16 commits in total), because they can contain changes already tagged from previous commits.

As for RQ₂, instead, the annotators focused on the code changes (*i.e.*, modifications) made to the Solidity code. Examples of tags assigned are “add comment” and “rename variable.”

At this stage, the first annotator assigned a total of 650 (RQ₁) and 764 tags (RQ₂), while the second one assigned 685 tags (RQ₁) and 780 tags (RQ₂). A third annotator (author) checked the consistency among the tags made by the first two annotators for each commit (for both the RQs) to cross-validate the cases of conflicts. If any inconsistency was spotted, the three authors re-analyzed and discussed these cases together, aiming at reaching consensus.

This happened for at least a tag (considering both the research questions) for 149 commits ($\sim 25\%$). Table 2 depicts the number of tags assigned by the two annotators for each repository and the final number of tags determined after discussion when inconsistencies

Table 1 The repositories containing Solidity smart contracts involved in our study

Repository	Description	# Commits
transmissions11/solmate	Building blocks for smart contract development	167
bnb-chain/bsc-genesis-contract	The genesis contracts of BNB Smart Chain	70
enjin/erc-1155	A sample implementation of ERC-1155	59
Uniswap/v2-periphery	Smart contracts for interacting with Uniswap V2	58
cheapETH/bridge	Bridge smart contracts between L1 (Ethereum) and L2 (cheapETH)	53
BitGo/eth-multisig-v2	Multi-Sig Wallet v2 smart contract implementation with additional confirmAndExecute improvements	45
ourzora/nft-editions	Lazy-mint editioned smart contracts for ERC721s	35
marbleprotocol/flash-lending	Smart contracts for arbitrage opportunities within the Ethereum network	33
Jeiwan/flash-loans-comparison	Comparison of flash loan solutions on Ethereum	22
paco0x/amm-arbitrageur	An arbitrage bot between Uniswap AMMs	16
xtblock/xtt	XTblock Token smart contracts implementation	13
wighawag/template-ethereum-contracts	Examples of Ethereum smart contracts	14
ethernity-cloud/pox-smart-contract	Proof of execution Ethernity Cloud smart contract	3
fifikobayashi/Flash-Arb-Trader	Smart contract using flash liquidity for arbitrage between Sushiswap and UniswapV2	2

were found. Note that the final number can of tags can be higher than the sum of number of tags independently assigned by the two annotators: Sometimes, indeed, both the annotators were right and, thus, all their tags were kept. The level of agreement between the two annotators resulted in a Cohen's Kappa of 0.78 for RQ₁ and 0.92 for RQ₂. The obtained score, according to the interpretation recommendation (Cicchetti and Sparrow 1981), is considered “excellent” for both of them.

After having collected the definitive set of tags for each commit, three of the authors performed a card sorting activity (Spencer 2009) aimed at merging semantically similar tags and categorizing them. Card sorting is a technique used to help design information structures. The process starts by grouping the similar tags into categories, based on their similarity. The process is iterative, and the authors discuss the categories to ensure that they

Table 2 Number of tags independently assigned by the annotators (A1 and A2), and total number of tags after the inconsistency check (Total) for each repository

Repository	RQ ₁ (Why)			RQ ₂ (How)		
	A1	A2	Total	A1	A2	Total
transmissions11/solmate	191	194	194	228	228	228
bnb-chain/bsc-genesis-contract	86	92	92	118	118	118
Uniswap/v2-periphery	58	66	66	55	63	63
enjin/erc-1155	59	60	60	63	63	63
cheapETH/bridge	53	54	55	52	54	54
BitGo/eth-multisig-v2	44	48	47	53	58	58
marbleprotocol/flash-lending	44	47	47	64	64	64
ourzora/nft-editions	35	38	39	37	37	37
Jeiwan/flash-loans-comparison	22	22	22	23	24	24
wighawag/template-ethereum-contracts	20	20	20	25	25	25
paco0x/amm-arbitrageur	16	18	18	18	18	18
xtblock/xtt	14	14	14	14	14	14
ethernity-cloud/pox-smart-contract	6	7	7	12	12	12
fifikobayashi/Flash-Arb-Trader	2	2	2	2	2	2

are coherent and that they cover all the tags. The obtained categories are then grouped into higher-level categories, and the process is repeated applying several refinements until the involved authors reach a consensus. This activity, done for both research questions, resulted in two distinct taxonomies: One aimed at describing the reasons behind the maintenance activity performed (RQ₁, developers' intent), and one aimed at characterizing the type of modifications performed (RQ₂, code changes applied by developers).

To answer RQ₁, we discuss the first taxonomy and, specifically, each category identified regarding the reasons behind the modifications. To answer RQ₂, we do the same (but, this time, for the modifications patterns). Additionally, we report different sub-taxonomies regarding the four macro-categories of maintenance reasons identified in the first taxonomy (*i.e.*, *Adaptive*, *Corrective*, *Perfective*, and *Preventive* maintenance) to understand what patterns are used for the different maintenance activities. To this end, we only considered non-tangled commits (*i.e.*, we do not include commits that are assigned with more than a tag in RQ₁).

4 Results

In this section, we report the results of our analysis and the answers to our two research questions.

4.1 RQ₁: Why Developers Maintain Smart Contracts

We report in Fig. 2 the taxonomy of reasons why developers maintain smart contracts, along with the absolute number of commits interested in those changes. Note that the sum of the macro-categories is larger than the number of commits we analyzed. This is because a relatively large percentage of commits (~16%, *i.e.*, 92 out of 590 92/) were assigned with

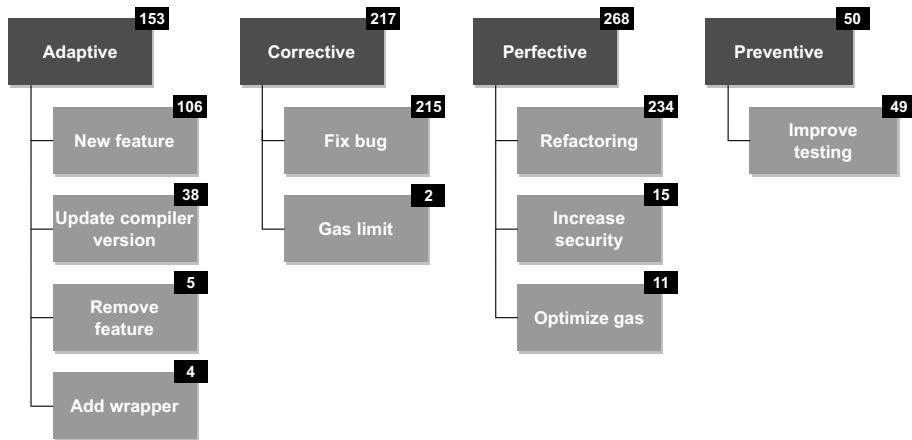


Fig. 2 Taxonomy of the reasons for maintaining smart contracts

more than one tag. Such a prevalence of possibly *tangled* commits is in line with the one previously observed by Herzig and Zeller on Java projects (Herzig and Zeller 2013).

We identified four macro-categories of reasons, based on the classic types of maintenance activities that exist for general software systems (ISO/IEC 14764:2006 2006): *adaptive*, *corrective*, *perfective*, and *preventive*. It is worth noting that some of those terms are not universally linked to a specific meaning (Sommerville 2011). Therefore, we explicitly report what we mean with each category:

- *Adaptive* maintenance: Changes aimed at adapting the smart contract to a new or changed environment; we used this category for new feature implementation.
- *Corrective* maintenance: Changes aimed at removing implementation errors from the smart contract.
- *Perfective* maintenance: Changes aimed at maintaining the functionality of the smart contract by improving non-functional attributes of the smart contract (e.g., maintainability or security).
- *Preventive* maintenance: Changes aimed at preventing the introduction of errors in the future (e.g., the introduction of test cases).

A first clear result is that the large majority of changes are *perfective* (39.0%) and *corrective* (31.5%), while *adaptive* (22.2%) and *preventive* (7.0%) changes are less frequent. Again, some commits are tangled, thus the sum is not 100%. We describe below the four macro-categories we identified, ordered by frequency.

4.1.1 Perfective Maintenance

Refactoring (234 changes) is, surprisingly, the most frequent reason why developers maintain smart contract. In this context, we categorized as refactoring also changes aimed at improving the readability of the smart contracts (e.g., by introducing comments or changing identifiers).

A first example is depicted in Fig. 3, which is an excerpt of a commit for the project Flash Lending². The previous version of the documentation comment only contained a gen-

² Commit 5b1eb1 of Flash Lending: <https://bit.ly/3QPHY2n>.

```

/**
 * @dev Deposit tokens to the bank.
+ * @param token Address of token to deposit. 0x0 for ETH
+ * @param amount Amount of token to deposit.
 */
function deposit(address token, uint256 amount)
    external onlyOwner payable {
    transferFrom(token, msg.sender, this, amount);
}

```

Fig. 3 Example of refactoring operation

```

if (value > 0 && curBurnRatio > 0) {
-   uint256 toBurn = value * curBurnRatio / BURN_RATIO_SCALE;
+   uint256 toBurn = value.mul(curBurnRatio).div(BURN_RATIO_SCALE);
    if (toBurn > 0) {
        address(uint160(BURN\_ADDRESS)).transfer(toBurn);
        emit feeBurned(toBurn);

-   value = value - toBurn;
+   value = value.sub(toBurn);
    }
}

```

Fig. 4 Example of security-improving commit

```

+ // Storing these outside the loop saves 15 gas per iteration.
+ uint256 id;
+ uint256 amount;

for (uint256 i = 0; i < idsLength; ) {
-   uint256 id = ids[i];
-   uint256 amount = amounts[i];
+   id = ids[i];
+   amount = amounts[i];
}

```

Fig. 5 Example of gas optimization commit

```

function checkValidatorSet(Validator[] memory
    validatorSet) private pure
    returns(bool, string memory) {
+   if (validatorSet.length > MAX_NUM_OF_VALIDATORS){
+       return (false, "the number of validators exceed the limit");
+   }
}

```

Fig. 6 Example of bug fixing commit

eral definition of what the function does, while the commit introduces a description of the two function parameters (`token` and `amount`). In another commit of the same project³, a developer renamed a contract from `Bank` to `BankInterface`. Similarly to the previous example, also in this case developers aimed at improving the code understandability by choosing a more representative identifier.

A minority of commits (15) are aimed at **increasing the security** of the smart contracts. Figure 4 reports an excerpt from one of those commits made for the `bsc-genesis-contract` repository.⁴ In this case, the patch updates the arithmetic operations to use the `SafeMath` library, which is a common practice in Solidity to avoid arithmetic overflows and underflows.

Two perfective commits are specifically aimed at introducing the “Safe math” library (Fig. 9a). Such a component allows to automatically handle numeric errors such as integer overflow/underflow: When such situations occur, an exception is thrown.

Finally, in only 11 commits (*i.e.*, 4%) the developers modified the smart contracts to **optimize gas consumption** and, thus, their performance. Figure 5 shows an example of such a change from the `Solmate` repository.⁵ Developers moved the declaration of two variables outside the `for` loop to “*save ~15 gas*,” as they explicitly report in a comment.

4.1.2 Corrective Maintenance

Bug fixing is, clearly, the only corrective maintenance-related activity, and it occurred very frequently, as previously mentioned (215 commits). An example of bug-fixing commit is provided in Fig. 6 from the BSC Genesis Contract.⁶ In this case, a preliminary check is added to a function that controls a set of validators: If the size of the given validator set exceeds the maximum amount allowed for this smart contract, the function returns `false`. Note that this is a specific requirement of BSC Genesis: It would be hard for any automated approach to spot those kind of bugs.

4.1.3 Adaptive Maintenance

Differently from the two previous categories, adaptive changes are more balanced in the sub-categories of our taxonomy. The **introduction of a new feature** in the contract is the most frequent reason for adapting them (106 commits). For example, in commit 4e3235 of XTT,⁷ developers introduced the interface of ERC-20 (a standard for implementing fungible tokens).

Developers also frequently need to adapt their smart contracts to **new compiler versions** (38 commits). Being it relatively new, the Solidity language is constantly evolving. Thus, several features are being added and removed. Thus, developers sometimes need to adapt their code to new language/compiler versions or constraint the compilation to a specific range of versions. This happens, for example, in commit cc75056 of

³ Commit 6a999e of Flash Lending: <https://bit.ly/3HdKFHP>.

⁴ Commit 3152a05 of bsc-genesis-contract: <http://bit.ly/3T8FqPK>.

⁵ Commit 2a3bd8 of Solmate: <https://bit.ly/3ZNs0de>.

⁶ Commit 2f10b7 of BSC Genesis Contract: <https://bit.ly/3GQcvJ5>.

⁷ Commit 4e3235 of XTT: <https://bit.ly/3QNSI1g>.

```
// SPDX-License-Identifier: UNLICENSED
-pragma solidity ^0.8.13;
+pragma solidity ^0.8.16;
```

Fig. 7 Example of version update

flash-loans-comparison⁸, also reported in Fig. 7: Developers updated the version constraint of the contract to enable the build for newer versions of the Solidity compiler.

Finally, in a minority of cases (only 4 commits) developers introduced wrappers that allow for the interaction with a decentralized exchange (*i.e.*, a marketplace where users buy and sell cryptocurrencies) in a simpler way. An example is provided in commit 4e6cdf of Flash Lending,⁹ in which developers included a wrapper for EtherDelta.¹⁰

4.1.4 Preventive Maintenance

The only specific way in which developer make preventive maintenance is by improving the tests of the smart contracts (49 commits). For example, in Solmate — commit 9202f8,¹¹ developers introduced several tests for new methods introduced in a utility library named SafeCastLib.

4.1.5 Mapping Issues to the Literature

Chen et al. (2021) provide a taxonomy of well-known issues in the literature regarding smart contract maintenance. In Table 3, we provide a mapping between the reasons why developers maintain smart contracts (from our results) and the categories of issues identified by in the literature by Chen et al.. The two most basic and generic reasons for maintaining smart contracts (*i.e.*, *new feature* and *fix bug*) are clearly independent from smart contract-specific issues. On the other hand, we could trace back all the other categories to such root problems.

We found two reasons possibly related to the *scalability issues* of Solidity (*e.g.*, lack of useful libraries and APIs and limited support for handling memory). First, developers might update the compiler version: A newer Solidity version might provide better support to developers. Second, they might remove features altogether (*e.g.*, to save memory for other core features). As previously mentioned, the introduction of wrappers is often done to make the smart contract more compatible with other systems, which partially compensates for the *lack of standards*. The *difficulty of handling the gas system* reported in the literature can traced back to two reasons for maintaining smart contracts: (i) fixing gas-related bugs, and (ii) preemptively optimizing the gas usage. Analogously, the *lack of advanced SE approaches and research data* probably causes the many refactoring operations we observed and the changes aimed at improved testing. Refactoring, however, might also be so frequent because of the *lack of high quality reference code*, which makes smart contracts developers constantly find themselves facing sub-optimal code they want to improve. Finally, the *unpredictability of callee contracts* mostly causes changes aimed to improve security (*e.g.*, fixing reentrancy vulnerabilities).

⁸ Commit cc75056 of flash-loans-comparison: <https://bit.ly/4bJKaCP>.

⁹ Commit 4e6cdf of Flash Lending: <https://bit.ly/3XJlrWp>.

¹⁰ <https://etherdelta.com/>

¹¹ Commit 9202f8 of Solmate: <https://bit.ly/3krUIWe>.

Table 3 Mapping between the maintenance reasons found in our study and the issues from the literature identified by Chen et al. (2021)

Maintenance Reason	Issue from the Literature (Chen et al. 2021)
Adaptive	
New feature	—
Update compiler version	Scalability Issues
Remove feature	Scalability Issues
Add wrapper	Lack of Standards
Corrective	
Fix bug	—
Gas limit	Difficulty of Handling the Gas System
Perfective	
Refactoring	Lack of SE Approaches & Research Data Lack of High Quality Reference Code
Increase security	Unpredictable Callee Contracts
Optimize gas	Difficulty of Handling the Gas System
Preventive	
Improve testing	Lack of SE Approaches & Research Data

Answer to RQ₁

Most of the changes in smart contract maintenance and evolution are perfective and, specifically, aimed at improving the internal quality of the code through refactoring operations. A similarly large quantity of commits are aimed at fixing bugs, while adaptive and preventive changes are less frequent.

4.2 RQ₂: How Developers Maintain Smart Contracts

Figure 8 reports the taxonomy of source code modifications performed by smart contract developers. Most of the commits (291, *i.e.*, 37%) contain changes aimed at modify the code structure. A lower amount of commits (247, *i.e.*, 32%) resulted in implementation-related changes (*i.e.*, changes aimed at modifying the logic of the smart contract) and, surprisingly, a similar amount of commits (*i.e.*, 243) contains aesthetic changes (*i.e.*, changes aimed at making the code look better by improving identifiers, comments, etc.) Also in this case, the sum is larger than 100% because, clearly, a commit might contain more than an operation (*e.g.*, rename function and move code).

We provide more details about those categories below.

4.2.1 Structural Changes

The most frequent structural change is the creation of new functions (99 commits). Some of them are test cases (in 15 commits). In 56 cases developers moved code from a function to another or from a contract to another, or directly add a new contract (58 cases). With a lower

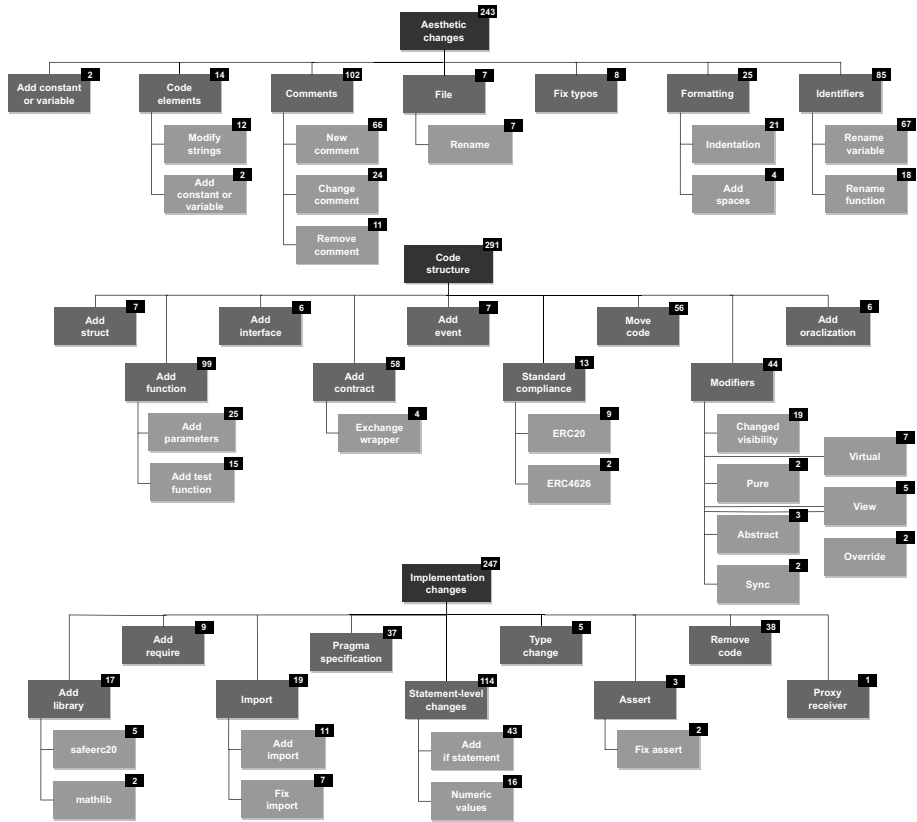


Fig. 8 Taxonomy of changes made for maintaining smart contracts

frequency, instead, developers change the function modifiers (44 commits). Solidity offers a wide variety of modifiers, some of them specific for smart contracts. For example, `payable` indicates that the function can be called by attaching an arbitrary amount of money. In most of the cases (19 commits), developers simply change the function visibility (e.g., public to private). Finally, other less frequent structural changes are the introduction of oraclization, which allows to interact with the web (6 cases), new interfaces (6 cases) and structures (7 case).

Interestingly, in 13 commits, developers introduced standards in their smart contracts. ERC-20 is the most frequently implemented one (9 commits). Specifically, it is used for implementing fungible tokens in Ethereum. Other standard implementations we found are ERC-4626 for tokenized vaults (2 commits) and ERC-721 for non-fungible tokens, or NFTs (in a single commit).

4.2.2 Implementation-related

Most of the changes aimed at modifying the logic of the smart contracts were at **implementation statements level** (114). More specifically, 43 of such commits consisted in the introduction of `if` statements, while 16 of them in the modification of numeric values in the code.

Another typical implementation-related modification is the **change to pragma statements** (37). Such statements allow developers to configure compiler features and, in most of the cases, it is used to specify the compiler versions that are compatible with the smart contract at hand.

Finally, other relevant changes regard the **introduction of libraries** (17), which usually occurs by copying the entire library code inside the repository, and the addition of `import` statements for including local and external files in a given contract (19).

4.2.3 Aesthetic Changes

Comments are the most frequently modified elements in smart contracts (102 commits). In most of the cases, developers introduce new comments (66 commits), while they rarely remove them (11 cases). Identifiers are also often modified (85 commits): In this case, most of the changes (67) are aimed at renaming variables, while a lower amount of them change the function names (18). Finally, in a (relative) minority of cases (25), developers re-format the code by fixing indentation (21 cases) or adding spaces between tokens, *e.g.*, between identifiers and operators operator (4 cases).

4.2.4 Change Patterns by Maintenance Type

Figure 9 depicts the taxonomies of change patterns by type of changes (only on non-tangled commits). *Adaptive* commits mostly modify the code structure (113 commits) and the implementation (77 commits). The most frequent operation is the introduction of new contracts (46 cases), followed by adding new functions (42) and updating the `pragma` statement (36 cases). Similarly, *corrective* commits mostly change the implementation (143 cases) and, less frequently, the structure (113 cases). While doing so, in 2 cases, developers also introduced constant or variables, which we categorized as aesthetic changes. Most of the bug fixes regard, as it could be expected, the implementation logic, and thus they were corrected by performing statement-level changes (98 cases). Also, 15 fixes involved a change of function visibility. An example of such cases can be found in commit 6f00c0 of BSC Genesis Contract:¹² Some constants and variables are made publicly visible to fix a compile error.

As it could be partially expected from the results of RQ₁, most of the *perfective* commits (209) included aesthetic changes, while a minority of them included structural changes (83). Since almost the totality of aesthetic changes we found are in perfective commits, the same conclusions we drawn for the aesthetic changes in general apply also here. Finally, *preventive* commits, as expected, mostly aim at adding test functions (24 cases).

Answer to RQ₂

Implementation-related, aesthetic and structural changes are almost equally distributed in smart contract maintenance and evolution. Also, aesthetic changes are almost always related to perfective maintenance.

¹² Commit 6f00c0 of BSC Genesis Contract: <https://bit.ly/3DkDyLJ>.

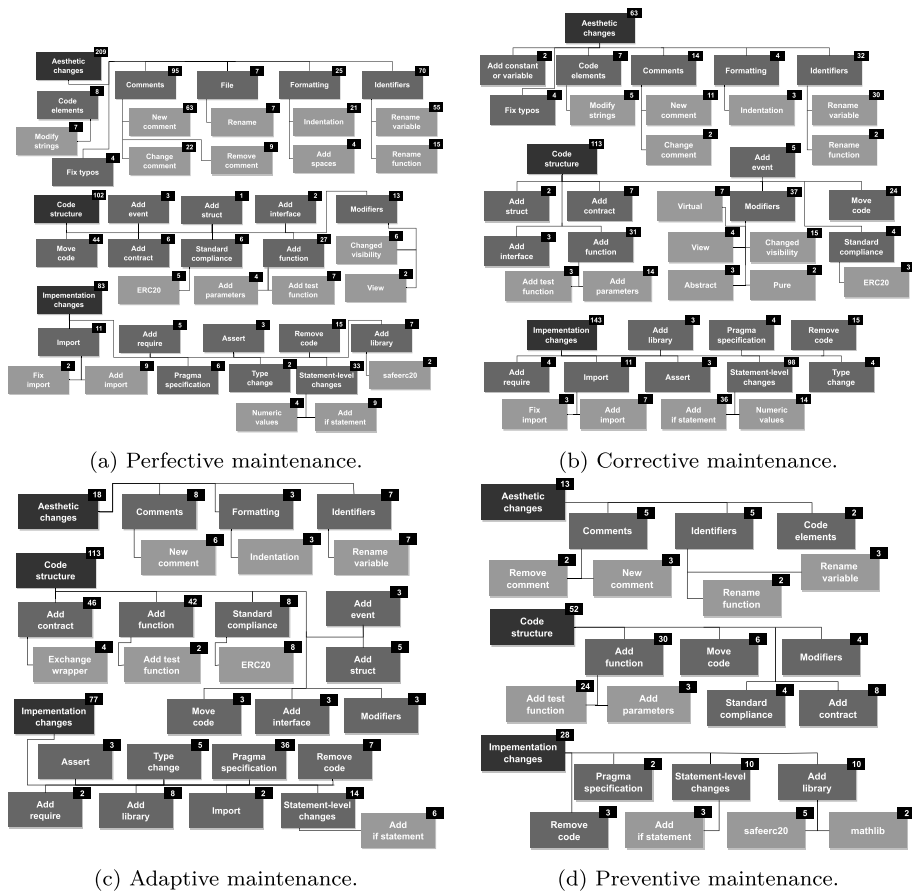


Fig. 9 Taxonomies of change patterns made for maintaining smart contracts divided by maintenance type

5 Discussion

Our results have several implications for researchers and practitioners. We report in this section the lessons learned for researchers and the implications of our results directly usable by practitioners.

5.1 Lessons Learned for Researchers

Researchers can get several insights from our results on the maintenance activities performed on 14 open-source smart contract repositories.

First, several approaches have been investigated in the literature for finding a wide range of defects and security issues early in smart contracts (Di Angelo et al. 2023; Piantadosi et al. 2023; Di Angelo et al. 2024; Vidal et al. 2024). In addition, there are recently proposed approaches able to perform automated repair of smart contracts (Gao et al. 2024; Wang et al. 2024). Our results confirm that this research trend is well justified: A large percentage of commits in smart contract repositories are aimed at fixing bugs. Our results show that either

the existing tools are still not enough to fully prevent bugs or that they are still not popular among developers.

Lesson Learned #1. New research is needed to understand why developers need to fix bugs in smart contracts, and if such bugs can be avoided by using state-of-the-art approaches and tools. Also, there is a need of better understand the developers' needs in that direction to propose better tools to improve the quality of the developed smart contracts.

One of the most interesting results we obtained is that *perfective* changes and, specifically, refactoring operations are very frequent in smart contract development. Most of the changes we observed in RQ2 are *aesthetic*, thus suggesting that developers want to improve the readability/understandability of their code. While previous work proposed approaches for estimating the readability of smart contracts, this research area is still new and not much explored. Examples of that are tools proposed in other contexts, such as measuring source code readability (Scalabrino et al. 2018), or to detect (Ksontini et al. 2024) and suggest refactoring operations (Rosa et al. 2024) for Dockerfiles. Such approaches could be adapted to smart contracts to help developers in their maintenance operations.

In general, the results of RQ₂ should be considered when proposing new approaches for assisting smart contract developers. For example, our results suggest that tools that help developers fix import (7 occurrences) might not be as useful as tools that help them write, change, or remove comments (102 occurrences).

Lesson Learned #2. Future work should explore the introduction of approaches for partially automating the most common refactoring operations typically performed on smart contracts, including handling comments (e.g., code summarization), identifiers (e.g., renaming operations), and move code operations.

While we observed that relatively few commits were aimed at improving testing, it is clear that it is not rare that developers write test cases for smart contracts. At the moment, there are a few approaches for automatically generating test cases for smart contracts (Driessen et al. 2021; Fooladgar et al. 2021; Li et al. 2019; Olsthoorn et al. 2022b, a; Wang et al. 2019), but, again, this area is under-explored, considering the developers' interest.

Lesson Learned #3. Future work should explore more in-depth the possibility of automatically generating test cases for smart contracts.

Finally, we observed that one of the most frequent adaptive changes involves updating the compiler version in a *pragma* statement. This pattern suggests that developers might benefit from a tool able to identify the correct Solidity version (or range of versions) for a given contract. Similar approaches exist in different contexts (e.g., for detecting compatibility issues in Android APIs Li et al. 2018; Scalabrino et al. 2020; Wei et al. 2016), and an adaptation might benefit Solidity developers.

Lesson Learned #4. Approaches for detecting and resolving compatibility issues in Solidity might benefit developers.

5.2 Implications for Practitioners

Given the large prevalence of corrective maintenance, we suggest practitioners to consider defining and constantly adopting better pipelines of available tools to find common errors earlier. For example, developers could adopt *GasReducer* (Chen et al. 2018) and *GasChecker* (Chen et al. 2020) to find gas-related issues, *Mythril* (Mueller 2018), *Slither* (Feist et al. 2019), and *Smartcheck* (Tikhomirov et al. 2018) to find vulnerabilities, and *SynTest-Solidity* (Olsthoom et al. 2022b) to generate test cases. Note that there are plenty of other tools that can support practitioners in detecting issues (Piantadosi et al. 2023). Besides, this research field is quite active. Thus, we suggest developers to keep themselves up to date and read research papers, even if previous work already reports that most of them already do it (Chen et al. 2021).

Besides, the high frequency of perfective maintenance operations might be a symptom of poor or insufficient attention by developers to internal quality. Note that smart contracts are relatively small and have shorter histories than other kinds of software projects. Therefore, we would have expected much fewer refactoring operations. We recommend developers to constantly check the internal quality of the smart contracts. It would be ideal if specialized tools to do that existed. To the best of our knowledge, there are only a few of such tools, such as *iSCREAM* (Canfora et al. 2021), which detects unreadable smart contracts. Nevertheless, we argue that this can be done manually at a small cost, given the limited size of most smart contracts.

6 Threats to Validity

Construct Validity A possible limitation is related to the filter we used to select candidate repositories to analyze, which is based on their popularity (*i.e.*, number of stars). In the process, we might have excluded unpopular repositories that might have been interesting to analyze. On the other hand, it is worth noting that we found this criterion quite effective to select sufficiently relevant repositories, despite the small number of valid commits in some of them (*e.g.*, only two for *Flash Arb Trader*). Also, a proper sampling strategy would have required to randomly select the commits from *all* the extracted repositories. This, however, is not feasible since, as previously mentioned, tagging commits requires some degree of knowledge on the whole repository. Another possible limitation of our analysis is the threshold used to filter out repositories during the sample selection. We selected repositories with no more than 200 commits to ensure that the manual analysis was feasible. This criterion could have resulted in the exclusion of more mature repositories with a higher number of commits. To understand the possible impact of this threat, we analyzed all the repositories containing smart contracts from GitHub that have more than 100 stars using the same methodology we used in the study and analyzed the distribution of commits. The third quartile (Q3) of the distribution is 294.5. Note that such results represent an upper bound since we did not filter out repositories based on their type in such an analysis (*i.e.*, we did not manually exclude tools and collections of repositories, which likely have a larger number of commits as compared to proper smart contracts). This shows that the threshold we adopted still allows us to include typical smart contract repositories. However, our results could lack some aspects that could be only observed by studying the more mature repositories. While we excluded archived repositories, two of the repositories we considered have been later archived by the respective owners (*i.e.*, *paco0x/amm-arbitrageu* and *BitGo/eth-multisig-v2*). This

shows that replicating our study in the future could imply the selection of different repositories. This, however, is true for most mining studies that rely on evolving properties of the repositories for the selection (including the number of stars or number of commits).

Internal Validity It is possible that some of the tags we assigned, and the defined taxonomies, may contain some bias due to the subjectivity of the manual evaluation. This risk is particularly present in this analysis because the developers of some considered repositories did not always report clear commit messages or insightful pull-request information. Besides that, the information in the commit messages and related issues and pull requests might be unreliable in the first place. For example, some information might be missing and there might be inconsistencies between the change and the intention reported in the message. To mitigate this threat, we performed some basic sanity checks on the consistency between declared rationale (in the commit message) and the code change performed. For example, we made sure that there was no commit aimed at fixing bugs that simply modified comments or renamed variables. We did not find any inconsistency. To mitigate these risks, two annotators independently tagged each entry, and a third annotator intervened when we found conflicts. If we were not entirely sure of a tag, we discarded the entire commit. This happened in three cases, only for RQ₁. For RQ₂, instead, tagging was always possible since code diffs were (obviously) always available. In this way in case we discarded the information reported by developers for the cases in which there are ambiguous, incongruent, or have missing details.

External Validity The conclusions we drew from our results could be incomplete because we analyzed a sample of commits and a sample of repositories. As for the latter, we are quite confident that our sample of repositories is representative: It is also worth noting that the repositories we selected are quite diverse in terms of popularity: The least popular repository we considered `cheapETH/bridge`¹³ has ~110 stars, while the most popular one (`transmissions11/solmate`¹⁴) has more than 3.7k stars. To reduce the threat related to the sample size of the number of analyzed commits, we made sure to select a sample large enough to ensure an acceptable margin of error (< 5%). Thus, while it might be possible that a replication of this study results in different percentages, it is doubtful that the main message (and, specifically, the lessons learned) drastically change. On the other hand, we only analyzed 14 smart contract repositories, and we limited our analysis to a programming language (Solidity) for a specific blockchain platform (Ethereum). Thus, our results might not generalize to other platforms or languages. Finally, since the technologies we focused on in this paper are relatively young, it is possible that, in time, the way developers maintain and evolve smart contracts change.

7 Related Work

Previous work evaluated the development, maintenance, and security of smart contracts.

Piantadosi et al. (2023) performed a literature review on the detection of bugs and vulnerabilities in smart contracts. The results show that most of the approaches proposed in the literature are based on static analysis and focused on the assessment of security weaknesses. Also, most of them are not released as prototype tools.

¹³ <https://github.com/cheapETH/bridge>

¹⁴ <https://github.com/transmissions11/solmate>

Another relevant study is the literature review by Vacca et al. (2021). They evaluated 96 papers related to blockchain and smart contracts research. Most of them are related to the Ethereum platform. The most investigated topics are security and finding bugs, including testing, where only fuzz-testing is investigated. Other topics related to testing, such as test case generation, integration, and regression testing, are almost unexplored. There is a lack of supporting tools for developers, such as the detection of code patterns related to high-gas consumption. Also, they did not report the existence of refactoring tools.

Chen et al. (2021) performed a systematic literature review focused on maintenance activities for smart contracts. They evaluated four types of software maintenance activities (*i.e.*, corrective, adaptive, perfective, and preventive) analyzing 131 research papers and complementing their finding with a developer survey. As a result, they identified a total of 9 types of issues related to maintenance. Some of them are the lack of mature tools, low readability, and, in general, poor code quality for Ethereum open-source smart contracts combined with a lack of reference examples from Solidity. Our study is complementary to the one performed by Chen et al. and aims at studying the same phenomenon (maintenance of smart contracts) with a completely different methodology. Specifically, while Chen et al. (2021) rely on the scientific literature and personal opinions of the developers (collected in a survey) to extract the maintenance activities, we performed an in-depth analysis of the code changes performed in open-source smart contracts repositories *in practice*. While we confirm some of their findings (*e.g.*, there is a need of tools to support testing and auditing of smart contracts), our results provide a more specialized taxonomy reporting the specific types of change performed by developers, such as fixing bugs and improving the readability of the code, and their frequency (which can not be reliably done with a survey).

Other studies investigated the development activities and challenges by performing developer surveys and interviews. For example, Zou et al. (2019) performed an empirical study to explore the current state of development and challenges for Ethereum smart contracts. They performed a semi-structured interview with 20 experienced smart contract developers from industry and open-source, and a survey with 232 practitioners to validate their findings. Among their results, they also report that the existing tools are very basic for both development and maintenance activities.

Kannengießer et al. (2021) conducted a developers survey on challenges in smart contract development, including maintainability-related ones that are related to the activities of updating, adding functionality, correcting flaws, and improving code efficiency. Their results report, for Ethereum, the main challenges are related to its maintainability issues due to the immutability of a smart contract once deployed.

While these studies are related, more in general, to tools and approaches for the detection of issues in smart contracts, they confirm our findings in terms of a lack of advanced tools to support test case generation and other testing activities. They also show the limits of the current research on smart contracts, where there are still lowly explored topics (*e.g.*, those related to testing activities and refactoring).

It is worth saying that, recently, new approaches have been proposed that can support maintenance activities. For example, in the context of automated test generation, Driessen et al. (2021) proposed the tool AGSolT to automatically generate test cases for Solidity smart contracts using a random testing approach (*i.e.*, a fuzzer), and a guided-search approach (*i.e.*, DynaMOSA Panichella et al. 2017). Moreover, Olsthoorn et al. (2022a) proposed a novel approach for the generation of test cases for smart contracts, integrated with their testing suite, SynTest-Solidity (Olsthoorn et al. 2022b), which outperforms the existing state-of-the-art.

Other tools are more related to code quality, such as the detection of code smells (Chen et al. 2020) and the assessment of code readability (Canfora et al. 2021).

8 Conclusion

In this paper, we qualitatively analyzed the revision history of 14 smart contract repositories, for a total of 590 commits, aiming at understanding why developers modify them and how. We defined a taxonomy of reasons behind maintenance and evolution of smart contracts, and a taxonomy of change patterns, both generic and filtered for the different macro-categories of reasons. Researchers can benefit from such a taxonomy by using it to decide what refactoring activities are more worth supporting. Practitioners, instead, can more informatively decide what maintenance aspects require more attention and properly manage them.

Acknowledgements The authors would like to thank Federica Patriarca (University of Molise, Italy) for performing a preliminary analysis and identifying some of the refactoring operations presented in this work.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Giovanni Rosa, Simone Scalabrino, and Stefano Mastrostefano. The first draft of the manuscript was written by Giovanni Rosa and all authors reviewed and edited the final manuscript, which has been read and approved by all authors.

Funding Open access funding provided by Università degli Studi del Molise within the CRUI-CARE Agreement. This work has been partially supported by the European Union - NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 “DevProDev: Profiling Software Developers for Developer-Centered Recommender Systems”, grant n. 2022S49T4W, CUP: H53D23003610001.

Data Availability Statement To make our results verifiable and replicable, we provide a publicly available replication package (Rosa et al. 2024), which contains all the details of the tags we assigned to each commit.

Declarations

Conflicts of Interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- ISO/IEC 14764:2006 (2006) ISO/IEC/IEEE international standard for software engineering - software life cycle processes - maintenance. Tech rep
- Bai X, Cheng Z, Duan Z, Hu K (2018) Formal modeling and verification of smart contracts. In: Proceedings of the 2018 7th international conference on software and computer applications, pp 322–326
- Buterin V et al (2014) A next-generation smart contract and decentralized application platform. White Paper 3(37):2–1
- Canfora G, Di Sorbo A, Fredella M, Vacca A, Visaggio CA (2021) iSCREAM: a suite for Smart Contract READability assessMent. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp 579–583
- Chen J, Xia X, Lo D, Grundy J, Luo X, Chen T (2020) Defining smart contract defects on ethereum. IEEE Trans Softw Eng
- Chen J, Xia X, Lo D, Grundy J, Yang X (2021) Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges. Empir Softw Eng 26(6):1–44

- Chen T, Feng Y, Li Z, Zhou H, Luo X, Li X, Xiao X, Chen J, Zhang X (2020) Gaschecker: scalable analysis for discovering gas-inefficient smart contracts. *IEEE Trans Emerg Top Comput* 9(3):1433–1448
- Chen T, Li Z, Zhou H, Chen J, Luo X, Li X, Zhang X (2018) Towards saving money in using smart contracts. In: *Proceedings of the 40th international conference on software engineering: new ideas and emerging results*, pp 81–84
- Cicchetti DV, Sparrow SA (1981) Developing criteria for establishing interrater reliability of specific items: applications to assessment of adaptive behavior. *Am J Ment Defic* 86(2):127–137
- Cochran WG (1977) *Sampling techniques*, 3rd edn. John Wiley & Sons
- Di Angelo M, Durieux T, Ferreira JF, Salzer G (2023) Smartbugs 2.0: an execution framework for weakness detection in ethereum smart contracts. In: *2023 38th IEEE/ACM international conference on Automated Software Engineering (ASE)*. IEEE, pp 2102–2105
- Di Angelo M, Durieux T, Ferreira JF, Salzer G (2024) Evolution of automated weakness detection in ethereum bytecode: a comprehensive study. *Empir Softw Eng* 29(2):41
- Driessen S, Di Nucci D, Monsieur G, Tamburri DA, Heuvel WJvd (2021) Automated test-case generation for solidity smart contracts: the agsolt approach and its evaluation. *arXiv preprint arXiv:2102.08864*
- Falkon S (2017) The story of the DAO - its history and consequences. <https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee>
- Feist J, Grieco G, Groce A (2019) Slither: a static analysis framework for smart contracts. In: *2019 IEEE/ACM 2nd international Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, pp 8–15
- Fooladgar M, Arefzadeh A, Faghieh F (2021) Testsmart: a tool for automated generation of effective test cases for smart contracts. In: *2021 11th International Conference on Computer Engineering and Knowledge (ICCKE)*. IEEE, pp 476–481
- Gao C, Yang W, Ye J, Xue Y, Sun J (2024) sguard+: machine learning guided rule-based automated vulnerability repair on smart contracts. *ACM Trans Softw Eng Methodol* 33(5):1–55
- Herzig K, Zeller A (2013) The impact of tangled code changes. In: *2013 10th working conference on Mining Software Repositories (MSR)*. IEEE, pp 121–130
- Kannengießer N, Lins S, Sander C, Winter K, Frey H, Sunyaev A (2021) Challenges and common solutions in smart contract development. *IEEE Trans Software Eng* 48(11):4291–4318
- Kolluri A, Nikolic I, Sergey I, Hobor A, Saxena P (2019) Exploiting the laws of order in smart contracts. In: *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pp 363–373
- Ksontini E, Abid A, Khalsi R, Kessentini M (2024) Drminer: a tool for identifying and analyzing refactorings in dockerfile. In: *Proceedings of the 21st international conference on mining software repositories*, pp 584–594
- Li L, Bissyandé TF, Wang H, Klein J (2018) Cid: automating the detection of api-related compatibility issues in android apps. In: *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pp 153–163
- Li Z, Wu H, Xu J, Wang X, Zhang L, Chen Z (2019) Musc: a tool for mutation testing of ethereum smart contract. In: *2019 34th IEEE/ACM international conference on Automated Software Engineering (ASE)*. IEEE, pp 1198–1201
- Mueller B (2018) Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam* 9(54):4–17
- Olsthoom M, van Deursen A, Panichella A (2022a) Guiding automated test case generation for transaction-reverting statements in smart contracts. In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp 163–174
- Olsthoom M, Stallenberg D, van Deursen A, Panichella A (2022b) Syntest-solidity: automated test case generation and fuzzing for smart contracts. In: *The 44th international conference on software engineering-demonstration track*. IEEE/ACM
- Panichella A, Kifetew FM, Tonella P (2017) Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Trans Software Eng* 44(2):122–158
- Piantadosi V, Rosa G, Placella D, Scalabrino S, Oliveto R (2023) Detecting functional and security-related issues in smart contracts: a systematic literature review. *Software: Practice and Experience* 53(2):465–495
- Rosa G, Scalabrino S, Mastrostefano S, Oliveto R (2024) Replication package for “Why and how developers maintain smart contracts”. <https://doi.org/10.6084/m9.figshare.21919404>
- Rosa G, Zappone F, Scalabrino S, Oliveto R (2024) Fixing dockerfile smells: an empirical study. *Empir Softw Eng* 29(5):108
- Scalabrino S, Bavota G, Linares-Vásquez M, Piantadosi V, Lanza M, Oliveto R (2020) Api compatibility issues in android: causes and effectiveness of data-driven detection techniques. *Empir Softw Eng* 25(6):5006–5046

- Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D (2018) A comprehensive model for code readability. *J Softw: Evol Process* 30(6):e1958
- Serebryantseva V (2022) What are the top smart contract platforms and how to choose the right one? <https://pixelplex.io/blog/smart-contract-platforms/>
- Sommerville I (2011) *Software engineering* 9th edition. ISBN-10 137035152
- Spencer D (2009) *Card sorting: designing usable categories*. Rosenfeld Media
- Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y (2018) Smartcheck: static analysis of ethereum smart contracts. In: *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, pp 9–16
- Vacca A, Di Sorbo A, Visaggio CA, Canfora G (2021) A systematic literature review of blockchain and smart contract development: techniques, tools, and open challenges. *J Syst Softw* 174:110891
- Vidal FR, Ivaki N, Laranjeiro N (2024) Vulnerability detection techniques for smart contracts: a systematic literature review. *J Syst Softw*, 112160
- Wang C, Zhang J, Gao J, Xia L, Guan Z, Chen Z (2024) Contracttinker: Llm-empowered vulnerability repair for real-world smart contracts. *arXiv preprint arXiv:2409.09661*
- Wang X, Wu H, Sun W, Zhao Y (2019) Towards generating cost-effective test-suite for ethereum smart contract. In: *2019 IEEE 26th international conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp 549–553
- Wei L, Liu Y, Cheung SC (2016) Taming android fragmentation: characterizing and detecting compatibility issues for android apps. In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pp 226–237
- Wood G et al (2014) *Ethereum: a secure decentralised generalised transaction ledger*. Ethereum Project Yellow Paper 151(2014):1–32
- Zou W, Lo D, Kochhar PS, Le XBD, Xia X, Feng Y, Chen Z, Xu B (2019) Smart contract development: challenges and opportunities. *IEEE Trans Software Eng* 47(10):2084–2106

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.