A Comparative Gas Cost Analysis of Proxy and Diamond Patterns in EVM Blockchains for Trusted Smart Contract Engineering

No Author Given

No Institute Given

Abstract. Blockchain applications are witnessing rapid evolution, necessitating the integration of upgradeable smart contracts. Software patterns have been proposed to summarize upgradeable smart contract best practices. However, research is missing on the comparison of these upgradeable smart contract patterns, especially regarding gas costs related to deployment and execution. This study aims to provide an in-depth analysis of gas costs associated with two prevalent upgradeable smart contract patterns: the proxy and diamond patterns. The proxy pattern utilizes a proxy pointing to a logic contract, while the diamond pattern enables a proxy to point to multiple logic contracts. A comparative analysis of gas costs for both patterns is conducted and compared to a traditional non-upgradeable smart contract, also known as the classic pattern. From this analysis, a theoretical contribution is derived in the form of two consolidated blockchain patterns and a corresponding decision model.

Keywords: Blockchain · Software Patterns · Upgradeable Smart Contracts · Proxy Pattern · Diamond Pattern

1 Introduction

Smart contracts are pivotal for orchestrating digital transactions in a reliable and secure manner in blockchain platforms [19]. Smart contracts are particularly important on Ethereum and similar blockchain platforms, where they are used in diverse areas including digital finance or industrial traceability.

As blockchain applications evolve, the need for smart contracts to be not only secure but also upgradeable becomes apparent [8]. In this context, a set of upgradeable smart contract patterns, including the proxy and diamond patterns, have surfaced to answer the lack of classical smart contracts' adaptability whose logic cannot be changed once deployed [23, 28]. The proxy pattern uses a proxy contract to delegate calls to an implementation contract, providing a flexible and upgradeable solution. The diamond pattern, introduced in EIP-2535, addresses concerns like contract size limitations and facilitates enhanced maintainability and versioning through multiple implementation contracts.

Research has focused on identifying upgradeable smart contracts pattern families, pointing towards proxy and diamond-based strategies [14, 22, 23, 28].

However, the papers do not provide an in-depth analysis of the functional and non-functional properties of these patterns, nor gas costs behaviors. Hence, a research gap is identified regarding a thorough study of the proxy and diamond patterns, especially in terms of a gas costs analysis.

To address this issue, this paper aims to answer the following research questions: *(RQ1) How do the classic, proxy, and diamond patterns differ in terms of gas consumption, scalability, and ease of use?* And *(RQ2) What implications do these differences have for the development of blockchain applications, considering the traditional classic pattern as a baseline?*

In this paper, we contribute to the literature through a unified approach to compare gas costs in upgradeable smart contracts. We leverage this methodology to provide a comparative gas cost analysis of the proxy and diamond patterns in EVM blockchains, compared against a monolithic non-upgradeable smart contract, which is used as a baseline. Based on these results, we derive a theoretical contribution in the form of two smart contract patterns adhering to the Alexandrian form format following the standard proposed by Christopher Alexander [1]. These patterns include the results of our comparative analysis and contribute to the broader understanding of upgradeable smart contract patterns and their use in blockchain applications. Based on these patterns, a decision model for using these patterns is proposed, emphasizing functional and non-functional properties for each design decision.

The remainder of this paper is structured as follows. Section 2 introduces key concepts related to smart contracts and blockchain patterns, and section 3 presents studies already made on linked concepts. Section 4 presents the method used to carry a comparative analysis on proxy versus diamond gas costs. Section 5 presents the results of the tests made on the different patterns. Section 6 leverages these findings to propose two consolidated proxy and diamond patterns as well as a decision model for using these patterns. Section 7 finally concludes the paper with a summary and a discussion of the results and some considerations for future work.

2 Background

2.1 Smart Contracts

A smart contract is a program hosted on a blockchain network [31]. When a smart contract executes, its updated state (or storage) is registered into a transaction. Then, that transaction is stored in the blockchain ledger making it immutable and tamper-proof [4]. More precisely, the final validated results are stored in a Merkle Patricia trie whose nodes correspond to an account or a smart contract. A smart contract comprises both variables and functions. Smart contract functions can execute arbitrary code, access the state of the variables and optionally update them. The default function's mutability gives the right to read and modify state variables. However, function mutability can be constrained to add more security when it comes to accessing these variables. On the one hand,

a `pure` function cannot read nor modify the state of the contract; it is only used to compute a value, often using the parameters passed to it. On the other hand, a `view` function can read the state of the contract, but cannot modify it.

Smart contracts are immutable, meaning that once deployed, they cannot be modified [7, 32]. This is a security feature, as it prevents malicious actors from modifying the code, so that the contract can be trustable and unbreakable. However, this also means that if a bug is found in the code, it cannot be fixed, and the only solution is to deploy a new contract. Also, if a new feature is added to the contract, the only way to do it is to deploy a new contract. This is a problem for users, as they have to migrate to the new contract, and for developers, as they have to maintain multiple contracts. To address this issue, several patterns have emerged, which are discussed later in the paper [8, 23].

Smart contracts used for this study have been developed using Foundry, a smart contract development framework built in Rust [17]. This toolkit is chosen because it compiles and executes faster than other well-known frameworks such as Truffle and Hardhat. Moreover, it eliminates the need for an additional programming language for writing tests and scripts, unlike Truffle and Hardhat with JavaScript, and generates easily gas reports per smart contract function.

2.2 Gas and Storage in the Ethereum Virtual Machine (EVM)

The EVM is a Turing-complete virtual machine that runs on every node of the Ethereum network and other EVM-based blockchains. It provides a secure and isolated environment for smart contract execution [16]. Gas is the unit of computation in the EVM, and every operation performed by the EVM has a gas cost associated with it [15]. The user pays this cost in the form of gas fees, to the miner/validator who executes the smart contract. It is the miner's incentive to execute the smart contract and record the results on the blockchain, but also a way to prevent spam and denial of service attacks.

Storage in the EVM is linked to the concept of gas, as every operation involving storage – whether it is writing new data or modifying existing data – incurs a gas cost. This cost is proportional to the storage resources consumed, reflecting the principle that the more network resources (like memory and storage space) a transaction uses, the more it needs to compensate the network. Every smart contract has its own storage space, which is isolated and theoretically unlimited.

Smart contracts' storage utilizes a key-value store, with these key-value pairs referred to as storage slots. A key for a storage slot is determined by the index of the slot, which is numbered contiguously from 0 to $2^{256} - 1$. A value is a 32-byte (or 256-bit) word, also called an item. Data smaller than 32 bytes can be packed into a single slot, but if it is larger, the transaction is reverted [12]. Some types of data are stored in multiple slots, such as arrays and mappings which are stored in multiple contiguous slots [13]. Strings are also stored in multiple slots, where one slot stores the length of the string, and the other slots store each 32-byte chunk of the string [11]. A `struct`, which is a collection of variables, is stored in a single slot if it fits, otherwise, it is stored in multiple contiguous slots. The gas cost of writing to storage increases with the number of slots written to. This

means that writing multiple variables, if they fit within a single slot, incurs the same gas cost as writing a single variable.

3 Related Work

Software patterns provide well-tested solutions for frequently encountered application development use cases [2]. Pattern formatting such as the Alexandrian Form Format provides a standardized approach to the presentation of software engineering patterns, including their description, the forces or tradeoffs at stake, the benefits and drawbacks, and their main applications.

A family of software patterns focusing on blockchain patterns has recently emerged in the literature [29, 34]. The later include on-chain data management, domain-based, smart-contract related best practices such as upgradability patterns. Upgradability patterns refer to decentralized application maintenance strategies that can be applied for adding new features or fixing issues are needed [8]. Two recurring upgradability patterns emerge through the literature, namely the proxy and diamond patterns [14, 22, 23, 28].

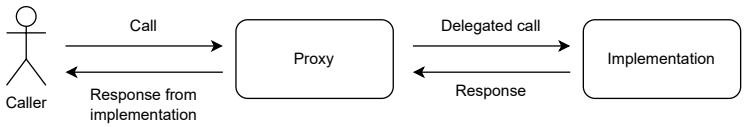


Fig. 1. Proxy pattern diagram.

The proxy pattern, pictured in Figure 1 enables smart contract updates without changing the contract's address or requiring data migration [26]. Comprising two contracts – the user-interacted proxy and the logic-holding implementation – the proxy forwards calls to the implementation via delegated calls. Shared storage ensures that if the implementation is updated, the storage persists. To update the implementation, a new contract is deployed, and the proxy is directed to it, eliminating the need for user migration and preventing data loss. However, a critical issue is the risk of storage collisions, where both contracts use the same storage slot, leading to data loss. This can be easily avoided by using namespaced storage layouts, which is a convention for naming struct holding storage variables [18]. Another critical issue is the risk of function selector clashes, where different functions have the same selector, overriding each other [27]. Since proxy patterns require cross-contract interaction, the Solidity compiler cannot detect these clashes: it is the developer's responsibility to avoid them. Compound, a key DeFi protocol, exemplifies this pattern through multiple upgrades, including Compound III [1]. Similarly, USDC, a widely used stable coin, leverages the proxy pattern for notable upgrades like the transition to USDC 2.0, showcasing its adaptability in the evolving digital currency landscape.

[1] `https://compound.finance/`

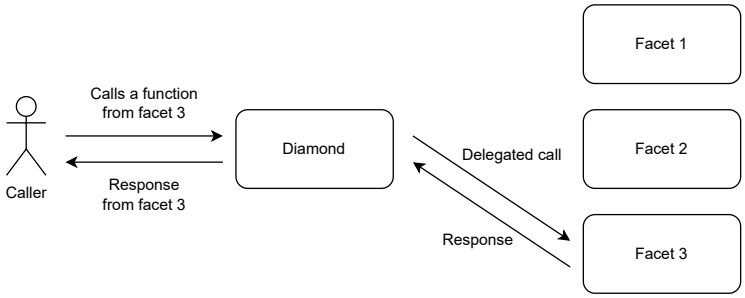


Fig. 2. Diamond pattern diagram.

The diamond pattern, depicted in Figure 2 is a more upgradeable version of the proxy pattern. This pattern solves the maximum contract size problem, which is a limitation of the proxy pattern. Indeed, logic can be separated into small contracts referred to as facets. A main contract, referred to as the implementation or diamond contract, points at the different facets to retrieve the applicable logic. The diamond contract can be upgraded by adding, replacing or removing facets. However, this pattern requires developers to manage the diamond storage manually because of the multiple implementation contracts. Also, it has the same critical issues as the proxy pattern, namely storage collisions and function selector clashes. The diamond pattern is being utilized by a diverse array of projects, as documented in Nick Mudge's Awesome Diamonds repository [25]. For example, Aavegotchi, a Non-Fungible Token (NFT) based gaming protocol, employs a single diamond pattern with eight distinct facets [2]. Each diamond and facet serves a specific purpose, contributing to the protocol's overall functionality.

A set of studies provide insight on upgradeable smart contract patterns. Kannengiesser et al. conduct a study on key smart contract development challenges across various distributed ledger technology (DLT) protocols, including Ethereum, Hyperledger, and EOSIO [19]. They highlight upgradability as a significant challenge and reference two upgradeable smart contract patterns, namely the diamond (referred to as the façade pattern) pattern and the proxy pattern. However, the paper lacks an in-depth analysis of these patterns. Two papers identify the proxy pattern but do not detail the forces, advantages, drawbacks, or gas costs considerations of this pattern [14, 22]. There is no mention of the diamond pattern. Two other works present a set of upgradeable smart contract patterns, including the proxy and diamond patterns [23, 28]. However, the papers do not provide an in-depth analysis of the functional and non-functional properties of these patterns, nor gas costs behaviors.

Additionally, it is to note that two studies focus on gas cost efficiency strategies in smart contracts development. Zarir et al.'s work focuses on transaction parameters rather than architectural design [35]. The study by Di et al. iden-

[2] `https://www.aavegotchi.com/`

tifies smart contract coding metrics impacting gas costs [10]. However, it does not specifically focus on upgradeable smart contracts. These contracts possess unique functionalities like proxy pointers and proxy contract management.

In summary, a research gap is identified regarding a thorough study of the proxy and diamond patterns. There is a lack of studies about gas costs analysis and formalization of the proxy and diamond patterns, especially using the Alexandrian form format.

4 Methodology

The methodology section of this paper details the approach employed for comparing the proxy and diamond patterns. This involves the definition of a baseline scenario and the development of a gas consumption evaluation test bench.

4.1 Protocol

Smart contract deployments are one of the most expensive operations in terms of gas costs. Therefore, assessing the deployment costs associated with various patterns is essential. Additionally, upgrades often necessitate deploying additional smart contracts, making it crucial to compare the deployment costs of each subsequent upgrade.

The protocol leverages a file notarization scenario, a standard use of smart contracts to ensure the integrity of critical data such as diplomas or scientific workflows [5, 9, 21]. The scenario is implemented using the proxy and diamond patterns, as well as a reference monolithic non-upgradeable smart contract referred to as the classic pattern.

For each pattern, the initial deployment of the notarization smart contract is referred to as version 1. The study then proceeds to two sequential upgrades of the application, a minor one (version 2) and a major one (version 3), to evaluate the gas costs. Features are added as updates are made. Figure 3 demonstrates the upgrades for each pattern.

Then the analysis turns to comparing the average gas cost of each function, for seeing which pattern is the less gas intensive when it comes to execution. Each function is accompanied by a test case ran hundreds of times, to simulate real-world usage of this file notarization application over an extended period and to check the robustness of the code.

4.2 Implementation

The code is written in Solidity, the main language used to develop decentralized application on EVM blockchains. For the experiment, the Universal Upgradeable Proxy Standard (UUPS) proxy pattern is used because it is the recommended standard at the time of writing by OpenZeppelin [6]. For this study, the diamond pattern implemented by SolidState is used. It utilizes Nick Mudge's gas-efficient Diamond 2 model [24, 30] in a plug-and-play fashion: the developer only needs to import the diamond made by SolidState without any configuration needed.

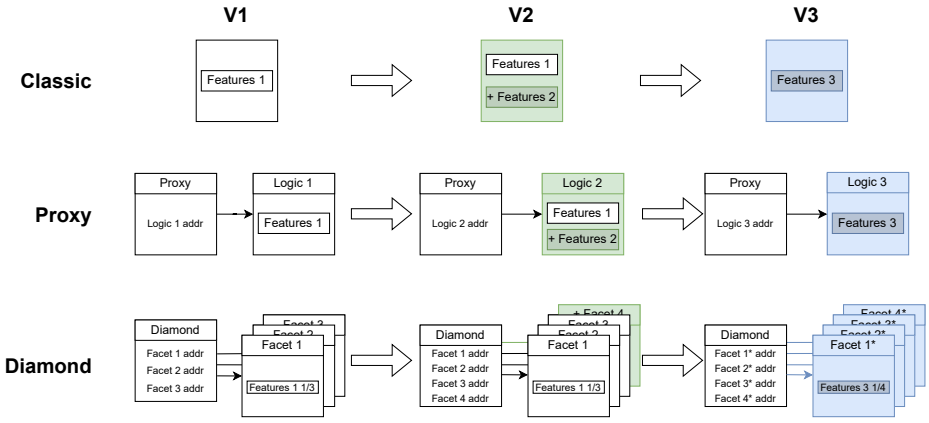


Fig. 3. Evolution of the notarization application across versions by patterns.

Version 1: Basic Notarization Application In the first version of the notarization use case, files are represented by a mapping between their name and their hash. The logic encompasses several functions. The function `addFile` is designed to notarize a file on the blockchain and modifies the state of the contract. The `getFileName` function is a view function that returns the name of the file. Similarly, `getFileHash` is a view function that provides the hash of the file. Lastly, `compareHash` is a pure function for comparing two hashes passed as parameters; while this operation is ideally performed off-chain, it is included here to demonstrate the gas cost of a pure function.

Version 2: Updatable File In case of file modifications, a function for updating it on the blockchain is needed. The second version of this notarization application contains the `updateFile` function, added on top of the previous version. This function updates the file hash of a notarized file and so modifies the state of the contract. This version can be seen as a minor update.

Version 3: Access Control For security reasons, access control is mandatory, so that only the owner of a file can modify or delete it. This involves creating a `File` structure containing the owner's address, the hash of the file's contents, the creation timestamp and the last modification timestamp. The previous mapping is replaced by one between the file's name and its `File` structure instance. Then, the `addFile` and `updateFile` functions are modified to work with this new `File` structure, and access control is added, where the new logic smart contract ensures that the caller interacts only with his own files. This version can be seen as a major update.

4.3 Unit Tests

As mentioned in section 4.1, each function has a unit test that is run hundreds of times (704 iterations for the addFile function across all unit tests of version 3, for

example). It calls the function with random parameters (file name, hash, etc.) and checks that the result is correct. This is done to simulate real-world usage of this file notarization application. An expected result is computed before the function call, and the actual result is compared to it. In the `addFile` unit test, we expect the smart contract to add the file name and hash to its storage, using either the proxy or diamond pattern. The `updateFile` unit test should update the file hash in the contract's storage, while the `deleteFile` test ensures the file's removal. In the `compareHash` test, the outcome should be true for matching hashes and false otherwise. The `getFileName`, `getFileHash`, `getFileOwner`, `getFileCreatedAt`, and `getFileLastModifiedAt` unit tests respectively verify the return of the file's name, hash, owner address, creation timestamp, and last modification timestamp. Finally, the `getFileDetails` test checks for the return of all the previously mentioned file properties.

Testing each single function is important to ensure that the code is robust and that it does not break when upgrading the smart contract. These tests are also conducted to compare the gas costs across various patterns, specifically examining each type of function, including pure, view, and state-modifying functions. By testing pure functions, the gas costs of computations without storage access can be assessed and compared across patterns. View functions are tested to evaluate the gas costs of computations with storage access. Finally, state-modifying functions are tested to assess the gas costs of storage writes.

Foundry plays a critical role in this testing process by autonomously simulating an Ethereum Virtual Machine (EVM) blockchain environment. During test execution, it deploys the contracts and carries out the testing scenarios within this emulated setting, utilizing the default configuration of this local blockchain.

The entire source code, tests and results here are available in the accompanying source code repository [3].

4.4 Results Retrieval

Through these unit tests, Foundry produces gas reports that provide insights into the gas consumption for each function, alongside the gas costs associated with deployment and the size of the contracts. The cost of function calls is quantified in gas units, and the contract size is measured in bytes.

These gas cost results for each function are derived by summing the gas costs associated with every operation performed within the function. Each operation, also known as opcode, has a fixed gas cost, which is the same for all patterns. Those instructions are described in the Ethereum Yellow Paper [32]. For example, the `SLOAD` opcode, which reads a value from storage, has a fixed cost of 100 for warm access, and 2100 gas for cold access. Then, the deployment cost is the sum of several components. First, the `TRANSACTION` opcode incurs 21,000 gas units, representing the base cost of every transaction on the EVM. Additionally, there is the `CREATE` opcode, costing 32,000 gas units, which is used for creating a new contract. Next, the cost related to the bytecode includes 4 times the

[3] `https://anonymous.4open.science/r/proxy-diamond-patterns-gas-analysis`

number of 0 bytes and 16 times the number of non-zero bytes. Furthermore, 200 gas units are added for every byte of the contract's size. Finally, if a constructor function is present, its cost is also included in the deployment cost.

In addition, the framework furnishes complete stack traces of the calls, detailing the gas consumption for each operation. These traces are utilized to construct charts that capture the gas cost of each function call. This level of detail supplements gas reports, which typically provide only a summary of costs, including the minimum, average, median, and maximum values.

5 Evaluation

This section presents the gas costs evaluation of the proxy and diamond smart contract patterns, against a baseline built using the classic pattern in the context of an app deployment and upgrade [4].

5.1 Gas Cost During Smart Contract Deployment and Upgrades

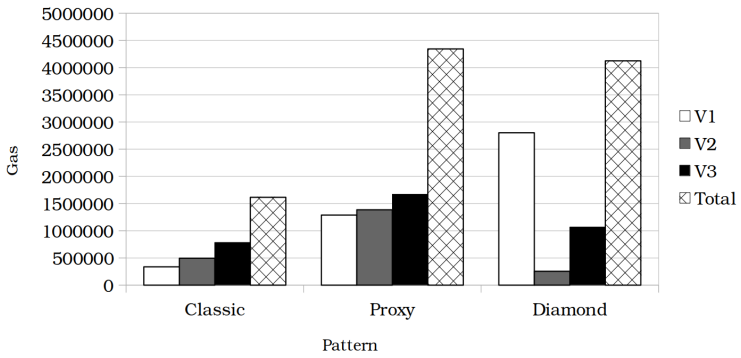


Fig. 4. Deployment cost by pattern and version.

The primary aim is to compare the gas costs during the deployment (version 1) and upgrades (version 2 and 3) of the file notarization application implemented with the three different patterns. For the classic and proxy pattern, a new contract is deployed for each version. For the proxy, the proxy pointer is updated in the smart contract implementation. For the diamond pattern, in version 2, only facet is changed, and in version 3, all facets are changed.

Figure 4 presents the deployment costs, in gas units, of each version by pattern. The cost of deployment increases with each version, except for the diamond

[4] All results can be found here: `https://anonymous.4open.science/r/proxy-diamond-patterns-gas-analysis/data`

pattern, where the initial cost of deployment is significant. In all, the classic pattern requires just 1,614,545 units of gas, while the proxy and diamond patterns consume around 2.6 times as much, at 4,343,104 and 4,123,977 units of gas respectively. The classic pattern appears the most gas efficient. The increased consumption at deployment of the proxy and diamond patterns relates to the need to deploy more contracts compared to the classic pattern: two for the proxy pattern, and four for the diamond one.

5.2 Gas Cost During Smart Contract Execution

The analysis now turns to comparing the average gas cost of each function, for seeing which pattern is the most gas efficient when it comes to execution.

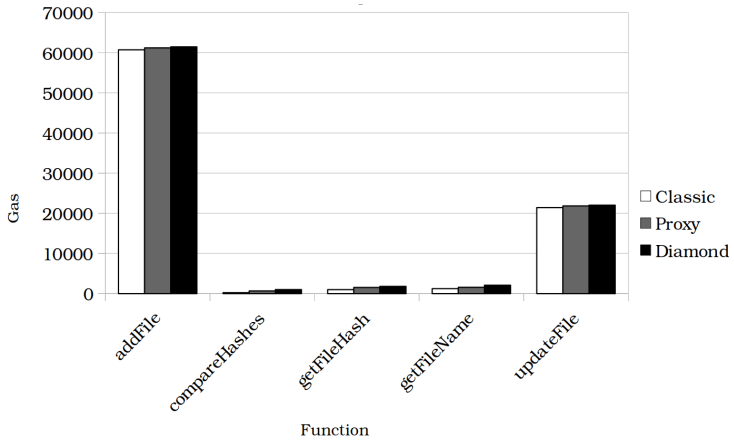


Fig. 5. Average function cost by pattern for version 2.

Figure 5 depicts the gas costs observed for version 2 averaged per function. It is noted that the same gas costs behavior is observed in version 1 and 3 (c.f., online gas reports). The `addFile` function is the costliest, followed by the `updateFile` function. The other functions require less gas. There is no significant difference in gas cost among the three patterns. The proxy pattern is slightly more expensive than the classic pattern, and the diamond pattern is slightly more expensive than the proxy pattern. The most important difference is for `compareHashes` being 300 times cheaper than `addFile` for the classic pattern.

The difference in gas costs between `compareHashes` and `addFile` is due to the type of operations done in the functions. The reason is that the `addFile` does two writings in the storage, whereas the `updateFile` does only one. The other functions only do reads in the storage, which cost less gas than writes.

The classic pattern does not delegate any call to another contract, whereas the proxy and diamond patterns do, so it adds a cost. Then, when calling a diamond, it needs to do more operations than a proxy to find the right facet to call.

This issue is not present with the proxy pattern, as there is only one implementation contract. Then, the getters (functions like `getFileName`, `getFileHash`, etc.) are `view` functions, which means that they only do a state lookup. This lookup costs 2100 gas for a cold storage load (when reading a variable for the first time) [20]. Here, the cold storage load (or cold access) means that it has not been read yet in the current transaction, whereas warm access means that the value has already been read. `compareHashes` is a `pure` function, which means that it does not do any state lookup or modification. It only does some computations on given parameters, blockchain's global variables or not. This is why it requires less gas than the getters in the classic pattern.

In reviewing the data obtained from the comparative analyses, the deployment phase exhibits the most significant variation among the different patterns, primarily due to the fluctuating number of smart contracts deployed. For this metric, the diamond pattern is the cheapest, especially for minor upgrades. During execution, the gas costs are similar across the patterns, though there is a marginal escalation in costs when transitioning from the classic architecture to the proxy pattern, and subsequently from the proxy to the diamond pattern. This increase in gas costs is due to the additional operations required to delegate calls to the logic contract(s) in the proxy and diamond patterns.

6 Discussion

This section summarizes and consolidates the proxy and diamond patterns by following the Alexandrian form format proposed by Christopher Alexander [2]. A decision model is proposed to help developers choose between the proxy and diamond patterns.

6.1 Proxy Pattern Outline

- **Summary:** The proxy pattern facilitates upgradability in smart contracts through a proxy contract pointing to the latest version of a logic contract.
- **Context:** A smart contract must be upgraded due to evolving requirements and potential improvements [7].
- **Problem:** Traditional smart contracts lack the ability to be updated without manual storage migration, posing challenges in addressing vulnerabilities, enhancing functionality, and adapting to changing circumstances.
- **Forces (tradeoffs):** The problem requires balancing the following forces:
  - *Immutability vs. Upgradability.* Smart contracts on blockchain platforms are traditionally immutable once deployed, ensuring security but hindering adaptability;
  - *Gas Costs vs. Flexibility.* The balance between minimizing gas costs and providing flexibility for contract upgrades is a critical challenge;
  - *Trust vs. Transparency.* Establishing trust in the upgrade process while maintaining transparency is a delicate balance.

- **Solution:** The smart contract proxy pattern introduces a proxy contract as an intermediary layer. This proxy delegates calls to a logic contract, allowing seamless upgrades by deploying new logic contracts and updating the proxy to point to the latest version, hence allowing for dynamic updates.
- **Consequences:**
  - Benefits:
    - *Upgradability.* Allows upgrades without the need to change the contract address for a seamless user experience and without requiring data migration;
    - *Simplest Upgradeable Pattern.* The proxy pattern is the simplest upgradeable pattern, requiring only two contracts (proxy and logic), and the upgrade process is straightforward. It requires only the deployment of a new logic contract and a proxy update to point to the new logic contract through a proxy administration function.
  - Drawbacks:
    - *Compatibility Maintenance.* Ensuring compatibility between different contract versions in the proxy pattern requires meticulous consideration. Preserving consistent function selectors and storage layouts becomes crucial for a smooth transition during upgrades;
    - *Limited Direct Function Visibility.* Indirect access to the logic functions visibility. Users only have direct access to the proxy administration functions, necessitating consultation of accompanying documentation to identify callable functions. This may potentially introduce usability challenges for end-users;
    - *Storage Collision.* The proxy pattern requires careful consideration of storage layout to avoid storage overlap between the proxy and the logic contract;
    - *Function Selector Clash.* Because of the cross-contract interaction, the Solidity compiler cannot detect function selector clashes between the proxy and logic contracts. Therefore, the proxy pattern requires careful consideration of function selector naming to avoid clashes, so that no function is overwritten.
- **Related patterns:** Diamond pattern.
- **Known uses:**
  - *EIP-897.* An Ethereum Improvement Proposal outlining a standardized implementation of proxy contracts [23];
  - *OpenZeppelin's Proxy Patterns.* Practical implementations and documentation in popular smart contract development frameworks [3];
  - *Compound and USDC.* Respectively, a DeFi protocol and a stable coin leveraging the proxy pattern for upgrades.

6.2 Diamond Pattern Outline

- **Summary:** The diamond pattern addresses the challenge of contract size limitations and enhances maintainability and versioning by employing multiple implementation contracts;

- **Context:** Need for a solution to improve maintainability in large contracts.
- **Problem:** Traditional approaches face challenges in managing contract size and versioning effectively, limiting the scalability and maintenance of smart contracts. The need for more scalable and maintainable smart contract structures is evident.
- **Forces (tradeoffs):** The problem requires balancing the following forces:
  - *Contract Size vs. Modularity* Balancing the need for compact contract sizes with the demand for modular, well-organized code structures poses a challenge;
  - *Maintainability vs. Simplicity* Achieving improved maintainability without introducing unnecessary complexity is an ongoing struggle;
  - *Scalability vs. Consistency* Scaling smart contract applications can hinder consistency as it requires accommodating versioning and updates.
- **Solution:** The diamond pattern introduces a structure where a proxy can point to multiple logic contracts. Its deployment involves the deployment of all contracts (diamond and facets), retrieving facets function selectors, and leveraging both to implement a diamond cut.
- **Consequences:**
  - Benefits:
    - *Better upgradability.* Same as the proxy pattern (no address change, no data migration), but thanks to the possibility of code division into multiple facets, it enables the deployment of smaller contracts during upgrades or updates to already deployed facets;
    - *Modularity.* Generic code can be reused across multiple contracts, facilitating code organization and maintenance. A facet can be used in multiple diamonds;
    - *Contract size.* Thanks to a modular structure, it can theoretically support an infinite number of facets. Therefore, the whole smart contract system has no size limit;
    - *Cheaper minor upgrades.* Most of the time, only one facet is updated, so only small contracts are deployed for low gas costs;
    - *Shorter compilation time.* Only modified facets need to be compiled, so for the same logic code, the compilation time is shorter than for the classic pattern and proxy pattern.
  - Drawbacks:
    - *Implementation Complexity.* The diamond pattern introduces a more complex structure than the classic and proxy patterns, which can be challenging to grasp for developers new to the pattern. This increased complexity demands specific knowledge and careful consideration during the implementation phase as there is currently a lack of supporting implementation libraries;
    - *Complexity in managing multiple logic contracts.* The diamond pattern's enhanced flexibility comes at the cost of increased complexity, necessitating meticulous planning and testing to ensure smooth data interaction and integrity across different contract versions. The intricacies of managing multiple logic contracts require careful consideration to avoid unintended consequences during upgrades;

* *Limited Direct Function Visibility.* Like for the proxy pattern, there is indirect access to the logic functions visibility. Users only have direct access to the diamond administration functions, necessitating consultation of accompanying documentation to identify callable functions.

– *Related patterns:* Proxy pattern.
– *Known uses:*
  • *Aavegotchi.* A Non-Fungible Token (NFT) based gaming protocol;
  • *GeoWeb.* A dApp managing digital land property rights using NFTs.

6.3 Choosing between the Proxy and Diamond Patterns

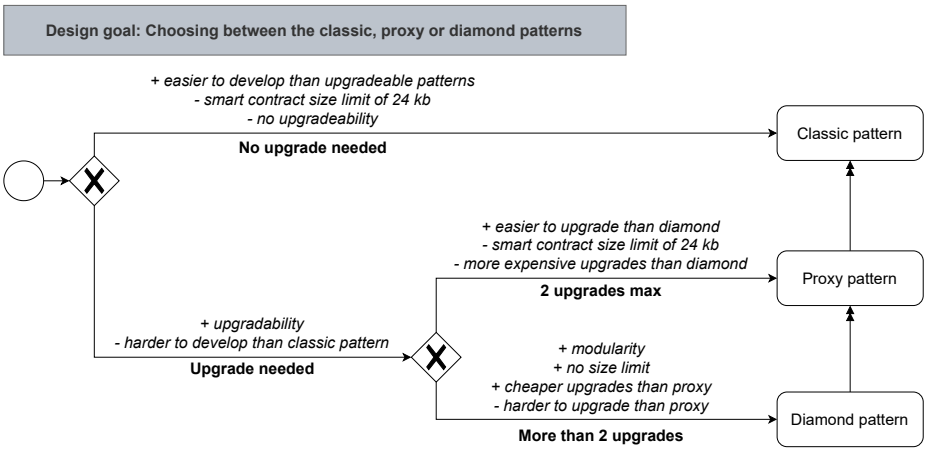


Fig. 6. Decision model for upgradeable smart contract pattern usage.

Figure 6 proposes a decision model to orient between a proxy or a diamond pattern in the design stage of an upgradeable smart contract. It follows the design goal decision model introduced by Xu where decisions are modeled using a logical BPMN flow [33]. More precisely, arrows, logical gateways, and functional and non-functional properties orient the decision path.

The choice of smart contract pattern largely depends on the need for upgradability. For scenarios without upgradability needs, the classic pattern is preferable due to its straightforward development process, relying on contract inheritance and library imports. However, upgrades in the classic pattern often involve complex and resource-intensive data migration. This pattern also poses challenges in communicating new contract addresses to users, potentially affecting the user experience. For extensive upgradeable features, the diamond pattern is recommended. Its modular nature allows for easy addition or removal of facets, reducing compilation time. However, it is less cost-effective initially compared to

the proxy pattern and requires in-depth knowledge of smart contract storage and facet-library management. The proxy pattern is advised for limited code sizes or infrequent upgrades. It simplifies development and integrates easily with libraries like OpenZeppelin's. This pattern enhances upgradability by separating logic and state, reducing the need for data migration. But it offers less flexibility and modularity compared to the diamond pattern and demands careful consideration to maintain compatibility across versions.

In the end, while the classic pattern excels in execution ease, the proxy and diamond patterns provide a more manageable framework for upgrades, simplifying contract interactions for users during updates. The diamond pattern is more suitable for extensive upgrades, while the proxy pattern is recommended for limited upgrades and simpler development.

7 Conclusion

In conclusion, this comprehensive study delves into the intricacies of upgradeable smart contract design patterns—a critical facet of contemporary blockchain applications. The comparative analysis specifically focuses on the gas costs associated with deploying, using, and upgrading decentralized applications using two prominent upgradeable patterns: the proxy and diamond patterns.

Each pattern unfolds with distinct strengths and weaknesses, delineating its applicability across diverse scenarios. The classic pattern implies an unpractical and costly approach to smart contract upgrades because of the need of manual data migration. The proxy pattern offers the simplest solution for upgradability, but security concerns such as storage collisions and function selector clashes remain. This demands a developer's careful attention and thus results in a more challenging pattern to utilize. The diamond pattern is the most complex of the three, but it offers the most flexibility and maintainability thanks to its modularity. Moreover, the diamond pattern is the most gas efficient when it comes to doing more than two upgrades. This is because the diamond pattern does not need to deploy long contracts, but only small facets. Finally, this pattern is the most scalable because it does not have a contract size limit.

Despite these considerations, real-world implementations in projects like Compound, USDC, GeoWeb, and Aavegotchi underscore the substantial impact of the proxy and diamond patterns in the blockchain domain. The proxy and diamond patterns stand as facilitators of flexible, upgradeable, and scalable smart contracts, showcasing their adaptability across a spectrum of blockchain applications.

To generalize these initial findings, the study advocates for extending experiments beyond the notarization scenario used as a comparison baseline for this paper. Essential to this endeavor is the necessity for additional experiments encompassing diverse blockchain networks and pattern libraries to extrapolate and validate the findings. For future work, a replication of the study on alternative upgradeable patterns would also provide a more comprehensive understanding of upgradeable smart contract patterns.

References

1. Alexander, C.: A pattern language: towns, buildings, construction. Oxford university press (1977)
2. Alexander, C.: The timeless way of building, vol. 1. New york: Oxford university press (1979)
3. Amri, S.A., Aniello, L., Sassone, V.: A review of upgradeable smart contract patterns based on openzeppelin technique. The Journal of The British Blockchain Association (2023)
4. Ayub, M., Saleem, T., Janjua, M., Ahmad, T.: Storage state analysis and extraction of ethereum blockchain smart contracts. ACM Transactions on Software Engineering and Methodology **32**(3), 1–32 (2023)
5. Badr, A., Rafferty, L., Mahmoud, Q.H., Elgazzar, K., Hung, P.C.: A permissioned blockchain-based system for verification of academic records. In: 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–5. IEEE (2019)
6. Barros, G., Gallagher, P.: Erc-1822: Universal upgradeable proxy standard (uups). `https://eips.ethereum.org/EIPS/eip-1822` (2019-03-04), accessed: March 4, 2019
7. Buterin, V., et al.: Ethereum: a next generation smart contract and decentralized application platform (2013). URL {http://ethereum. org/ethereum. html} (2017)
8. Chen, J., Xia, X., Lo, D., Grundy, J., Yang, X.: Maintaining smart contracts on ethereum: Issues, techniques, and future challenges. arXiv preprint arXiv:2007.00286 (2020)
9. Coelho, R., Braga, R., David, J.M.N., Stroele, V., Campos, F., Dantas, M.: A blockchain-based architecture for trust in collaborative scientific experimentation. Journal of Grid Computing **20**(4), 1–31 (2022)
10. Di Sorbo, A., Laudanna, S., Vacca, A., Visaggio, C.A., Canfora, G.: Profiling gas consumption in solidity smart contracts. Journal of Systems and Software **186**, 111193 (2022)
11. Docs, S.: Bytes and string. `https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html#bytes-and-string` (2023-04-14), accessed: April 14, 2023
12. Docs, S.: Layout of state variables in storage. `https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html` (2023-04-14), accessed: April 14, 2023
13. Docs, S.: Mappings and dynamic arrays. `https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html#mappings-and-dynamic-arrays` (2023-04-14), accessed: April 14, 2023
14. Ebrahimi, A.M., Adams, B., Oliva, G.A., Hassan, A.E.: A large-scale exploratory study on the proxy pattern in ethereum
15. Ethereum: Gas and fees. `https://ethereum.org/en/developers/docs/gas/` (2023-07-19), accessed: July 19, 2023
16. Ethereum: Ethereum virtual machine (evm). `https://ethereum.org/en/developers/docs/evm/` (2023-09-02), accessed: September 2, 2023
17. Foundry: Foundry book. `https://book.getfoundry.sh/` (2023-11-22), accessed: November 22, 2023
18. Francisco Giordano, Hadrien Croubois, E.G., Lau, E.: Erc-7201: Namespaced storage layout. `https://eips.ethereum.org/EIPS/eip-7201`

19. Kannengießer, N., Lins, S., Sander, C., Winter, K., Frey, H., Sunyaev, A.: Challenges and common solutions in smart contract development. IEEE Transactions on Software Engineering **48**(11), 4291–4318 (2021)
20. Kostamis, P., Sendros, A., Efraimidis, P.: Exploring ethereum's data stores: A cost and performance comparison. In: 2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). pp. 53–60. IEEE (2021)
21. Leible, S., Schlager, S., Schubotz, M., Gipp, B.: A review on blockchain technology and blockchain projects fostering open science. Frontiers in Blockchain p. 16 (2019)
22. Marchesi, L., Marchesi, M., Destefanis, G., Barabino, G., Tigano, D.: Design patterns for gas optimization in ethereum. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). pp. 9–15. IEEE (2020)
23. Meisami, S., Bodell III, W.E.: A comprehensive survey of upgradeable smart contract patterns. arXiv preprint arXiv:2304.03405 (2023)
24. Mudge, N.: Diamond 2 hardhat implementation. `https://github.com/mudgen/diamond-2-hardhat` (2022-12-16), accessed: December 16, 2022
25. Mudge, N.: Awesome diamonds. `https://github.com/mudgen/awesome-diamonds` (2023-11-01), accessed: November 1, 2023
26. OpenZeppelin: Proxy patterns. `https://blog.openzeppelin.com/proxy-patterns` (2018-04-19), accessed: April 19, 2018
27. Palladino, P.: Malicious backdoors in ethereum proxies. `https://medium.com/nomic-foundation-blog/malicious-backdoors-in-ethereum-proxies-62629adf3357` (2018-06-01)
28. Qasse, I., Hamdaqa, M., Jónsson, B.T.: Smart contract upgradeability on the ethereum blockchain platform: An exploratory study. arXiv preprint arXiv:2304.06568 (2023)
29. Six, N., Herbaut, N., Salinesi, C.: Blockchain software patterns for the design of decentralized applications: A systematic literature review. Blockchain: Research and Applications **3**(2), 100061 (2022)
30. SolidState: Solidstate diamond. `https://github.com/solidstate-network/solidstate-solidity/tree/master/contracts/proxy/diamond` (2023-10-12), accessed: October 12, 2023
31. Szabo, N.: Formalizing and securing relationships on public networks. First monday (1997)
32. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)
33. Xu, X., Bandara, H.D., Lu, Q., Weber, I., Bass, L., Zhu, L.: A decision model for choosing patterns in blockchain-based applications. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA). pp. 47–57. IEEE (2021)
34. Xu, X., Pautasso, C., Zhu, L., Lu, Q., Weber, I.: A pattern collection for blockchain-based applications. In: Proceedings of the 23rd European Conference on Pattern Languages of Programs. pp. 1–20 (2018)
35. Zarir, A.A., Oliva, G.A., Jiang, Z.M., Hassan, A.E.: Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(3), 1–38 (2021)

Dear reviewers,

Please find hereinafter our answers to your helpful feedback.

I - Review 1

- **C1. Some sentences are lengthy and could benefit from being broken down for better readability. Minor grammatical corrections could further refine the language quality.**

  We thank the reviewer for its feedback. Those errors were inadvertent errors. The whole paper has been revised and improved in terms of grammar, length of sentences & readability.

- **C2. The exact calculation of the gas costs should be described in detail for the reader.**

  We add more details about how the gas cost is computed for both deployment & execution in section 4.4:

  "These gas cost results for each function are derived by summing the gas costs associated with every operation performed within the function. Each operation, also known as opcode, has a fixed gas cost, which is the same for all patterns. Those instructions are described in the Ethereum Yellow Paper [32]. For example, the SLOAD opcode, which reads a value from storage, has a fixed cost of 100 for warm access, and 2100 gas for cold access. Then, the deployment cost is the sum of several components. First, the TRANSACTION opcode incurs 21,000 gas units, representing the base cost of every transaction on the EVM. Additionally, there is the CREATE opcode, costing 32,000 gas units, which is used for creating a new contract. Next, the cost related to the bytecode includes 4 times the n number of 0 bytes and 16 times the number of non-zero bytes. Furthermore, 200 gas units are added for every byte of the contract's size. Finally, if a constructor function is present, its cost is also included in the deployment cost. "

- **C3. A more thorough explanation of the choice of specific test cases could enhance this section.**

  We added a paragraph to justify the choice of each unit test in the section 4.3 as follows:

  "As mentioned in section 4.1, each function has a unit test that is run hundreds of times (704 iterations for the addFile function across all unit tests of version 3, for example). It calls the function with random parameters (file name, hash, etc.) and checks that the result is correct. This is done to simulate real-world usage of this file notarization application. An expected result is computed before the function call, and the actual result is compared to it. In the addFile unit test, we expect the smart contract to add the file name and hash to its storage, using either the proxy or diamond pattern. The updateFile unit test should update the file hash in the contract's storage, while the deleteFile test ensures the file's removal. In the compareHash test, the outcome should be true for matching hashes and false otherwise. The getFileName, getFileHash, getFileOwner, getFileCreatedAt, and getFileLastModifiedAt unit tests respectively verify the return of the file's name, hash,

owner address, creation timestamp, and last modification timestamp. Finally, the getFileDetails test checks for the return of all the previously mentioned file properties."

- **C4. Lack of In-Depth Analysis: The evaluation section primarily presents quantitative results without a sufficient qualitative analysis.**

  Section 6 (Discussion) aims at including a qualitative analysis of the patterns to complement the evaluation section. To deepen the qualitative analysis, we have increased in the revised version of the paper the consequences item (benefits and drawbacks) of each pattern:

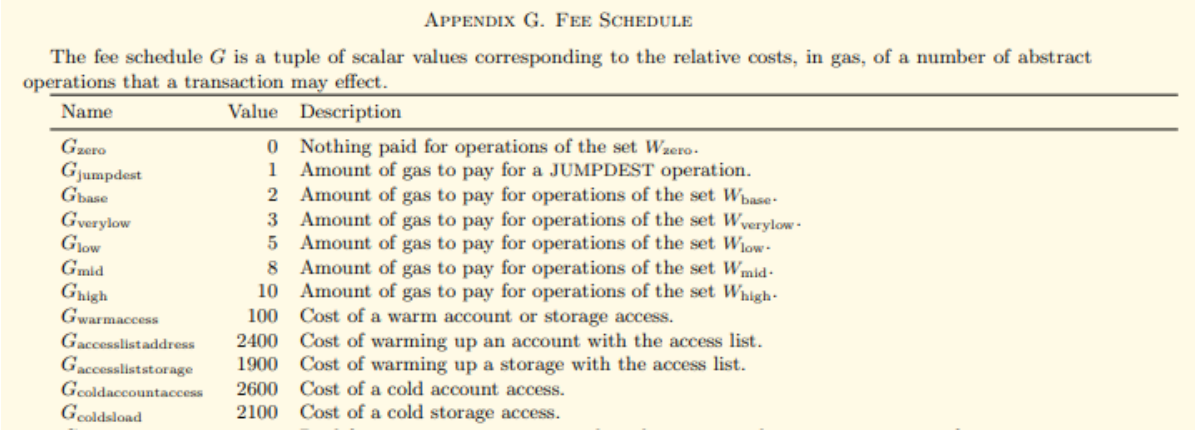
  In section 6.1 (proxy pattern):
  - *Benefits:*
    - *Upgradability. Allows upgrades without the need to change the contract address for a seamless user experience and without requiring data migration;*
    - *Simplest Upgradeable Pattern. The proxy pattern is the simplest upgradeable pattern, requiring only two contracts (proxy and logic), and the upgrade process is straightforward. It requires only the deployment of a new logic contract and a proxy update to point to the new logic contract through a proxy administration function.*
  - *Drawbacks:*
    - *[...]*
    - *Storage Collision. The proxy pattern requires careful consideration of storage layout to avoid storage overlap between the proxy and the logic contract;*
    - *Function Selector Clash. Because of the cross-contract interaction, the Solidity compiler cannot detect function selector clashes between the proxy and logic contracts. Therefore, the proxy pattern requires careful consideration of function selector naming to avoid clashes, so that no function is overwritten."*

  In section 6.2 (diamond pattern):
  - *"Benefits:*
    - *Better upgradability. Same as the proxy pattern (no address change, no data migration), but thanks to the possibility of code division into multiple facets, it enables the deployment of smaller contracts during upgrades or updates to already deployed facets. Consequently, for identical logic upgrades, the diamond pattern can become more gas-efficient than the proxy pattern after a few upgrades and might even surpass the classic pattern in terms of gas efficiency over time;*
    - *Modularity. Generic code can be reused across multiple contracts, facilitating code organization and maintenance. A facet can be used in multiple diamonds;*
    - *Contract size. Thanks to a modular structure, it can theoretically support an infinite number of facets. Therefore, the whole smart contract system has no size limit;*
    - *Cheaper minor upgrades. Most of the time, only one facet is updated, so only small contracts are deployed for low gas costs;*
    - *[...]*

- *Drawbacks:*
  - *[...]*
  - *Limited Direct Function Visibility. Like for the proxy pattern, there is indirect access to the logic functions visibility. Users only have direct access to the diamond administration functions, necessitating consultation of accompanying documentation to identify callable functions."*

- **C5. Lack of deeper exploration into why certain patterns consume more gas and the implications of these findings for smart contract developers (here a detailed description of how the gas costs are calculated would help).**

  As mentioned earlier in C2 this is addressed in section 4.4 of the paper, where we talk about opcodes. We describe in detail how deployment gas cost is calculated, and we refer to the Ethereum Yellow Paper which has the list of all EVM instructions with their cost in gas units:



APPENDIX G. FEE SCHEDULE

The fee schedule G is a tuple of scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{jumpdest}	1	Amount of gas to pay for a JUMPDEST operation.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
G_{verylow}	3	Amount of gas to pay for operations of the set W_{verylow}.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{\text{warmaccess}}$	100	Cost of a warm account or storage access.
$G_{\text{accesslistaddress}}$	2400	Cost of warming up an account with the access list.
$G_{\text{accessliststorage}}$	1900	Cost of warming up a storage with the access list.
$G_{\text{coldaccountaccess}}$	2600	Cost of a cold account access.
$G_{\text{coldsload}}$	2100	Cost of a cold storage access.

- **C6. Limited Contextual Comparison: The comparison of gas costs is restricted to the Proxy and Diamond patterns, without considering a wider range of smart contract deployment patterns. This narrow focus may limit the paper's applicability to a broader context of smart contract engineering.**

  We acknowledge this issue, and we mention it as a limitation of the study in the conclusion. We will proceed to thorough comparison in future works (also indicated in the conclusion):

  "To generalize these initial findings, the study advocates for extending experiments beyond the notarization scenario used as a comparison baseline for this paper. Essential to this endeavor is the necessity for additional experiments encompassing diverse blockchain networks and pattern libraries to extrapolate and validate the findings. For future work, a replication of the study on alternative upgradeable patterns would also provide a more comprehensive understanding of upgradeable smart contract patterns."

- **C7. Methodological Transparency: The criteria for selecting test cases in the evaluation are not adequately explained. A more transparent explanation of why these particular cases were chosen and how they represent common or critical scenarios in smart contract deployment would enhance the credibility of the evaluation.**

  We refer the reader to the answer to I.C3

- **C8. Benchmarking Issues: The paper does not benchmark its findings against existing studies or industry standards. Inclusion of benchmarking would have provided a clearer understanding of the performance of Proxy and Diamond patterns in a broader landscape.**

  We deeply understand and agree with this concern. To the best of our knowledge, there is no existing study regarding the gas cost comparison between the diamond pattern and the proxy pattern that we could use as a baseline for this study.

- **C9. Practical Implications Underexplored: The practical implications of the findings are underexplored. The paper could significantly benefit from a discussion on how the gas cost differences might affect the choice of patterns in real-world applications and the trade-offs involved.**

  We thank the reviewer for this feedback and agree that more research can be done about real-word usage of these patterns, and consequences on those applications. It is mentioned as a limitation of this study planned for future work in the conclusion of the paper, cited in C6.

  Moreover, we reviewed the decision model to integrate quantitative metrics based on the experiment, thus helping the choice of patterns. We plan

- **C10. Scalability and Performance Considerations: There is a lack of discussion on scalability and performance considerations beyond gas costs. Factors such as the ease of upgradability, security implications, and long-term maintainability are crucial for smart contract engineering but are not addressed in depth.**

  To answer this concern, we discuss these elements more thoroughly in the discussion section, under the consequences of each pattern (benefits and drawbacks) in Section 6.

  For the proxy pattern, the following elements are identified:
  Pros:
  - Simplest upgradeable pattern to develop;
  - No manual data migration;
  - Simpler to upgrade than the diamond pattern;
  - No data migration needed.

Cons:
  - More expensive upgrades than the diamond pattern;
  - Same 24 kb smart contract size limit as the classic pattern.

  For the diamond pattern, the following elements are identified:
  Pros:
  - Gas-efficient upgrades;
  - No smart contract size limit;
  - Modularity thus better maintainability;
  - No manual data migration needed.

Cons:

- Harder to develop than the classic or proxy pattern (developers must be careful of storage collision & function selectors clashes);
- Complex to upgrade (meta-programming or external libraries needed).

A paragraph additionally summarizes these elements in the conclusion:

"Each pattern unfolds with distinct strengths and weaknesses, delineating its applicability across diverse scenarios. The classic pattern implies an unpractical and costly approach to smart contract upgrades because of the need of manual data migration. The proxy pattern offers the simplest solution for upgradability, but security concerns such as storage collisions and function selector clashes remain. This demands a developer's careful attention and thus results in a more challenging pattern to utilize. The diamond pattern is the most complex of the three, but it offers the most flexibility and maintainability thanks to its modularity. Moreover, the diamond pattern is the most gas efficient when it comes to doing more than two upgrades. This is because the diamond pattern does not need to deploy long contracts, but only small facets. Finally, this pattern is the most scalable because it does not have a contract size limit."

II - Review 2

- **C1: The entire evaluation of the paper lies on a simple and custom implementation of a file retrieval system. The simple nature of the contract suggests that conclusions based on it may not be apply beyond this example.**

  We thank the reviewer for this feedback. The scenario was chosen after reviewing the related work, as no baseline scenario has been identified that we could build or compare on. Hence this current implementation aims at providing a testbed to identify the key differences between the proxy and diamond, that other benchmark papers could reuse to complexify the analysis.

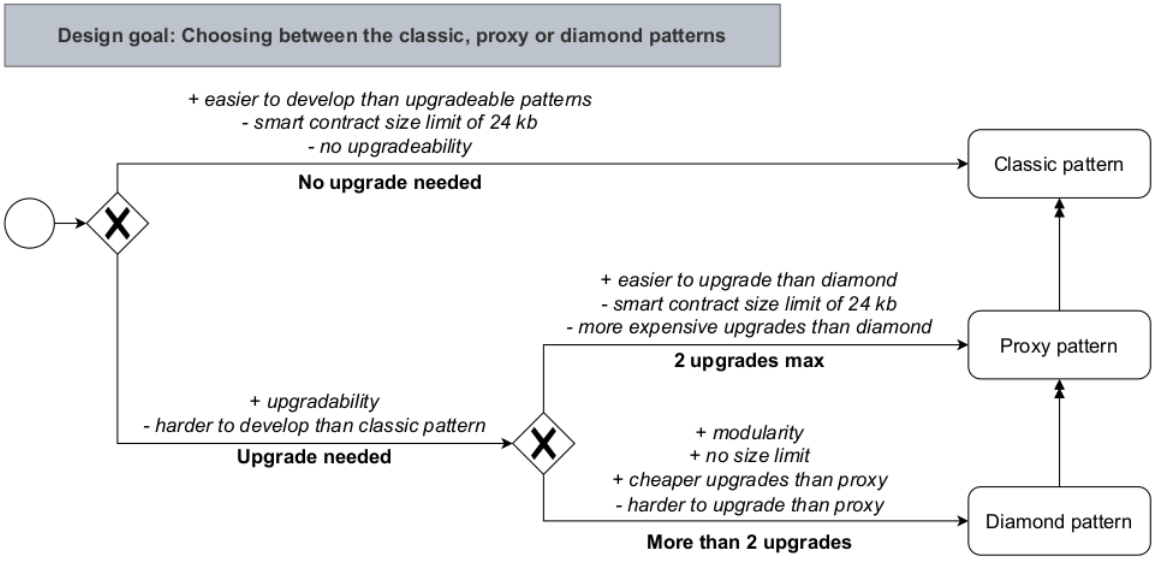
  We add in the conclusion section of the revised version of the paper the need to expand the testbed to more complex scenario, which is planned in our future work:

  "To generalize these initial findings, the study advocates for extending experiments beyond the notarization scenario used as a comparison baseline for this paper." (Conclusion section)

- **C2: The recommendations provided by the paper are unclear and undefined. For example Figure 6 uses the criteria "complex or numerous features", yet the complexity threshold is never defined and is subject to interpretation.**

  Figure 6 has been updated to define this threshold in the decision model. The threshold is identified based on the results obtained when comparing the deployment and upgrades of each pattern (c.f., Figure 4, where the second diamond pattern upgrade is starting to cost less than proxy ones).

We also made the pros and cons of each pattern clearer. Here is the edited decision model:



- **C3: The paper fails to mention some of the critical issues of the diamond standard, such as the risk of storage collision.**

  We thank the reviewer for this feedback. We added some references to critical issues of the diamond standard in the related work section when presenting each pattern:

  "However, a critical issue is the risk of storage collisions, where both contracts use the same storage slot, leading to data loss. This can be easily avoided by using namespaced storage layouts, which is a convention for naming struct holding storage variables [18]. Another critical issue is the risk of function selector clashes, where different functions have the same selector, overriding each other [27]. Since proxy patterns require cross-contract interaction, the Solidity compiler cannot detect these clashes: it is the developer's responsibility to avoid them. [...] Also, it has the same critical issues as the proxy pattern, namely storage collisions and function selector clashes."

- **C4: Novelty and significance: The paper does not provide any insight that was not already public knowledge (ex: adding a lookup table makes the execution more costly).**

  There is a research gap in gas cost comparison between the diamond pattern and the proxy one. Moreover, there is no software engineering pattern description for those smart contract patterns in the current literature. This is a limiting factor for sharing best practices in the smart contract domain.

- **C5: Note that better-defined standards already exist (e.g., https://eips.ethereum.org/EIPS/eip-7201) that solve some of the diamond pattern's shortcomings.**

  We thank the reviewer for this feedback. However, we believe there is a misunderstanding. Indeed, the EIP-7201 is taken from the diamond pattern EIP (EIP-2535). As it is mentioned it the EIP-7201:

“This pattern has sometimes been referred to as “diamond storage”. This causes it to be conflated with the “diamond proxy pattern”, even though they can be used independently of each other. This EIP has chosen to use a different name to clearly differentiate it from the proxy pattern.”

And in the diamond pattern EIP, we can find the exact same pattern, written three years earlier. At the time of the EIP-2535, this storage standard has been coded, but not officially proposed as an EIP itself:

“A state variable or storage layout organizational pattern is needed because Solidity’s builtin storage layout system doesn’t support proxy contracts or diamonds. The particular layout of storage is not defined in this EIP, but may be defined by later proposals. Examples of storage layout patterns that work with diamonds are Diamond Storage and AppStorage.”

Here is a snippet of the Diamond Storage contract, mentioned in the EIP-2535, which is the diamond storage standard later defined in the EIP-7201:

```
bytes32 constant ERC721_POSITION = keccak256("erc721.storage");

struct ERC721Storage {
    // tokenId => owner
    mapping (uint256 => address) tokenIdToOwner;
    // owner => count of tokens owned
    mapping (address => uint256) ownerToNFTokenCount;

    string name;
    string symbol;
}

// Return ERC721 storage struct for reading and writing
function getStorage() internal pure returns (ERC721Storage storage
storageStruct) {
    bytes32 position = ERC721_POSITION;
    assembly {
        storageStruct.slot := position
    }
}
```

III - Review 3

- **C1: It would have been nice to have numbers for Figure 6, in order to define upgradability or flexibility for instance. Because basically, if we want "some" upgradability we should choose Proxy, and for "a lot" of upgradability we should choose Diamond. Where is the threshold?**

We thank the reviewer for his feedback helping us to improve our paper.

More upgrades are planned for future work. This will help us define a better threshold for upgradeability.

We refer the reader to the answer to review 2.C2

- **C2: In some cases it is even more cost efficient to use Diamond patterns. This has not been addressed in the paper. Since minor updates are the cheapest, for a contract with many minor updates it is cheapest to use the Diamond pattern. It would have been interesting to compare costs based on the number of minor or major updates as well (These differences are called negligible in 5.4.).**

  To answer this concern, we added more details about upgrades and suppressed the use of the word “negligible” since it is too vague in section 5.4. Here is the edited paragraph in the evaluation section:

  "In reviewing the data obtained from the comparative analyses, the deployment phase exhibits the most significant variation among the different patterns, primarily due to the fluctuating number of smart contracts deployed. For this metric, the diamond pattern is the cheapest, especially for minor upgrades. During execution, the gas costs are similar across the patterns, though there is a marginal escalation in costs when transitioning from the classic architecture to the proxy pattern, and subsequently from the proxy to the diamond pattern. This increase in gas costs is due to the additional operations required to delegate calls to the logic contract(s) in the proxy and diamond patterns.”

  Moreover, we will conduct more upgrades in future work to have a clearer picture of the diamond pattern’s cost-effectiveness.

- **C3: The goal of the paper is to do a gas cost analysis of Proxy and Diamond patterns. I do not understand the role of Section 5.3 which only analyzes the addFile function, with no relevance to the Classic X Proxy X Diamond analysis.**

  We agree with the reviewer that this section was irrelevant to our study. The results were similar across the different patterns: it was just highlighting the usage of more storage slots, which is not important for that experiment. It is removed in the revised version of the paper.

- **C4: In Section 6.1, I do not understand the use of the term "Forces", since it is not the advantages of the pattern: it only describes an apparent dilemma with smart contracts.**

  We are sorry for this misunderstanding. Since we are following the Alexandrian format, which has a reference to its definition in our paper, we did not explain what it meant. To address this issue, we added the word “tradeoffs” next to “forces” to directly understand the following content.

- **C5: Form and format.**

  As mentioned in C1 for the first reviewer, those were inadvertent errors. The entire paper has been revised, addressing issues related to form, format, and grammar. We thank the reviewer for their thorough review.

We sincerely appreciate the valuable insights and constructive feedback provided by the reviewers, which have significantly contributed to enhancing the quality of our scientific paper.