There's two ways to do contract upgrades under consideration: delegatecall

with proxies, and enshrined via slow state. The slow updates tree has not been seriously considered for enshrining, and proxies were therefore the only remaining solution. The recent development of SharedMutableStorage (aka SlowJoe) however enables upgrades by just enshrining storage of a contract class id in a specific slot (like EIP-1967 does) with some special read/write semantics, reviving this discussion.

This document explores what it'd take to follow the path of the EVM and do upgrades via proxies and delegatecall, covering known issues we'd inherit as well as the changes that would need to be implemented. The alternative solution via slow state is more thoroughly discussed here.

About half the links in this document go to articles or standards written by Palla. Thanks Palla for being awesome.

Problem Statement

We want to be able to fully upgrade a contract. This means changing the behavior of current functions, removing existing functions and adding new functions, all while preserving the current state and address. We also want for an upgradeable contract to look exactly the same as a non-upgradeable one to an outside observer: calls and return values should be handled the same, etc.

For the most part, this is achievable on the EVM using proxies. Aztec currently lacks some of the features required to pull this off though: to begin with, a trivial non-upgradeable proxy requires [both a fallback function and delegatecall

](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/aed22fbc2203a59cf8d092e8f2b51cc534d79355/contracts/proxy/Proxy.sol).

Fallback Function

The [fallback

function](https://docs.soliditylang.org/en/latest/contracts.html#fallback-function) is a contract function that gets executed if the selector matches no other function. This is a Solidity concept, not an EVM one, because there are no functions on the EVM: it is Solidity that implements dispatching via the function selector. Things don't work the same in Aztec however, so implementing this feature is a bit trickier.

Note that while the fallback function might seem at first like a generic construct that'd be useful regardless of proxies, we have not found any other sensible use cases that'd warrant inclusion of this feature.

Implementing fallbacks requires:

- a private fallback function, called by the private kernel when presented with a non-inclusion proof of the function selector.
- a public fallback function, inserted by the transpiler into the function dispatch logic (akin to the Solidity case).
- an unconstrained fallback function, called by the simulator.

All three of this must be able to return variable-length data, since the proxy usecase demands that the fallback returns whatever was returned by the underlying contract, and each contract function might return differently sized data.

They also must be able to somehow reference the variable-length data they received, possibly via args_hash

.

delegatecall

We'd probably do this slightly differently from Ethereum bydistinguishing contract classes and instances, but the core mechanism remains be the same: make a call preserving some of the current context (notably sender and contract addresses).

Similar to the fallback scenario, the only reasonable use case we've found for delegatecall

is contract upgrades. Other use cases are:

- Hacky workarounds to EVM gas and bytecode size constraints.
- This is usually caused by monolothic contracts and poor application architecture. It's usually preferred to split up a very complex system into multiple contracts.

- Even as a workaround, using delegatecall

for this is quite difficult to develop and maintain.

- This is usually caused by monolothic contracts and poor application architecture. It’s usually preferred to split up a very complex system into multiple contracts.

- Even as a workaround, using delegatecall

for this is quite difficult to develop and maintain.

- One-time “scripts” to be executed on behalf of a governing contract.

- This can be solved by enqueuing calls in an entrypoint-like function, like the one we already have in account contracts.

- Or by implementing “scripts” in private-land, though public-land would not be covered.

- Even then, this is far from the only way to get generic script-like behavior. Many large systems such as MakerDAO or Balancer achieve this by simply granting the required permissions to a given script contract, which provides more granular control as a nice side-effect.

- This can be solved by enqueuing calls in an entrypoint-like function, like the one we already have in account contracts.

- Or by implementing “scripts” in private-land, though public-land would not be covered.

- Even then, this is far from the only way to get generic script-like behavior. Many large systems such as MakerDAO or Balancer achieve this by simply granting the required permissions to a given script contract, which provides more granular control as a nice side-effect.

- The [Multicall

](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.0/contracts/utils/Multicall.sol) pattern * We don’t need this since user accounts are smart accounts and not EOAs.

- We don’t need this since user accounts are smart accounts and not EOAs.

Since we have three execution modes, we again require three flavors of delegatecall

: private, public, and unconstrained. Note that these are actual calls, meaning they take up a call slot in private increasing proving time, and increase transaction costs in public.

Additionally, each of the three flavors must be able to pass variable-sized data to the callee, and receive variable-sized data from it. This is because the proxy needs to be able to call contract functions with arbitrary signatures. This is done differently in private (where we pass args_hash

and unpack via an oracle call) and in public (where presumably the approach would be similar to Solidity’s).

Aztec-specific Issues

The proxy needs to store the implementation address (public data) in such a way that it can be used in private without contention, while still being mutable. The current proposal for implementing this kind if storage is called Shared Mutable State, aka SlowJoe (see this forum post for a detailed explanation of alternative approaches and why they don’t work).

This mechanism would exist only at the application level and not be enshrined. However, the undesirable consequences of using this mechanism (e.g. ‘downtime’ close to an update epoch) are the same as when enshrining upgrades themselves.

The sole exception are account contracts, in which we can get away with using private storage (e.g. a Singleton) since contention is not an issue there. Still, using a Singleton for storing the current class id for an account contract means that every tx sent from that account requires an extra commitment and nullifier to be emitted.

Known Issues with Solidity Upgrades

There’s actually many ways to do Solidity upgrades - we’re going to focus on the Universal Upgradeable Proxy Standard (UUPS). For a (very) detailed exploration of the entire upgrades space, see this work of art.

While Solidity proxies do allow for fully upgradable contracts, this pattern famously has a bit of a bad reputation in the community. This is in no small part due to how difficult and error-prone it is to develop upgradable contracts. There’s multiple causes to this, some of which we can avoid, and others which are inherent to proxies.

Avoidable Issues

The vast majority of contracts are written in Solidity, which due to historically opposing support for contract upgrades results in unaddressed problems that would entirely solvable by the compiler providing the required tools.

These two large pain points are major reasons why upgradable contracts are considered hard and dangerous to write:

- constructors are not exposed, which due to their implementation as EVM creation code makes it impossible to use constructors with upgradeable contracts. Contracts are written to be either upgradeable or non-upgradeable. Libraries such as OpenZeppelin Contracts require an entire fork that deals with this issue.

- the storage layout cannot be user-defined, forcing users to resort to hacks in order to avoid slot collisions.

These fall under the umbrella of aztec.nr, and can (should!) be properly tackled by us. Note that we'll need to do this regardless of whether we use delegatecall

or enshrine upgrades.

Inherent delegatecall

Obscurity

These are consequences of using upgradeable proxies and delegatecall

, assuming one follows state of the art design patterns resulting from 6+ years of research. Even if a proxy contract is correctly developed and deploted, it will always introduce some level of obscurity and confusion due to the underlying truth of users not interacting with the actual contract logic, but instead with a proxy.

Consider for example the Etherscan code view for USDC, which doesn't show the token source code nor list its ABI. Both the proxy and implementation methods are exposed (under 'view as proxy'), which is very confusing to users since they likely do not know what a proxy is, and may think they've followed the wrong link somewhere.

Events are also problematic, since decoding them requires using the underlying implementation ABI (which may change!). Etherscan handles the USDC events by realizing that the signature looks similar to the standard ERC20 events, and interprets them as such.

EIP-1967 attempts to standardize this, partially to ease development of block explorers, but as a standard it is merely a suggestion. A naive block explorer could even be tricked by a malicious proxy contract into displaying an incorrect implementation contract by populating this standardized storage slot with garbage, potentially tricking end users and resulting in scams and theft.