The Complete Guide to Account Abstraction in Ethereum (Part 1)

Having observed the significant changes that were introduced to Ethereum through the Deneb upgrade, we have begun looking ahead to what the next hardfork, called Pectra, will introduce. To help shape the discussions to come, we seek to describe the current landscape of account abstraction surrounding Ethereum and its rollup ecosystem to potentially map a clear path forward.

Account abstraction seeks to improve user and developer experiences across the entire Ethereum ecosystem. Along with making on-chain experiences more accessible and enjoyable to users, it also empowers developers to be able to do more powerful things on Ethereum and serve users in even more meaningful ways. In this three-part series we will be comprehensively covering the account abstraction landscape within Ethereum, carefully exploring different approaches and solutions proposed along their lines and thereafter critically evaluating whether they are serious candidates for consideration in adopting within Ethereum whether it be a protocol change or set of standards the community should collectively start to follow.

An overview of the account abstraction landscape

Our classification of the approaches to account abstraction is as follows:

1. **EOA programmability**: Protocol-level changes that enable EOAs to redefine the execution logic portion of their validity rules.
2. **EOA conversion/migration**: Complete conversion of EOAs to CAs.
3. **Smart Accounts**: Designs that enable both EOAs and CAs to behave as "smart accounts" by allowing them to totally redefine their validity rules.

With **EOA programmability**, this involves considering changes to make to how EOA (externally-owned accounts) work and introducing new capabilities. As well-known within the development community, EOAs are accounts typically associated with end-users, therefore solutions that fall within this approach will empower end-user accounts with more control over what sort of actions they can authorize compared to how this can be managed today.

With **EOA conversion/migration**, The integral idea of this approach is that contract accounts already offer most of the benefits offered by smart accounts, so there should be no need to complicate things anymore; everyone should simply use a contract account as their primary account (through smart contract wallets).

With **Smart Accounts**, this approach features mechanisms which allow an EOA to transition to a CA, without having to move its assets, such as EIP 7377 and EIP 5003 (when considered alongside EIP 3074). Various proposals have been previously made for the creation of smart accounts and account abstraction enshrinement at the protocol level; EIP-86 and EIP-2938 are some of the more cited ones. However, there was a lot of pushback due to the perceived complexity introduced by this design and the somewhat majority opinion that Ethereum isn't ready for such complexity. Following Vitalik's revival of the topic after The Merge, ERC-4337 was proposed as an opt-in version of the smart account standard, similar to the PBS (Proposer-Builder Separation) infrastructure for MEV (Maximal Extractable Value). Thus, users who seek to access the benefits of smart accounts could simply use the 4337 pipeline to redefine their account's logic and transactions' validity rules in structures referred to as a UserOperation (or UserOps for short). ERC 4337 brings the benefits of smart accounts to present day Ethereum without enshrining any of the complexity, by functioning as an out-of-protocol alternative to enshrined smart accounts. However, this doesn't mean that the infrastructure is optimal in its current state as its own complexity is still a considerable point-of-failure. To address this complexity, RIP 7560 was drafted as an enshrined version of the ERC 4337 infrastructure across Ethereum and its L2s, so that it inherits the network's sybil-resistance schemes rather than having to define a new suite of rules (as ERC 4337 does with ERC 7562).

In this report, we will be focused on exploring EOA programmability covering the various EIPs that describe solutions along this line and discuss their merits and drawbacks. In Part 2 and 3 of this series, we will cover the remaining two classes of approach to account abstraction being explored within Ethereum.

A Primer on Ethereum Accounts and Transactions

In order to seek out what can be abstracted, we need a (somewhat) full picture of the current account design. This section will mostly serve as a revision of sorts for what accounts on Ethereum actually are, and how their transactions are validated and executed.

Ethereum accounts are entities with an ether (ETH) balance and the ability to send transactions on the Ethereum blockchain. They are represented as a 42-character hexadecimal "address", which serves as a unique pointer to the account's holdings and transactions.

An address acts as a key into the blockchain's state trie. The leaf nodes of this trie are account data structures which can be decomposed into four fields:

a. **nonce**: A linear counter used to indicate the number of outbound transactions initiated by an account. Also crucial in preventing replay attacks.

b. **balance**: The wei-denominated amount of ether owned by an account.

c. **codeHash**: A hash of the EVM-executable code contained in an account. The EVM (Ethereum Virtual Machine) is Ethereum's bespoke execution environment responsible for handling complex state transitions beyond simple “send” transactions. The code content of an account is immutably programmed to carry out specific forms of state transition on the Ethereum blockchain, through the EVM.

d. **storageHash**: A hash of an account's storage root, used to represent the storage contents of an account as a 256-bit hash of a merkle patricia trie's root node. Simply, it is a hash of the state-variable data related to an account's code content.

The contents of these four fields are used to define an account's type, and ultimately go on to define the extent of its functionalities. Thus, the two types of Ethereum accounts are:

a. **Externally-owned Accounts (EOAs)**- which are initialized as a cryptographic key pair:
   - A ***private key*** which is an encryptable and provably random 64-hex character, and its complementary counterpart;
   - A ***public key*** which is derived from the private key using the ECDSA (Elliptic Curve Digital Signature Algorithm).

   They have empty codeHash and storageHash fields and can only be controlled by anyone who possesses the private keys.

   The address of an EOA is obtained from its public key by prefixing “0x” to the last twenty characters of the account's public key's keccak-256 hash.

b. **Contract Accounts (CAs)**- which can only be created by a pre-existing EOA. They are initialized due to an EOA deploying executable code content on the EVM. This code content (stored as the codeHash) is enshrined in the EVM, and is responsible for controlling the account by defining its logic and interactions.
   Their transactions are entirely pull-based and predicated on their deployed code's logic. Since these accounts can only be controlled by their code content, they have no need for a private key and only have a public key. Thus, any agent who has the ability to update/change a contract account's code content would be able to access its balance.

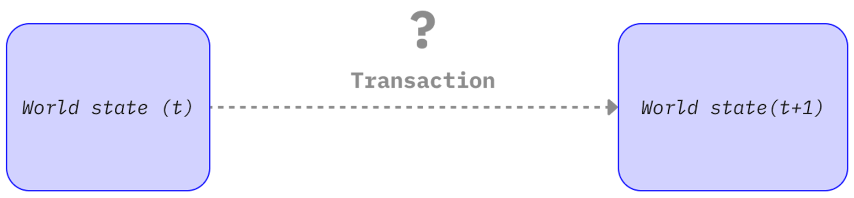
   The address of a CA is derived from its creator's address and its nonce up to the point of the contract's deployment.

Transactions

We recently described accounts as entities that possess the ability to send transactions across Ethereum. We can therefore understand that a primary purpose of an account is to send and receive transactions, while the blockchain acts as a ledger recording history of transactions as well as describing how transactions alter account fields based on rules described in the blockchain protocol specification.

So what are these “transactions”?

Transactions are operations sent from an account, which cause a change in the “state” of the network. They are cryptographically signed instructions from accounts, which result in a network-wide state update when executed.



Permissionlessness comes with the cost of perverse incentives, to deal with these, stringent guidelines (or validity rules) have to be defined for interactions in such environments.
In this context, transactions have to follow certain validity rules to be considered valid and executed. Most of these validity rules are implemented via constraints placed on the account sending the transaction, and vary based on what type of account it is.

Accounts and Transaction Validity

On Ethereum, EOAs are optimized for usability as they are end-user facing. They have the ability to send transactions *in a specific manner* and perfectly operate autonomously. They can also be created locally, the more common method being the use of wallet providers such as MetaMask, Rainbow, Rabby etc.
On the other hand, contract accounts can only send transactions permitted by their logic, **in response to being “*called*”**. They can only be created by an EOA which has a sufficient balance to pay for its network storage.

A more high-level description would be that EOAs can only hold a balance, while CAs can hold both a balance and logic that dictates how this balance can be spent.

These properties are due to the following logic parameters which define the rules an account's transactions must adhere to:

a. **Authentication Logic -** Used to define how an account proves its identity to the network while changing its balance and/or logic.
b. **Authorization Logic -** Used to define an account's access policy, i.e., who is able to access and make changes to the account's balance and/or logic.
c. **Nonce Logic** - which defines the order in which transactions from an account are to be executed.
d. **Gas Payment Logic** - Used to define the party responsible for settling a transaction's gas fee.
e. **Execution Logic -** Used to define what forms of transactions an account can send, or how a transaction is to be executed.

These parameters are designed to be rigid for EOAs thus:

- Authentication and authorization is provided by an ECDSA-based private key, i.e., a user who wishes to send a transaction from their EOA must use their private key to access the account and thus prove they have the right to carry out any changes to its balance.
- Nonce logic is a sequential counter scheme, which allows only one transaction per unique nonce to be executed sequentially per account.
- Gas payment logic specifies that the gas fee for transactions must be settled by the sender/originating account,
- Execution logic specifies that EOAs can only send the following transaction forms:
    a. Regular transfers between two EOAs.
    b. Contract deployment.
    c. CA *calls* which target a deployed CA's logic.

  More generally, the execution logic of EOAs constricts them to one transaction per valid signature.

On the other hand, CAs have more flexibility around these parameters:

- Authentication isn't necessary, as their transactions are consequential/pull-based in nature
- Authorization for CAs can take two forms:
    a. Ability to "*call*" the CAs content code (or execute its smart contract), which depends on the logic of the account's smart contract and its invariants.
    b. Ability to make changes to the CAs content code, which mostly depends on whether the content code is upgradeable or not.

       In most practical cases, the logic used is a multi-signature scheme which stipulates that an M of N valid signatures (where $M < N$) is required from specific accounts (commonly EOAs) in order for a change of the CA's logic to be valid.

- Transaction ordering is loosely nonce-based. The CA itself is able to send out as many transactions to as many diverse *callers* as possible, however each caller is limited based on their own capabilities.
- Gas payment is commonly handled by the *caller* of the CA's logic.
- The execution logic of CAs is more diverse to enable UX improvements such as muticall transactions and atomic transactions.

Evaluating these features, we observe that each type of account is designed to have a tradeoff between autonomy and programmability.

EOAs have full autonomy but limited programmability; they can authorize and send transactions whenever they want, but these transactions must follow a rigid format to be considered valid. CAs have full programmability (limited only by the EVM's design) but limited autonomy; their transactions do not have to follow any rigid format, but can only be sent out due to their logic being called first.

[EOAs have each logic strictly defined for them

CAs are able to define their execution logic via EVM-executable code

smart accounts are able to have each logic uniquely defined.]

In the following section, we will now study the implications of these design choices, and introduce our primary topic throughout this series– account abstraction.

How Account Abstraction Helps Ethereum

Now that we have a somewhat succinct knowledge of the different accounts' functionalities, we can easily identify their selling points as well as the issues they present to both user- and developer-experience on Ethereum.

As we previously mentioned, EOAs are designed as first-class accounts targeting end-users. Applications are designed to easily interact with them, there's almost no complexity to them, and of course there's no cost for creating one. However, its simplicity comes with a significant loss of novelty as they are designed to be strictly deterministic.

Some of the concerns around them are:

a. **Susceptibility to quantum attacks -** The ECDSA signature scheme used by their key pair is not quantum-resistant, and with an optimistic timeline of 5 - 10 years for industrial quantum systems being achieved, this poses a significant threat to Ethereum and its applications which heavily rely on the ECDSA scheme for cryptographic proofs and security.

b. **Lack of expression -** The rigid format of EOAs' validity rules eliminates the ability of users to express their transactions more succinctly via features such as transaction atomicity and batching, and transaction delegation.

c. **Self-sustainability** - Everyone's had their fair share of "i ran out of gas" moments in the middle of a transaction. This is due to the requirement that EOAs settle the gas for their transactions by themselves, which wouldn't be much of an ask if ether (ETH) wasn't the only acceptable gas currency. While this is a general issue with account-based state machines (and even UTXO-based ones), Ethereum always intended to be different.
Not everyone wants to (or would be able to) always hold ETH (i mean look at that price action), so the viable solutions would be to either allow multiple gas currencies (too hard, breaks too many invariants as described in the "Currency" section here), or to allow gas payments to be settled by another account that isn't the transaction's origin.

On the other end of the account spectrum, CAs target developers and a more technical user base. They serve as vehicles for smart contracts (i.e. we consider smart contracts to be their contained logic or code content) and so can implement novel transaction formats as enabled by the EVM.

However, for all these features they are glorified second-class accounts since they have no autonomy. Some of their drawbacks are as follows:

a. **Total lack of autonomy** - CAs cannot begin a transaction, they can only send out transactions in response to being invoked in a very particular manner.

b. **Susceptibility to human error in their logic** - The lack of rigidity often leads to incorrect definition of invariants and other such logic, which has led to billions of dollars in losses due to smart contract exploits and hacks. However, this is *almost* an entirely different topic which is beyond our scope here.

Having reviewed the design choices which led to the issues defined in this subsection, we can now proceed to evaluate the proposed solutions.

The concept of account abstraction (via smart accounts at least) has always been an integral part of Ethereum's roadmap. The lore is that the complexity surrounding its implementation threatened to further delay Ethereum's launch, and so it was scrapped for the current design with different accounts offering different functionalities. It was delayed again by Ethereum's focus on The Merge, and is now resurfacing as a principal part of the network's next major upgrade- Pectra. However, its complexity is still considered a significant drawback preventing its enshrinement, especially as Ethereum has pivoted to a rollup-centric roadmap.

The requirements are now two-fold:

a. Account standards have to be more expressive, but without loss of autonomy. A new standard that seals the divide between the EOA and CA standards.
b. The new standard has to bridge the gap between EOAs and CAs, while remaining thoroughly compatible across Ethereum and its L2 ecosystems.

Intuitively this concept plays a bigger role in the context of chain abstraction and interoperability, however our scope throughout this report is limited to the technical initiatives taken to achieve account abstraction itself.

Account abstraction aims to combine the best features of EOAs and CAs into a new account standard- ***smart accounts****, which allow full or partial separation of any account's validity rules into a validation logic and an execution one; so that accounts can define their own validity rules –as permitted by the EVM– just like contract accounts, while remaining fully autonomous just like externally-owned accounts*.

There is often confusion around the differences between both **smart accounts** and **smart contract wallets**; let us explicitly describe what these differences are below:

- **Smart accounts** are Ethereum accounts that are conceptualized to provide equal parts of programmability and autonomy. The idea is that both EOAs and CAs can become smart accounts simply by utilizing some mechanism (e.g. ERC 4337) that allows them to replace their network-imposed validity rules with bespoke validity rules, as they see fit.
- **Smart contract wallets** on the other hand, are simply wallets providers that serve as interfaces to contract accounts (yep, a wallet isn't an account).

The commercialisation of smart contract wallets eased the adoption of CAs by a wider market, allowing less technical users to take advantage of the features they offer. However, they still face the pitfalls associated with CAs.

Back to the conversation; we had previously discussed the parameters which are used to define the validity rules of transactions of accounts:

- Authentication
- Authorization
- Nonce logic
- Gas payment logic
- Execution logic

The values of the first four parameters may collectively be referred to as an account's **validation logic**, which are checks that occur **before** a transaction's execution begins.
The last parameter defines how the execution of the transaction is to proceed.

In the introduction, we provided a high-level overview of the current AA landscape in the form of a classification of sorts for the various proposed designs. We will now focus on the first class of solutions to Ethereum's account dilemma- EOA programmability.

Programmable EOAs

Ethereum's biggest appeal is its young but vibrant DeFi ecosystem which features a variety of decentralized applications that are its primary liquidity sinks. Most of these DApps are optimized to serve EOAs, thus they are hard to interface with CAs, and eventually smart accounts. While smart contract wallets do help CAs in this case, they come with their own limitations and an entirely different UX.

An interim solution being explored while both DApps and wallet providers get used to the smart account standard, is to provide temporary enhancements to EOAs that enable them to overcome most of their imposed restrictions, be it their validation or execution logic.

Below, we go over the specifications of three major EIPs which provide actionable routes to EOA programmability; from the less known EIP 5806, to the ambitious EIP 3074, and then finally to the record-breaking EIP 7702.

Programmability via EIP-5806

This proposal seeks to bring more functionality to the EOA standard by allowing it to perform delegate calls to a contract account's logic (its smart contract). This effectively causes the smart contract to be executed in the context of the caller EOA, i.e., the EOA remains in control of its validation logic, while its execution logic is handled by the corresponding CA's logic.

Before we proceed any further, let us take a trip down the Ethereum evolution memory lane to EIP-7.

EIP-7 proposed the creation of the 0xf4/DELEGATECALL opcode, which is used to send message calls into a primary account with a secondary account's logic, while maintaining the values of the primary account's [sender] and [value] fields.
In other words, the primary account "inherits" (or borrows if you prefer) the logic of the secondary account for some duration as specified in the message call, so that the latter's logic is executed in the context of the former.

This opcode allowed DApp developers to split their application's logic into multiple smart contracts while maintaining interdependence, so that they could easily skirt around code size barriers and gas barriers.

EIP-5806 summarized

Okay, so what delegate calls allow CAs to be interdependent? EIP-5806 uses EIP-7 as an inspiration to propose the expansion of the delegate call functionality to EOAs as well, i.e., let us allow EOAs to also be interdependent with CAs because why not.

Specifications
EIP 5806 introduces a new EIP-2718-compliant transaction type which is packed as follows:

`rlp([chainID, nonce, max_priority_fee_per_gas, max_fee_per_gas, gas_limit, destination, data, access_list, signature_y_parity, signature_r, signature_s])`.

These transactions are designed so that the [to] field – which represents the recipient's address – can only accept addresses as 20-byte inputs, disabling the sender from using the CREATE opcode.

The motivation behind each component of the RLP scheme are as follows:

- **chainID**: The current chain's EIP-115-compliant identifier padded to 32 bytes. This value provides replay attack protection, so that transactions on the original chain aren't trivially replicated on alternate EVM chains with a similar history and less economic security.
- **nonce**: A unique identifier for each transaction which also provides replay attack protection.
- **max_priority_fee_per_gas** and **max_fee_per_gas**: The values of the gas fee that an EIP-5806 transaction is to pay for ordering and inclusion respectively.
- **gas_limit**: The maximum amount of gas a single 5806-type transaction can consume.
- **destination**: The transaction recipient
- **data**: The executable code content
- **access_list**: Agents who are conditionally authorized to execute EIP-5806 transactions.
- **signature_y_parity**, **signature_r**, and **signature_s**: three values that together represent a secp256k1 signature over the message - `keccak256 (TX_TYPE || rlp ([chainID, nonce, max_priority_fee_per_gas, max_fee_per_gas, gas_limit, destination, data, access_list]))`.

While the packing of EIP-5806 transactions in EIP-2718 envelopes allows them to be greatly backwards compatible, EOAs aren't equivalent to CAs. So certain restrictions must be defined in the way an EOA utilizes delegate calls to prevent invariant breakage.

These restrictions are targeted at the following opcodes:

- SSTORE/0x55: This opcode allows an account to save a value to storage. It is restricted in 5806 transactions to prevent EOAs from setting/accessing storage using delegate calls, thus preventing potential issues that may arise in the future due to account migration.
- CREATE/0xF0, CREATE2/0xF5, and SELFDESTRUCT/0xFF: Access to these are restricted to prevent alteration of an EOA's nonce in a different execution frame (contract creation/destruction in this case) while it is performing an EIP-5806 transaction.

Potential Applicability/Use Cases
The primary applicability of EIP 5806 is execution abstraction for EOAs. Allowing EOAs to trustlessly interact with smart contracts beyond simple calls to their logic grants them features such as:

- Conditional execution of transactions
- Transaction batching
- Multicall transactions (e.g. approve and call)

Criticisms
The changes proposed by EIP-5806,while enabling a lot of features, aren't particularly novel; its existence is mostly predicated on an already functional EIP-7. This allows it to bypass many potential hurdles to acceptance.

One of the major concerns voiced in its early days was the potential risk of allowing EOAs to access storage and change it, just like CAs currently do. This breaks a lot of network-enshrined invariants concerning how EOAs are to transact, and so was dealt with by introducing the restrictions mentioned in the previous subsection.

A second criticism (which is a bit of a double-edged sword) is the simplicity of EIP-5806; there's some sentiment that the rewards due to accepting EIP-5806 might not be worth the cost, since it only enables execution abstraction and not much else. Every other validity restriction remains network-defined for EOAs which opt in to EIP-5806, in contrast to other somewhat similar EIPs which we discuss in the following sections.

Programmability via EIP-3074

The EIP-3074 initiative is currently the most discussed proposal for introducing programmability to EOAs, and enabling them to behave a bit more like smart accounts.
It proposes to allow EOAs to delegate most of their validation logic to specialized contract accounts, referred to as invokers, by superimposing the latter's authorization logic over theirs for specific forms of transactions.

This EIP proposes the addition of two new opcodes to the EVM:

- [AUTH] which sets a context-variable [authorized] account to act on behalf of a second [authority] account, based on the latter's ECDSA signature.
- [AUTHCALL] which sends/implements calls for the [authority] account from/as the [authorized] account.

These two opcodes allow an EOA to delegate control to a pre-established CA, and thus, act as one through it, without having to deploy a contract and incur the costs and externalities associated with that.

Specifications
EIP-3074 allows transactions to use a [MAGIC] signing format to prevent collisions with other transaction signing formats. The active account to which [AUTHCALL] instructions are passed is implemented as a context-variable field named [authorized], which only persists through a single transaction and must be redefined for every new [AUTHCALL].

Before addressing the complexities of each opcode, these are the entities involved in a EIP-3074 transaction:

- [**authority**]: The primary signing account (an EOA) that delegates access/control to a second account, which is typically a contract account.
- [**authorized**]: The account to which [AUTHCALL] instructions are to be passed for execution. In other words, it is the account in which the logic of an [AUTHCALL] is executed, on behalf of the [authority], using constraints defined by an [invoker].
- [**invoker**]: A subsidiary contract meant to manage the interactions between the [authorized] account and the logic of the [AUTHCALL], especially in cases where the primary logic of the latter's contract code is gas sponsorship.

  Invoker contracts receive [AUTH] messages with a [COMMIT] value from [authority]; this value defines the restrictions the account wishes to place on [authorized]'s execution of [AUTHCALL] instructions.
  Thus, invokers are responsible for ensuring that the [contract_code] defined in an [authorized] account is not malicious and has the ability to satisfy the invariants placed by the primary signing account in a [COMMIT] value.

The [AUTH] opcode has three stack element inputs; or more simply – it is defined by three inputs which compute a single output. These inputs are:

a. **authority**: which is the address of the EOA which generates the signature
b. offset
c. length

The last two inputs are used to describe a range of modifiable memory from 0 to 97, where:

a. [memory(offset : offset+1)] – [yParity]
b. [memory(offset+1 : offset+33] – [r]
c. [memory(offset+33 : offset+65)] – [s]
d. [memory(offset+65 : offset+97)] – [COMMIT]

The variables [yParity], [r] and [s] are collectively interpreted as an ECDSA signature, [magic], on the secp256k1 curve over the message:

```
[keccak256 (MAGIC || chainID || nonce || invoker address || COMMIT)]
```

where:

- [**MAGIC**] is an ECDSA signature resulting from the combination of the variables:
    - [**chainID**] which is the current chain's EIP 115-compliant identifier used to provide replay attack protection on alternate EVM chains with a similar history and less economic security.
    - [**nonce**] which is the transaction signer's address' current nonce, left-padded to 32 bytes.
    - [**invokerAddress**] which is the address of the contract which contains the logic for [AUTH]'s execution.
- [**COMMIT**] is a 32-byte value used to specify additional transaction validity conditions in the invoker's pre-processing logic.

If the computed signature is valid and the signer's address equal to [authority], the [authorized] field is updated to the value provided by [authority]. If any of these requirements aren't satisfied, the [authorized] field remains unchanged in its previous state, or as an unset value.

The gas cost for this opcode is computed as the sum of:

a. A fixed fee for the [ecrecover] precompile and extra for a keccak256 hash and some additional logic, valued at 3100 units
b. A memory expansion fee which is calculated similarly to the [RETURN] opcode, and applied when memory is expanded past the specified range of the current allocation (97 units)
c. A fixed cost of 100 units incurred for a warm [authority] and 2600 units for a cold one to prevent attacks due to mispricing of state-accessing opcodes.

[AUTH] is implemented to not modify memory, and takes [authority]'s value as an argument so that it is trivial to verify its value from the provided signature.

The [AUTHCALL] opcode has seven stack element inputs which are used to compute a single stack element output.
It has the same logic as the [CALL] opcode, i.e.; it is used to send message-calls into an account and trigger specific logic in its contracts. The only deviation in their logic is that [AUTHCALL] is designed to set the [CALLER]'s value before proceeding with execution.

Thus, [AUTHCALL] is used by the [authority] to trigger context-specific behavior in [authorized] with logical checks proceeding as follows:

a. Check for [authorized]'s value. If unset, the execution is deemed invalid and the frame is immediately exited. This helps prevent unfair charges due to unprecedented failures.
b. Checks for the gas cost of [authorized]'s intended behavior.
c. Checks for the [gas] operand's EIP 150 compliant-value.

d. Checks for the availability of the total gas cost –[value]– in [authority]'s balance.
e. Execution occurs after deduction of [value] from the balance of [authority]'s account. If [value] is greater than their balance, the execution is invalidated.

The gas cost for [AUTHCALL] is computed as the sum of:

- A fixed cost for calling [warm_storage_read]
- A memory expansion cost [memory_expansion_fee]; which is calculated similarly to the gas cost for the [CALL] opcode
- A dynamic cost [dynamic_gas]
- The execution cost of the subcall [subcall_gas]

The data returned from an [AUTHCALL] is accessed through:

- [RETURNDATASIZE] – which pushes the size of the return data buffer onto the output stack
- [RETURNDATACOPY] - which copies the data from the return data buffer to memory.

To bring it all together with much less of the tech-speak; Ethereum transactions typically specify two values:

1. **tx.origin** - which provides authorization for the transaction.
2. **msg.sender** - in which the transaction actually occurs.

In EOAs, as previously mentioned, authorization is tightly coupled with execution, i.e.; (tx.origin == msg.sender). This simple invariant is the biggest deterrent to most of the issues we explained in the "Accounts and Transaction Validity" subsection of this report.

[AUTH] messages from [authority] allows it to offset the tx.origin function to [authorized], while remaining the msg.sender. It also allows it to define restrictions to this privilege using a [COMMIT] value.
[AUTHCALL] then allows [authorized] to access a contract's logic, using an [invoker] as an intermediary to ensure the contract it wishes to access is harmless. That is, for every [AUTHCALL], [authorized] is to specify a particular [invoker] for their [COMMIT].

<u>Potential Applicability/Use Cases</u>

EIP 3074 is primarily responsible for allowing EOAs to delegate their authorization logic to a different account, however its open design enables a lot more in different contexts.
The entire validation logic of an EOA can be abstracted by applying various constrictions/innovations to the invoker as necessary, some of the possible designs based on their target logic include:

- **Nonce logic**: EIP-3074 allows the EOAs nonce to remain untouched after sending an [AUTH] message, meanwhile its nonce for every [AUTHCALL] depends on what invoker it is interacting with. In this way, it enables nonce parallelism for EOAs, so that they can send out multiple non-overlapping [AUTHCALL]s as they wish to.

- **Gas payment logic**: As specified in the EIP, invokers can be designed to allow gas sponsorship. As such, the gas fees for a user's [COMMIT] could be deducted from the transaction's origin, or from any supportive account, whether a personal one or a service-based relay (gas sponsorship services).

Also, execution logic is intuitively abstracted; after all the invoker (which is a CA) is now responsible for sending transaction requests behalf of the EOA.

Criticisms

- **Invoker Centralisation**
  - Quoting one of its authors: "*I would not expect wallets to expose functionality to sign over arbitrary invokers …*".
  - Perhaps the biggest problem poised by the 3074 initiative is that innovation atop it will very easily tend to permissioned and proprietary deal flows; just like the current iterations of Ethereum's MEV and PBS markets. By default, invoker contracts need to be greatly audited in order to prevent even worse attacks than are currently possible. This will inevitably tend to an ecosystem where only a handful of invoker contracts developed by influential figures will be adopted as the default for wallet developers. Thus, it boils down to a tradeoff between taking the hard decentralized pathway of constantly auditing and supporting invoker contracts at the risk of users' security; and simply adopting invoker contracts from established and reputable sources with better guarantees for user security and less oversight on the contract's safety.
  - Another aspect of this point is the cost function associated with designing, auditing, and marketing a functional and safe invoker. Smaller teams will almost always be outdone by bigger organizations –especially on the marketing front– which already have some established reputation, even if their product is better.

- **Forward-Compatibility Issues**
  - EIP-3074 entrenches the ECDSA signature scheme, since it is still considered more valid than the authorization scheme introduced via the invoker. While there are arguments that quantum-resistance isn't currently a definitive problem, and that there's much worse at stake in a future where ECDSA is corruptible; Ethereum's somewhat unstated aim is to *always* be ahead of such problems. Potentially sacrificing quantum- and censorship-resistance for marginal improvements in UX might not be the best of choices in the near future.
  - Another point on the forward-compatibility argument is that while the benefits of 3074 were still being assessed, ERC-4337 (which doesn't require any protocol changes) now has a great market, so you have to be compatible with it too to avoid compartmentalisation of ecosystems.

    - Even with the native account abstraction roadmap, the [AUTH] and [AUTHCALL] opcodes would eventually become obsolete in the EVM, introducing a great deal of technical debt to Ethereum in order to deliver a marginal amount of UX improvement.

- **ECDSA Scheme Irrevocability**
    - After sending an [AUTH] message and delegating control, the EOA is expected to avoid the usual private key authorization scheme, as sending out a "normal" transaction causes the authorization it delegated to every invoker to be revoked.
    - The ECDSA scheme remains strictly superior to any other scheme which the associated contracts might use, meaning that a loss of private keys would result in a total loss of the account's assets.

Programmability via EIP-7702

This proposal had initially set out as a somewhat minimalistic variation of EIP 3074, and was even meant to be an update to it. It was birthed to address the supposed inefficiencies of EIP 3074, especially the concerns around its incompatibility with the already flourishing 4337 ecosystem and the native account abstraction proposal– RIP 7560.

Its approach is the addition of a new EIP 2718-compliant transaction type –[SET_CODE_TX_TYPE]– which allows an EOA to behave as a smart account for specified transactions.

This design enables the same features as EIP 5806 and some more, while remaining compatible with the native account abstraction roadmap and existing initiatives.

Specifications
EIP-7702 allows an EOA to "import" the code content of a contract through a [SET_CODE_TX_TYPE] 2718-compliant transaction of the format:

```
rlp([chain_id, nonce, max_priority_fee_per_gas, max_fee_per_gas,
gas_limit, destination, value, data, access_list, authorization_list,
signature_y_parity, signature_r, signature_s])
```

This payload is entirely similar to that of EIP 5806, except that it introduces an "authorization list". This list is an ordered sequence of values of format:

```
[[chain_id, address, nonce, y_parity, r, s], ...]
```
where each tuple defines the [address]' value.

Before proceeding, the parties involved in a SET_CODE_TX_TYPE are:

- [**authority**]: which is the EOA/primary signing account
- [**address**]: which is the address of an account containing delegatable code.

When [authority] signs a SET_CODE_TX_TYPE specifying [address], a delegation designator is created. This is a “pointer programme” which causes all code retrieval requests due to the [authority]’s actions at any instant to be channeled to [address]’ observable code.

For each `[chain_id, address, nonce, y_parity, r, s]` tuple, the logic flow of a 7702-type transaction is as follows:

a. Verification of [authority]’s signature from the provided hash using: `authority = ecrecover(keccak(MAGIC || rlp([chain_id, address, nonce])), y_parity, r, s]`
b. Prevention of cross-chain replay attacks and other attack vectors by verification of the chain's ID.
c. Checking whether [authority] already has code content.
d. Nonce check to ensure [authority]’s nonce is equal to the nonce included in the tuple.
e. If the transaction is [authority]’s first SET_CODE_TX_TYPE transaction, it is charged a PER_EMPTY_ACCOUNT_COST fee. In the case that its balance is less than the value of this fee, the operation is abandoned.
f. Delegation designation occurs, wherein the code of [authority] is set to the a pointer of the [address].
g. The nonce of the signer –[authority]– is increased by one.
h. [authority] is added to a list- accessed addresses, which (oversimplified) is a set of addresses which are made such that the reversion of a scope of a transaction from them causes them (the address) to be set back to their previous state, before the reverted scope was entered. This is as defined in EIP-2929 to enable caching of reusable values and prevent unnecessary charges.

Phew! To tie it all back; this EIP allows EOAs to send transactions which set a pointer to a contract's code, allowing them to implement this logic as their own in subsequent transactions. In this way it is strictly stronger than EIP 5806, because it allows EOAs to actually have code content (as opposed to EIP 5806 which simply allows EOAs to send delegate-calls).

Potential Applicability/Use cases

- **Execution Abstraction**
    - While it could be argued that it isn't an abstraction anymore since the EOA actively takes in the logic it wishes to execute, it still isn't the “primary owner” of said logic. Also, it doesn't **directly** contain logic, it simply specifies a pointer to the logic (in order to reduce computational complexity). So we're going with execution abstraction!

- **Gas Sponsorship**

        - While the require(msg.sender == tx.origin) invariant is broken to allow self-sponsorship, the EIP still allows the integrations of sponsored transaction relayers. However, the caveat is that such relayers need a reputation- or stake-based system in order to prevent griefing attacks.

- **Conditional Access Policies**

EOAs can point only to a specific portion of the account's code, so that only the logic of that portion is executable in their context.

- **Cross-chain Smart Contract Deployment**
    - Due to its non-restrictive nature, a EIP-7702 transaction could allow a user to access the CREATE2 opcode and use it to deploy bytecode to an address without any other restrictive parameters such as fee market logic (e.g. EIP-1559 and EIP-4844). This allows the address to be recovered and used across multiple state machines, with the same bytecode, where its account on each chain is then responsible for defining the other context-variable parameters.

<u>**Criticisms**</u>

- **Lack of backwards-compatibility**
    - As EIP-7702 is still very recent, there hasn't been a lot of prototyping and testing for its dependencies and potential disadvantages, but its minimalistic model guarantees it a lot of flexibility, and thus utility, in different contexts. However it breaks way too many invariants and isn't backwards compatible.
    - Some of its logic includes:
        1. <u>Mid-transaction EOA nonce alteration</u>: EIP-7702 doesn't limit any opcodes in a bid to ensure consistency. This means an EOA can implement opcodes such as CREATE, CREATE2 and SSTORE while executing a EIP-7702 transaction, allowing its nonce to be increased.
        2. <u>Allowing accounts with a non-zero codeHash value to be transaction originators:</u> EIP-3607 was implemented to decrease the potential fallout of an “address collision” between EOAs and CAs. An address collision occurs when the 160-bit value of an EOA’s address is wholly equivalent to that of a CA's address. Most users aren't savvy to the actual contents of an account (or even the difference between an account and an address!), allowing address collisions means that an EOA could masquerade as a CA and attract user funds in a long-winded bit to eventually steal it all. EIP-3607 addressed this by stipulating that accounts which contain code should not be able to spend their balance using their own authorization logic. However, EIP 7702 breaks this invariant in order to enable EOAs to remain autonomous even after gaining some programmability.

- **Resemblance to EIP-3074**
    - Signing over an account's address instead of its code content is basically just like 3074's scheme with invokers. It doesn't provide strict guarantees around

cross-chain code content consistency, since an address could take on a different code content on different chains. This means that an address whose code content contains the same logic on one chain could be predatory or outright malicious on another chain, and this could lead to loss of user funds.

Allowing EOAs to execute code in any manner comes with pitfalls and potential blindsides; but it comes with unchallenged benefits to UX, if done well.
Ethereum's culture of open discussion makes it a great testing ground for such innovations since almost every implication of every design is thoroughly deconstructed by subject experts.

EIP-7702 is currently the poster child for mechanisms that seek to bring EVM programmability to EOAs, having been marked as a replacement for EIP 3074's slot in the Pectra upgrade. It inherits the open design of 3074's mechanism while greatly restricting what the logic an EOA chooses to execute can do or not. It also enables a lot more by avoiding 3074's restrictions to certain classes of opcodes.

While there's still some trivial amount of work to be done on the proposal's design, it has already garnered a lot of goodwill and support from developers, especially since it directly has Vitalik backing it.

While there are claims that this approach to account abstraction might be even better than smart accounts, since it requires less changes and isn't as complex, and EOAs are already enshrined; we must not forget the ultimate goal of **quantum resistance** at every level of the Ethereum network, which is currently infeasible at its very core due to its utilization of ECDSA-based signature schemes for EOA authorization.

Thus, EOA programmability is to be seen as a stop in the path and not the destination. It supercharges EOAs and enables better UX while remaining compatible with the ultimate account abstraction goal of smart accounts.

In our next report, we will be diving into EOA migration schemes to see how well they fit on the account abstraction roadmap, stay tuned!