Upgradeability: Good, Bad, Ugly

Mehdi Salehi, Jeremy Clark, and Mohammad Mannan

Concordia University

Abstract.

1 Introductory Remarks

2 Classification

2.1 Retail Changes

This is not a standardized pattern. The development team must consider the ways to upgrade the contracts before deploying the smart contract. Known patterns are different on the level of intervention to change the logic that they need to change in the future. The amount of changes are limited and system design can not be changed after deployment and just some system variables can be changed. We will describe three famous patterns here:

Parameter Configuration . The easiest way to upgrade the logic of the smart contract is to have some critical parameter that can change the whole logic of the system. For instance, in economy we have different variables which have effect in the interest rates. By changing those variables the governors will response to changes needed for the system. In this model we have a setter function to change the upgradable parameters if the system needs upgrades. The best example for this type of upgradeability is MakeDao project. In Maker there are some variables like Dai Saving Rate (DSR) or Stability fee that can be changed through governance vote. The logic behind the smart contract and the tokenomic of the Dapp completely depends on these variables.

Strategy pattern . The strategy pattern is an easy way for changing part of the code in a contract responsible for a specific feature. Instead of implementing a function in your contract to take care of a specific task, you call into a separate contract to take care of that and by switching implementations of that contract, you can switch between different strategies. An example for this pattern is Compound project and how they used strategy pattern for their interest rate model. There is a interest rate model contract in Compound that can be changed during the time.

Pluggable Modules . In this pattern we have a core contract that have some immutable features and then new contracts generated by the main contract and each have some or all features of the main contract. This pattern is mostly used in wallets and DeFi services like DeFi saver and InstaDapp. Users can decide to add new features into their wallet.

2.2 Wholesale Changes

In contrast to previous session, sometime we need to change the whole or a big part of the logic of our smart contract. This update could be a response to an incident happen to the smart contract or a planed upgrade of the system. Before deployment, the core developer team should have a plan for the upgrade events. We categorized these types into two main classes:

Contract Migration In the migration plan we should write a completely new contract with our desired new logic. In migration method our new version contract doesn't have any communication with the previous versions. The challenges we face in migration method are:

1. Grab the needed data (from previous contract or new data): It depends on the data type. It is easy for simple data structures (*e.g.,* uint, address, or even arrays) to collect the data just by reading storage slots from the 0 slot. we should take care of complex structures (*e.g.,* mapping) in the latest versions of our contract by adding event updates whenever a data added to a mapping variable. In case of an upgrade we can use Logs to find storage slot (using key hash) and collect the data. Sometime we need to push new data into our upgraded smart contract. For example in airdrops we need new coin to be added to some specific addresses.
2. Push the data into the new contract: Using Constructor, we can use batch transfer function with arrays of addresses and amounts as inputs. This way we can push lots of data using a single transaction. One limitation here is block gas limit. If we exceeds the block gas limit we need to push all data in different blocks (pausing in the first block and unpause at the end). Recently, Devs are using merkle distribution tree to push data on to the smart contracts. The most important thing here is the cost of pushing data to the new version. It depends on 1)the number of storage slots to be updated and 2)Method used to push the data on-chain. (Can be tested).
3. Stop the previous contract: Suppose that we have a token contract and we want to migrate to a new version. We should be confident that nobody can use the previous contract. If not, it is possible that a person sell a token from previous contract (which should be valueless after migration) to a person who doesn't aware of the migration plan.Because of the decentralized nature of the blockchain and Dapps you cannot reach to your customers to alert them from using the previous contract. One way to do that is have a pause option the your contracts and pause the old versions before migration.

Contract Migration is less riskier than other types of upgrades, not cost effective compare to some upgradeability types but more decentralized to the other solutions. Also, it's not good for frequent updates.The other advantage of this method is that it removes transaction gas cost needed for patterns like proxy, registry or call-based methods.

Data Separation patterns The other type of wholesale methods is to separate data and logic part of our codes. In the case of the upgrade we can keep the storage contract and just upgrade the logic contract and link the new version of logic contract to the storage contract. There is a debate on whether this type of upgradeability is cheaper or not in comparison to migration method. But, this method is more efficient for Dapps in which we need frequent updates. The other important issue here is who decides on the changes we need for the system which we will discuss on further sections. Here we have 2 different choices using Call method and Delegate Call method to link storage and logic contracts together.

Call based patterns In this type the interaction between logic and storage contact is handle by Call opcode in Ethereum Virtual Machine. In call based patterns user is supposed to call the logic contract and the logic contract will call the storage contract. The logic contract is the one that can be upgraded.

There are two concerns in this approach: how to store data and how to perform an upgrade.

Storage. The easiest way to store data in storage contract is to have a modifier on the setter functions in the storage contract that allow just the logic contract to change the variables. The owner of the contract can change the address of logic contract for the modifier. In this approach for adding a new persistent variable, a new data contract should be deployed which may be costly in case that the application needs lots of upgrades.

The other way to store data is so called Eternal storage (ERC930). Eternal storage uses mapping (key-value pair) to store data, using one mapping per type of variable. The EVM storage layout and how it handles mapping helps the Eternal storage pattern to be more amenable to evolution but also more complex.

Upgrade implementation. There are three main ways to implement upgrades using data separation pattern. The easiest way is to change the ownership of storage contract into new upgraded logic contract and then **Pause** the old contract or set its pointer to 0x0 address. The other solution is to forward the calls receive by the old contract into the new logic contract. The last option is to set a proxy contract that just keeps the address of logic contract and call into logic contract.

DelegateCall-based upgrades Similar to call based patterns here we have two contracts, Storage and Logic contract. we may have more than one logic contract. The difference here is that the user is calling storage contract first(called proxy contract), and the proxy contract DelegateCalls to the logic contract(s). The main difference between delegatecall and call-based approach is that in delegatecall proxy pattern the storage layout of proxy and logic contract should be the same. The difference between storage layouts will result in storage clashes. There are three different methods to mitigate the risk of storage clashes:

Inherited Storage. In this method the proxy contract and all logic contracts are inherited from a storage contract that contains storage variables. Using this method we are confident that the proxy and logic contracts are using the same

storage layout and storage clashes will be mitigated. After deployment if we need new logic contract with new storage variables, we should deploy a new storage contract that inherits the previous storage contract. Then the new logic contract must inherit the new version of the storage contract.

This method is not efficient because of variables that declared but not used in some logic contracts. On the other hand, each logic contract is coupled with a storage contract and it is hard to take care of this track.

Eternal Storage. In this pattern, we defined mappings for all variable types that we need to use in our logic smart contract. For storing mapping variables EVM selects random slots on the storage based on the variable's name so we can mitigate the clashes using this randomness.

The main problem of this type is that the logic contract and all other contracts that are using the storage must use the mapping structure to access the storage variables and use complex syntax whenever they want to access a variable. This also results in the gas usage inefficiency because we need to call and update a mapping each time we need to change a variable.

Also it is hard to use eternal storage for complex variables like mappings and structure (need mapping of mapping pattern). Also finding a state variable of the proxy smart contract is hard because we store them in arbitrary slots of the storage.

Unstructured Storage. The other way of mitigating the storage clashes is to assign some randomly selected slots to critical variables like address of logic contract. For instance, open zeppling uses hash of "org.zeppelinos.proxy.implementation" to store the address of the logic contract in this slot.

The downside of this approach is that we need getter and setter function for each variable. We also can use unstructured storage for simple variables and not for mapping and structures.

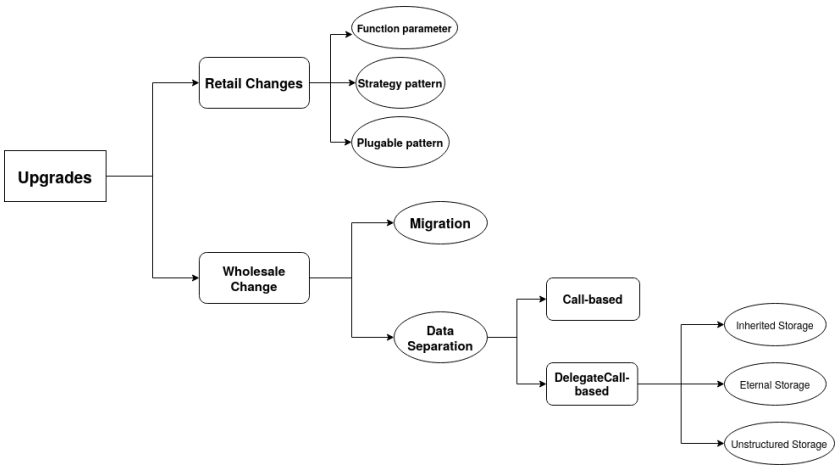

Fig. 1. Classification

3 Evaluation of different methods

In this section we compare and evaluate different methods discussed in previous section. There are some characteristics that can help the designer to decide which method should be used on the system and add upgradeability to the Dapp.

For our evaluation framework, we provide the definition of each evaluation criteria (i.e., column of the table), specifying what it means to receive a full dot (●), partial dot (◐), an empty dot (○) or to receive a square (⊠).

3.1 Extent of Upgradeability

An upgradeability method in which the upgrader can change the whole logic of the system earns a full circle(●). A method that could be used to change a small part of the system awarded an empty circle(○).

Retail Change methods limit the the upgrades to a small part of the system (○). However, using *Migration*, *Call-based* and *DelegateCall-based* approaches, the upgrader is able to deploy a new smart contract with a new set of functions and logics.

3.2 Upgrade Frequency

An upgradeability method that can move from proposal to finality within a single transaction is awarded a full dot (●). A process that requires more than a single block time in order to write and deploy a new logic contract receives half circle (◐). A method that needs more than just deploying a new logic contract and changing some parameters awarded empty circle (○).

The upgrader can process the whole upgrade using *Retail changes* method in a single transaction time window after proposing the change (●). On the other hand, *Call-based* and *DelegateCall-based* approaches are more time consuming because the upgrader needs a time window to write the new logic contract, deploy it to the mainnet, and change some address pointers (◐). *Migration* method is more time consuming and not good for frequent changes because Before the upgrade the developers need to collect data of old contract (*e.g.,* getting storage snapshot), write a new version of the logic smart contract, and push the old data into the newer version which is time consuming and sometimes it takes multiple blocks to push all data from the old version (because of block gas limit of a block) (○).

3.3 System Complexity

Applying each upgradeability method has different complexity effects on the system. An upgradeability method in which the designer do not need to change any logic of the contract receives empty circle (○). Upgradeability methods that cause changing the whole design or the logic of the smart contract but do not add possible security flaws or attack surfaces receive half circle (◐). Upgradeability

methods that force the developer to change the whole logic of the smart contract and adds considerations to smart contract security receives full circle (●).

Migration method does not add any complexity to the system. The system designer do not need to change considerable part of system design to have feature to migrate to the new version before deployment. So, it is the less complex choice that a designer can take (○).

In *Retail changes* method, it is hard and complex for system designers to design a system in which a parameter or a small part of system can change the logic of the system and be safe to change it but it does not add security considerations (◐).

In the *Call-based* upgradeability approach, the system designer must separate storage and logic pieces of the contract and change the whole design of the smart contract. It adds complexity to the system to be confident that these two are separated correctly. But it does not add considerable new attack surfaces to the system (◐).

On the other hand, *DelegateCall-based* patterns adds more complexity to the system because of using DelegateCall opcode as well as need to separating logic and storage contract. The main difference between *Call-based* and *DelegateCall-based* approach is that the developer should take care that using Delegatecall opcode needs to take care that the logic and storage contract must have similar storage layout. So using DelegateCall pattern changes the design of the contract and adds new attacks surfaces to the system (●).

3.4 Transaction Cost

An upgradeability pattern that adds an extra gas cost needed to be paid by users for a specific operation receives full circle (●), otherwise it awarded empty circle (○).

Retail Changes and *Migration* methods do not change the logic of a smart contract generally and so will not alter the transaction costs (○). However, *Call-based* and *DelegateCall-based* approaches increase the transaction cost for users to perform a specific operation (●). *Call-based* method will increase the transaction cost because whenever a transaction needs to add or change a data, the logic contract must **CALL** the storage contract which adds an extra gas. Also in *DelegateCall-based* approach all transactions will be **Delegate Called** to the logic contract using a **proxy** which adds an extra gas on each transaction.

We need a comparison between Call based and Delegate call based approaches (if possible. I think Call based approach is gas efficient for systems that do not need to add or change data frequently. Also there are other extra adds-on that can add useability but adds gas like adding feature for uninformed users in Call based approaches.)

3.5 Cost of Upgrade

One of the main differences between upgradeability approaches is how much does the upgrade process costs. An upgradeability pattern that costs like a nor-

mal transaction (*e.g.,* changing a parameter or a more complex variables like structures) receives an empty circle (○). Upgradeability methods that needs to deploy a new logic contract but do not need to push old data into it receive half circle (◐) and patterns in which the upgrader should deploy a new logic contract and push old data into the new version is the most expensive approach in the upgrading process and awarded full circle (●).

3.6 Useability

An upgradeability approach in which users do not feel any changes and do not need to take any action after the upgrade process receives a full circle (●). In patterns that the users just need to call a new smart contract address rewarded a half circle (◐). In some patterns although the users should take some actions in addition to using a new address (○).

In *Retail Change* approach the user does not feel the upgrade process unless she has a exposure to the parameter that is changing on the upgrade (●). In *Call-based* method, the logic contract (the contract that is called by users) will be changed. So, users must call a new contract after the upgrade which is not user friendly (◐). There are some approaches to mitigate this by using a proxy contract in between. The other way to mitigate this is to implement a way in which an old logic smart contract can call the newer version and pass the user's request to the newer version. These solutions add an extra gas to each transaction which is not usable as well. In *DelegateCall-based* approach users are calling the proxy contract which delegate call their request to the whitelisted logic contract. In the upgrade event the developers change the whitelisted address to the new version but users call the proxy like before. So, users do not feel the upgrading process (●).

On the other hand, users in a *Migration* plan need to work with a new smart contract . In some Migration plans, users themselves must withdraw their fund from the previous version and deposit it on the new version which is not user friendly and costly for them (○).

3.7 Fixing a Bug

As mentioned in the previous part, upgradeability could be used in two different situations; adding new features or fixing a bug. An upgradeability pattern that cannot be used for fixing a bug receives square (⊠). An upgradeability pattern which can be used for fixing a bug but not suitable for fixing a bug receives an empty circle (○) and the methods which are suitable for fixing a bug receives a full circle (●).

In an incident, *Retail Changes* approach won't help to response to a bug or hack because the extent of upgradeability is limited and the developers are not able to change the required parts of the system (⊠). *Migration* is not proper as well because it is not quick enough to respond in a limited time window. On the other hand, *Call-based* and *DelegateCall-based* are very well for fixing a bug. In

the event, the developers can find the root cause and patch it by deploying a new contract and change the implementation address in the logic contract.

3.8 Speed of an Upgrade

Another difference between upgradeability methods is the speed in which an upgrade can be processed. This depends on the type of decision makers which will be discussed in 4.1.

Retail changes method is the fastest way to upgrade a system comparing to other methods. Using an EOA as the decision maker is the fastest option of an upgrade. Using multi-sig is a bit slower than using EOA. Utilizing a decentralized governance scheme to decide about upgrades will put an inherit time delay to the upgrades.

Migration has the slowest upgrade process between other methods. The reasons are discussed in the previous parts (see 3.2).

Call-based and *DelegateCall-based* are very similar to each other in the speed of upgrade. These two are not as quick as *Retail changes* because the developer needs to find the root cause of a bug or find the upgrades needed for system and then implement the smart contract and deploy it to the system which is time consuming. On the other hand these two approaches are faster than *Migration* because as mentioned before, in Migration plans we need to collect and push old data into newer version as well.

3.9 Level of Decentralization

The last and one of the most important characteristics that are different in upgradeability methods is the level of decentralization. An upgradeability methods that a single third party decides the upgrading process receives a square (⊠). Using an **EOA** to decide about a change is the most central option that a system designer can choose regardless of the upgradeability method uses in the system. In case a group of whitelisted persons can decide on the changes orf the system using **Multi-sig** is not decentralized as well. Although it improves the level of decentralization of the system but at the end a specified number can decide to change the system. So it awarded an empty circle (○).

Utilizing a decentralized governance model to vote for a change is a good way to make the decision making on the upgrades more decentralized. *Retail Changes* using voting scheme is more decentral than *Call-based* and *DelegateCall-based* because boundary of changes are limited on the Retail methods so it awarded a full circle (●). But, in CAll-based and DelegateCall based methods the developers have the power to put some kind of backdoor in the system while upgrading and they receive a half circle (◐).

The *Migration* method is the most decentralized approach because it gives the users chance to decide whether to move to the newer version or not so it awarded a full circle (●). For instance, Uniswap uses this method for its upgrade and the users have choice to transfer their funds from Uniswap V2 to V3 or not and as we can see some users decide to stay on the previous version.

4 Upgrading process

4.1 decision maker(s)

There is a debate on who is responsible for upgrading a Dapp. Different systems can choose one of these schemes to upgrade their Dapp depending on the complexity of the system, frequency of the changes needed for the system and how fast does the system need to upgrade in the incidents.

Externally owned Address The easiest and the fastest way to upgrade a system is through a single address which is the owner of smart contract. This is the most centralized solution we have for upgrading a system. The main problem with this issue is the security of the system because it only depends on a single private key hold by the owner. In case of malicious party or if an attacker find the owner's private or if the owner lose the key the entire system is on the risk.

First Dapps on the ethereum blockchain used this method for the upgrade but it is not used these days because it is far from the idea of *Decentralization*.

Multi-Sig A *m out of n* Multi-sig wallet is a smart contract that can manage a transactions only if m number out of a specified n EOAs agree and sign the transaction. We can use address of a Multi-sig wallet as the owner of the system. In case of a upgrade or responding to an incident m number of the governors can permit to upgrade the system.

This is a better answer to the decision making of the upgrade compare to using an EOA in case of centrality while keeping the speed of an upgrade process. However, it is not decentralized. One way to reduce the level of centralization is to use different trusted teams who are stakeholders of the system in the multi-sig wallet.

Governance Voting The most decentralized way to decide on a system change is to do it using decentralized voting. This can be done by distributing governance voting tokens to the community and then they can vote on a change proposal by staking their voting token.

There are some critique to this method. Governance by voting has an inherit time delay to the upgrading process. This raises a problem when the system needs an instant upgrade (*e.g.,* responding to an incident). This means we need another mechanism to quickly fix bugs and upgrade the system on the event of incidents in conjunction with the voting process (*e.g.,* Global shutdown in MakerDAO).

It is also not cost-efficient for the voters because all token holders must send a transaction and needs to pay network fee.

The other problem with this method is fair distribution of the tokens. If the governance token does not distribute fairly and the majority of tokens granted to the limited number of users, then it is very similar to the multi-sig method which is more costly and complex. Because, whales of the governance token can vote to any desired change of the system similar to the multi-sig.

4.2 Mitigating risks

There are critical setups on the systems to mitigate the possible risks on the upgrading process. We mention some of them here with risk associated with them.

Timelocks In some project, there is a time window between every changes that approved on the system and when they affect the system. This gives opportunity to the users who are not satisfied with the upcoming upgrades to move their funds out of the system. However this is not proper in case of fixing a bug, because we need to patch the problem quickly.

Thereshold In multi-sig and governance upgrade methods we need a threshold on votes to decide whether a change is approved or not. This threshold should be big enough to be confident that upgrading event represents the majority of opinions. On the other hand, the threshold shouldn't be that big because a big threshold will delay a system change. The system designer should consider that a portion of voters (signers in multi-sig or governance token holders in voting method) may not be available in the event of the upgrade and having a big threshold may result in halting the change proposal for a long period of time. In fact threshold has a trade-off between security/decentralization and speed of the upgrade process.

Pausable In pauseable smart contracts, the decision makers (usually a multi-sig wallet) can freeze some or all operations of the system. Pausing a smart contract helps in some specific situations:

1. Time to react to a bug or hack: usually it takes time to analyze and find the reason of a hack and patch the bug. In this time period the core developer team needs to pause the system to stop attacker from draining all the fund.
2. Halting system in the upgrade process: For instance, in an ERC20 token contract upgrade we need to pause the system to stop users from transferring tokens during the upgrade.
3. Inactivating the previous version of the logic contracts: After an upgrade we need to have a plan to stop users from using the previous logic contracts. One way to do so is to make the logic contracts pauseable and pause them after the upgrade.

Escape Hatches A escape hatch is a mechanism that lets the users to move their fund out of the system in the pausing events. For instance, in MakerDAO we have an emergency shutdown mechanism that pauses the system in the black swan events. But, users have the ability to extract their funds out of the system while the system is paused.

Front-Runnign Upgrading a smart contract can be done by sending a transaction into the system. If the upgrade is a response to a unknown bug, then the upgrade process will hint attackers who is listening to the mempool to find the bug and hack the smart contract just before the upgrade. So there should be some mechanisms to mitigate front-running attacks. One solution to this issue is to use commit-reveal schemes. The team first sends a commitment of the upgrade (hash of the upgrade) to the system and after the timelock they can push and apply the original code which cannot be front run.

5 Measurement study

6 discussion

	Extent of Upgradeability	Upgrade Frequency	System Complexity	Transaction Cost	Cost of Upgrade	Useability	Fixing a Bug	Decision Maker	Speed of an Upgrade	Level of Decentralization
Retail changes	○	●	◐[1]	○	○	●	⊠	EOA	10	⊠
								Multi-Sig	9	○
								Voting	6	●
Migration	●	○	○	○	●	○	○	Not Important	3	●
Call-based	●	◐	◐	●	◐	◐	●	EOA	8	⊠
								Multi-Sig	7	○
								Voting	4	◐
DelegateCall-based	●	◐	●	●	◐	●	●	EOA	8	⊠
								Multi-Sig	7	○
								Voting	4	◐

Table 1. Evaluation

References

[1] Design of system in which a parameter can change the logic is hard