---

title: Upgrading smart contracts description: An overview of upgrade patterns for Ethereum smart contracts lang: en

---

Smart contracts on Ethereum are self-executing programs that run in the Ethereum Virtual Machine (EVM). These programs are immutable by design, which prevents any updates to the business logic once the contract is deployed.

While immutability is necessary for trustlessness, decentralization, and security of smart contracts, it may be a drawback in certain cases. For instance, immutable code can make it impossible for developers to fix vulnerable contracts.

However, increased research into improving smart contracts has led to the introduction of several upgrade patterns. These upgrade patterns enable developers to upgrade smart contracts (while preserving immutability) by placing business logic in different contracts.

Prerequisites {#prerequisites}

You should have a good understanding of smart contracts, smart contract anatomy, and the Ethereum Virtual Machine (EVM). This guide also assumes readers have a grasp of programming smart contracts.

What is a smart contract upgrade? {#what-is-a-smart-contract-upgrade}

A smart contract upgrade involves changing the business logic of a smart contract while preserving the contract's state. It is important to clarify that upgradeability and mutability are not the same, especially in the context of smart contracts.

You still cannot change a program deployed to an address on the Ethereum network. But you can change the code that's executed when users interact with a smart contract.

This can be done via the following methods:

1. Creating multiple versions of a smart contract and migrating state (i.e., data) from the old contract to a new instance of the contract.

2. Creating separate contracts to store business logic and state.

3. Using proxy patterns to delegate function calls from an immutable proxy contract to a modifiable logic contract.

4. Creating an immutable main contract that interfaces with and relies on flexible satellite contracts to execute specific functions.

5. Using the diamond pattern to delegate function calls from a proxy contract to logic contracts.

Upgrade mechanism #1: Contract migration {#contract-migration}

Contract migration is based on versioning—the idea of creating and managing unique states of the same software. Contract migration involves deploying a new instance of an existing smart contract and transferring storage and balances to the new contract.

The newly deployed contract will have an empty storage, allowing you to recover data from the old contract and write it to the new implementation. Afterward, you will need to update all contracts that interacted with the old contract to reflect the new address.

The last step in contract migration is to convince users to switch to using the new contract. The new contract version will retain user balances and addresses, which preserves immutability. If it's a token-based contract, you will also need to contact exchanges to discard the old contract and use the new contract.

Contract migration is a relatively straightforward and safe measure for upgrading smart contracts without breaking user interactions. However, manually migrating user storage and balances to the new contract is time-intensive and can incur high gas costs.

More on contract migration.

Upgrade mechanism #2: Data separation {#data-separation}

Another method for upgrading smart contracts is to separate business logic and data storage into separate contracts. This means users interact with the logic contract, while data is stored in the storage contract.

The logic contract contains the code executed when users interact with the application. It also holds the storage contract's address and interacts with it to get and set data.

Meanwhile, the storage contract holds the state associated with the smart contract, such as user balances and addresses. Note that the storage contract is owned by the logic contract and is configured with the latter's address at deployment. This prevents unauthorized contracts from calling the storage contract or updating its data.

By default, the storage contract is immutable—but you can replace the logic contract it points to with a new implementation. This will change the code that runs in the EVM, while keeping storage and balances intact.

Using this upgrade method requires updating the logic contract's address in the storage contract. You must also configure the new logic contract with the storage contract's address for reasons explained earlier.

The data separation pattern is arguably easier to implement compared to contract migration. However, you'll have to manage multiple contracts and implement complex authorization schemes to protect smart contracts from malicious upgrades.

Upgrade mechanism #3: Proxy patterns {#proxy-patterns}

The proxy pattern also uses data separation to keep business logic and data in separate contracts. However, in a proxy pattern, the storage contract (called a proxy) calls the logic contract during code execution. This is a reverse of the data separation method, where the logic contract calls the storage contract.

This is what happens in a proxy pattern:

1. Users interact with the proxy contract, which stores data, but doesn't hold the business logic.
2. The proxy contract stores the address of the logic contract and delegates all function calls to the logic contract (which holds the business logic) using the `delegatecall` function.
3. After the call is forwarded to the logic contract, the returned data from the logic contract is retrieved and returned to the user.

Using the proxy patterns requires an understanding of the**delegatecall** function. Basically, `delegatecall` is an opcode that allows a contract to call another contract, while the actual code execution happens in the context of the calling contract. An implication of using `delegatecall` in proxy patterns is that the proxy contract reads and writes to its storage and executes logic stored at the logic contract as if calling an internal function.

From the Solidity documentation:

> *There exists a special variant of a message call, named**delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context (i.e. at the address) of the calling contract and `msg.sender` and `msg.value` do not change their values. This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.*

The proxy contract knows to invoke`delegatecall` whenever a user calls a function because it has a`fallback` function built into it. In Solidity programming the fallback function is executed when a function call does not match functions specified in a contract.

Making the proxy pattern work requires writing a custom fallback function that specifies how the proxy contract should handle function calls it does not support. In this case the proxy's fallback function is programmed to initiate a delegatecall and reroute the user's request to the current logic contract implementation.

The proxy contract is immutable by default, but new logic contracts with updated business logic can be created. Performing the upgrade is then a matter of changing the address of the logic contract referenced in the proxy contract.

By pointing the proxy contract to a new logic contract, the code executed when users call the proxy contract function changes. This allows us to upgrade a contract's logic without asking users to interact with a new contract.

Proxy patterns are a popular method for upgrading smart contracts because they eliminate the difficulties associated with contract migration. However, proxy patterns are more complicated to use and can introduce critical flaws, such as function selector clashes, if used improperly.

More on proxy patterns.

Upgrade mechanism #4: Strategy pattern {#strategy-pattern}

This technique is influenced by the strategy pattern, which encourages creating software programs that interface with other programs to implement specific features. Applying the strategy pattern to Ethereum development would mean building a smart contract that calls functions from other contracts.

The main contract in this case contains the core business logic, but interfaces with other smart contracts ("satellite contracts") to execute certain functions. This main contract also stores the address for each satellite contract and can switch between different implementations of the satellite contract.

You can build a new satellite contract and configure the main contract with the new address. This allows you to change *strategies* (i.e., implement new logic) for a smart contract.

Although similar to the proxy pattern discussed earlier, the strategy pattern is different because the main contract, which users interact with, holds the business logic. Using this pattern affords you the opportunity to introduce limited changes to a smart contract without affecting the core infrastructure.

The main drawback is that this pattern is mostly useful for rolling out minor upgrades. Also, if the main contract is compromised (e.g., via a hack), you cannot use this upgrade method.

Upgrade mechanism #5: Diamond pattern {#diamond-pattern}

The diamond pattern can be considered an improvement on the proxy pattern. Diamond patterns differ from proxy patterns because the diamond proxy contract can delegate function calls to more than one logic contract.

The logic contracts in the diamond pattern are known as *facets*. To make the diamond pattern work, you need to create a mapping in the proxy contract that maps function selectors to different facet addresses.

When a user makes a function call, the proxy contract checks the mapping to find the facet responsible for executing that function. Then it invokes `delegatecall` (using the fallback function) and redirects the call to the appropriate logic contract.

The diamond upgrade pattern has some advantages over traditional proxy upgrade patterns:

1. It allows you to upgrade a small part of the contract without changing all of the code. Using the proxy pattern for upgrades requires creating an entirely new logic contract, even for minor upgrades.

2. All smart contracts (including logic contracts used in proxy patterns) have a 24KB size limit, which can be a limitation—especially for complex contracts requiring more functions. The diamond pattern makes it easy to solve this problem by splitting functions across multiple logic contracts.

3. Proxy patterns adopt a catch-all approach to access controls. An entity with access to upgrade functions can change the *entire* contract. But the diamond pattern enables a modular permissions approach, where you can restrict entities to upgrading certain functions within a smart contract.

More on the diamond pattern.

Pros and cons of upgrading smart contracts {#pros-and-cons-of-upgrading-smart-contracts}

| Pros | Cons | | --- | -- | | A smart contract upgrade can make it easier to fi vulnerabilities discovered in the post-deployment phase. | Upgrading smart contracts negates the idea of code immutability, which has implications for decentralization and security. | | Developers can use logic upgrades to add new features to decentralized applications. | Users must trust developers not to modify smart contracts arbitrarily. | | Smart contract upgrades can improve safety for end-users since bugs can be fixed quickly. | Programming upgrade functionality into smart contracts adds another layer

of complexity and increases the possibility of critical flaws. | | Contract upgrades give developers more room to experiment with different features and improve dapps over time. | The opportunity to upgrade smart contracts may encourage developers to launch projects faster without doing due diligence during the development phase. | | | Insecure access control or centralization in smart contracts can make it easier for malicious actors to perform unauthorized upgrades. |

Considerations for upgrading smart contracts {#considerations-for-upgrading-smart-contracts}

1. Use secure access control/authorization mechanisms to prevent unauthorized smart contract upgrades, especially if using proxy patterns, strategy patterns, or data separation. An example is restricting access to the upgrade function, such that only the contract's owner can call it.

2. Upgrading smart contracts is a complex activity and requires a high level of diligence to prevent the introduction of vulnerabilities.

3. Reduce trust assumptions by decentralizing the process of implementing upgrades. Possible strategies include using a multi-sig wallet contract to control upgrades, or requiring members of a DAO to vote on approving the upgrade.

4. Be aware of the costs involved in upgrading contracts. For instance, copying state (e.g., user balances) from an old contract to a new contract during contract migration may require more than one transaction, meaning more gas fees.

5. Consider implementing **timelocks** to protect users. A timelock refers to a delay enforced on changes to a system. Timelocks can be combined with a multi-sig governance system to control upgrades: if a proposed action reaches the required approval threshold, it doesn't execute until the predefined delay period elapses.

Timelocks give users some time to exit the system if they disagree with a proposed change (e.g., logic upgrade or new fee schemes). Without timelocks, users need to trust developers not to implement arbitrary changes in a smart contract without prior notice. The drawback here is that timelocks restrict the ability to quickly patch vulnerabilities.

Resources {#resources}

OpenZeppelin Upgrades Plugins - *A suite of tools for deploying and securing upgradeable smart contracts.*

- GitHub
- Documentation

Tutorials {#tutorials}

- Upgrading your Smart Contracts | YouTube Tutorial by Patrick Collins
- Ethereum Smart Contract Migration Tutorial by Austin Griffith
- Using the UUPS proxy pattern to upgrade smart contracts by Pranesh A.S
- Web3 Tutorial: Write upgradeable smart contract (proxy) using OpenZeppelin by fangjun.eth

Further reading {#further-reading}

- The State of Smart Contract Upgrades by Santiago Palladino
- Multiple ways to upgrade a Solidity smart contract - Crypto Market Pool blog
- Learn: Upgrading Smart Contracts - OpenZeppelin Docs
- Proxy Patterns For Upgradeability Of Solidity Contracts: Transparent vs UUPS Proxies by Naveen Sahu
- How Diamond Upgrades Work by Nick Mudge