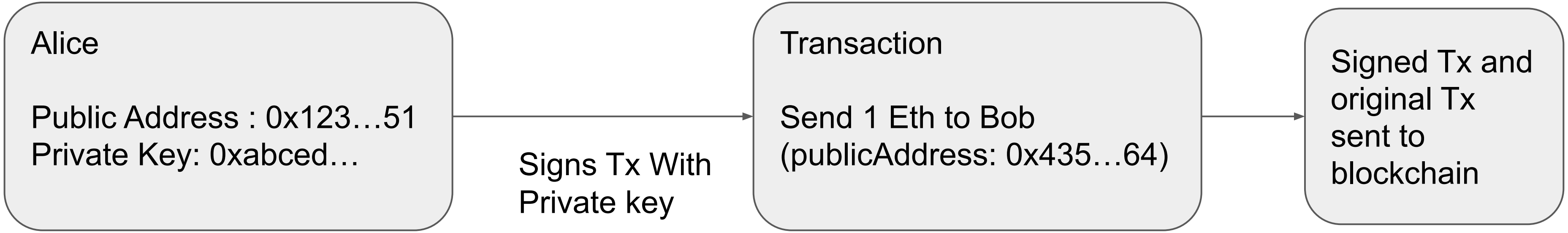
Account Abstraction

Public-Private key pair

- Public Key is generated by the private key
- Message encrypted by one key can be decrypted using the other
- Public and Private keys are very long so we usually use the hashed version of the public key in blockchain which is generally called public address
- Public key can be derived from the private key but not vice versa

Simplified flow of a TX in blockchain



Alice

Public Address : 0x123…51
Private Key: 0xabced…

Signs Tx With
Private key

Transaction

Send 1 Eth to Bob
(publicAddress: 0x435…64)

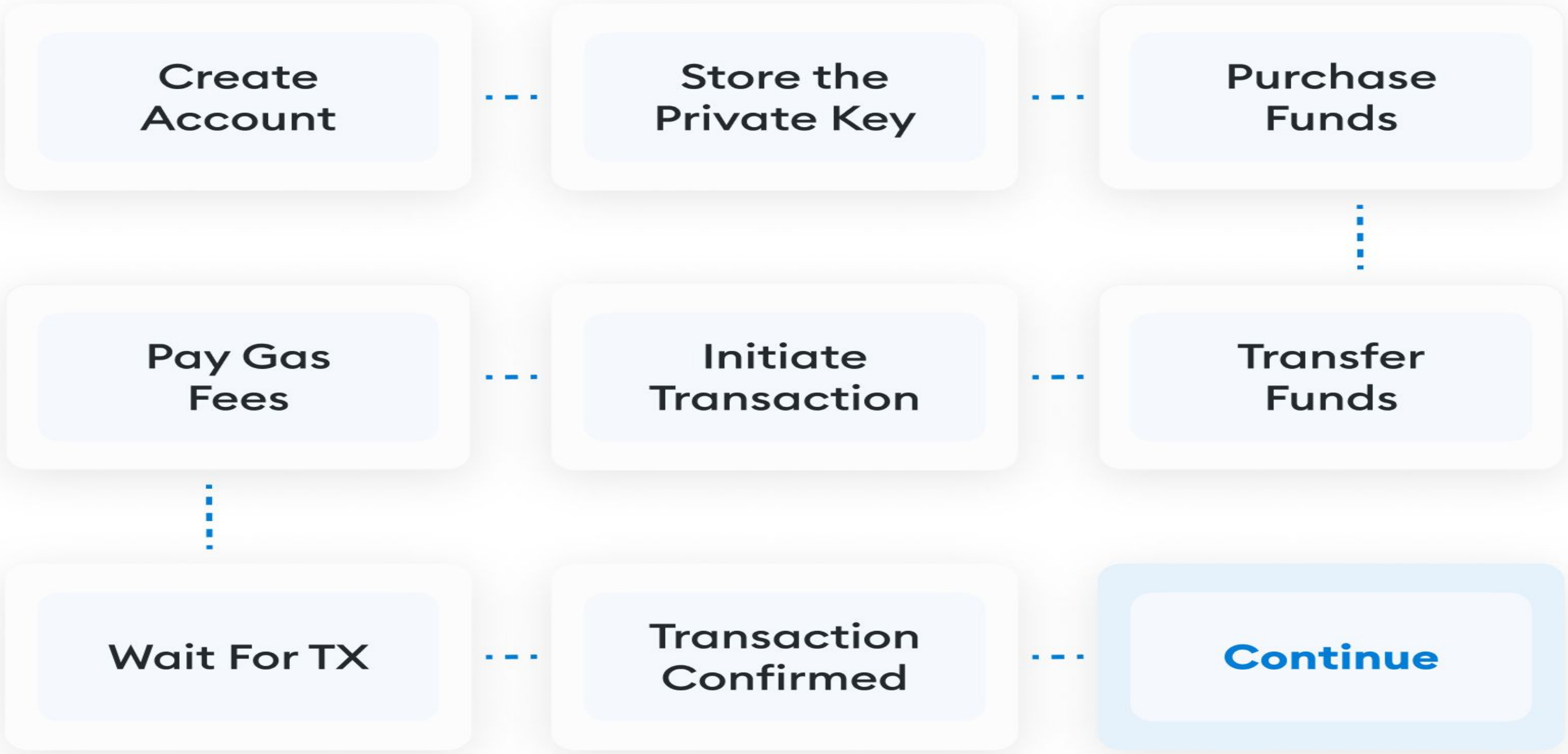
Signed Tx and
original Tx
sent to
blockchain

Blockchain Ledger

1) 0x123…51	(10-1) ETH
2) 0x435…64	(2+1) ETH

If we have the original message or transaction and the signed transaction (by private key), the public address can be retrieved through a mathematical equation

Workflow of creating an EOA and interacting with dapp



Create Account

Store the Private Key

Purchase Funds

Transfer Funds

Initiate Transaction

Pay Gas Fees

Wait For TX

Transaction Confirmed

Continue

Security-related drawbacks of EOAs

1. Cannot recover wallet access without seed phrase

2. Private keys create single points of failure and impose more burden on users for security

3. Considerable risk of losing assets (tokens/NFTs) to malicious actors via phishing, social engineering, hacks, etc.

METAMASK

Intro

Account Abstraction is a blockchain technology that allows users to use smart contracts as their accounts.

The default account for most users is an Externally Owned Account, or EOA. EOAs are accounts controlled by an external private key.On most blockchain networks using the Ethereum Virtual Machine (EVM), only an EOA can trigger transactions so this is the default account model for most users.

Unfortunately EOAs require users to know a lot about how a blockchain works to use them safely. We can create much better user experiences using Contract Accounts.

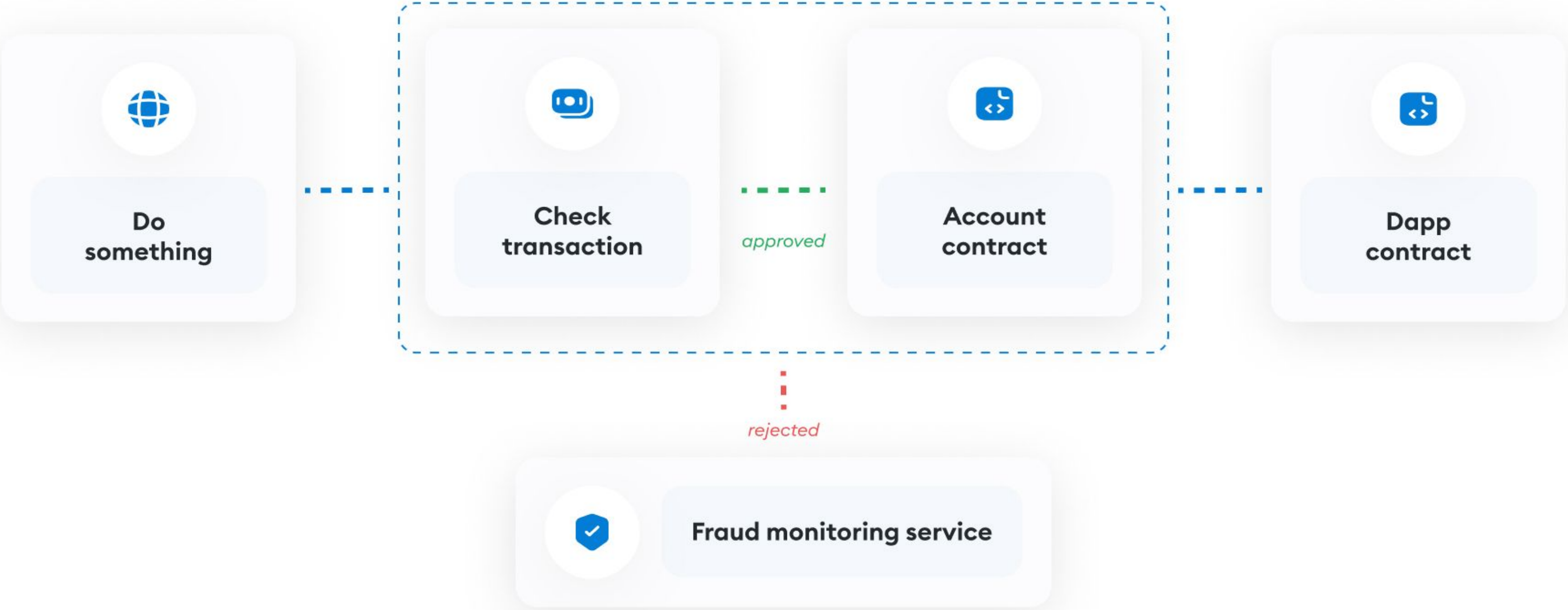
Here are some of the things contract accounts can enable that aren't possible with EOAs:

- 🔑 **Arbitrary verification logic:** Support single and multi sig verification and any arbitrary signature scheme.
- 💱 **Sponsored transactions:** Allow users to pay transaction fees in ERC-20 tokens or build your own fee logic, including sponsoring transaction fees on your app.
- 🔒 **Account security:** Enable social recovery and security features like time-locks and withdraw limits.
- ⚛️ **Atomic multi-operations:** Build flows that better align with your user's intent such as trading in one click rather than approving and swapping separately.

Account abstraction sounds great! But there are some down sides to also consider:

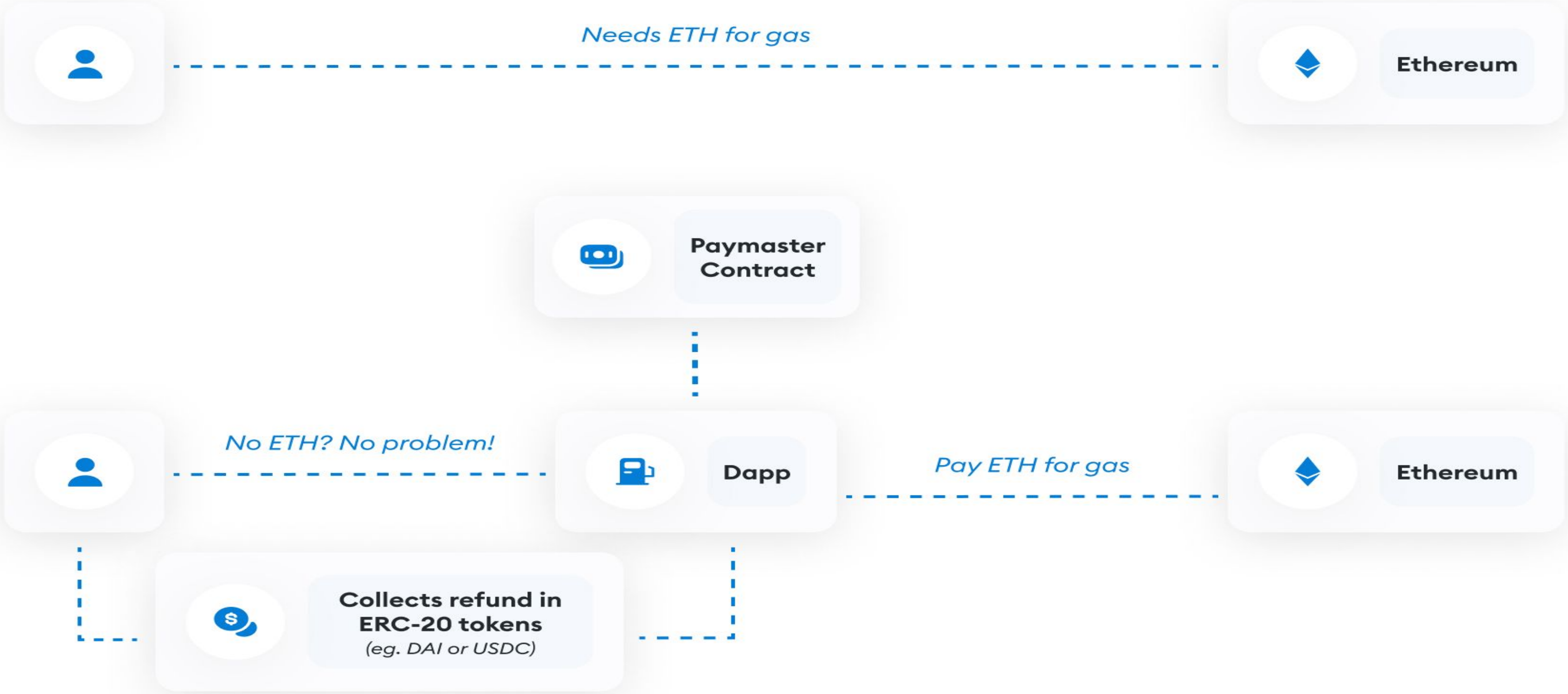
- ✍️ **Signing issues:** Ideally, all apps would following EIP-1271 to validate signatures. Unfortunately this is not always the case and those apps would be incompatible with contract accounts.
- ⛽ **Higher gas l:** On L2s and other scaling solutions this becomes less of a problem. However more research on how to reduce gas cost in this context, especially on Ethereum mainnet, is required.

Multi-factor authentication for web3 wallets



METAMASK

Normal transactions vs ETH-less transactions



Needs ETH for gas

Ethereum

Paymaster Contract

No ETH? No problem!

Dapp

Pay ETH for gas

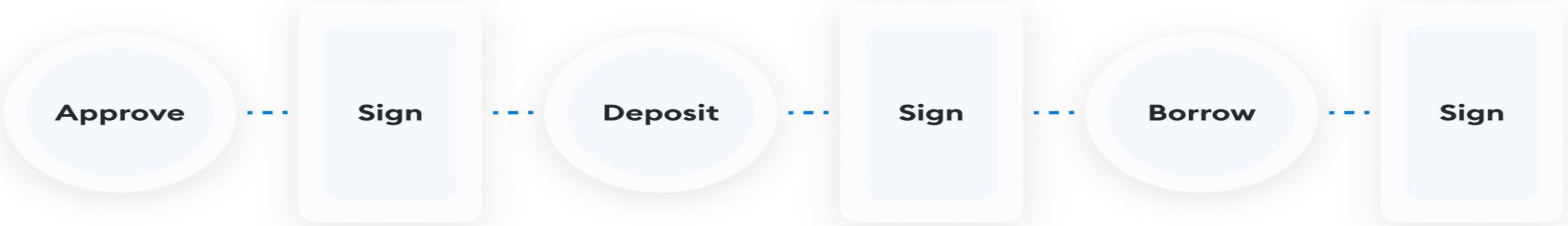
Ethereum

Collects refund in ERC-20 tokens *(eg. DAI or USDC)*

METAMASK

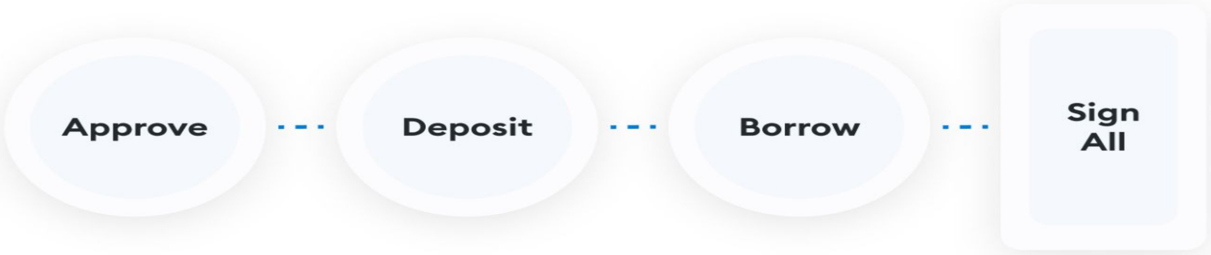
How transaction batching improves the workflow of using a DeFi dapp

DeFi dapps today



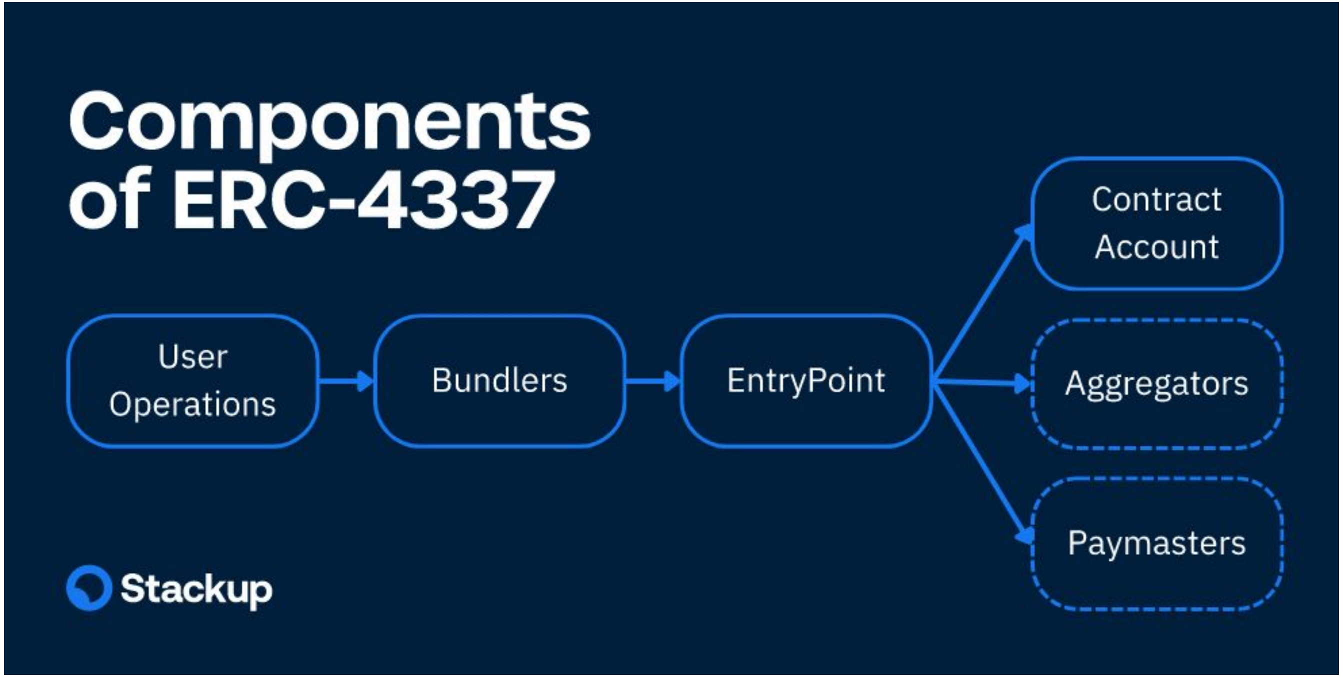
Approve - - - Sign - - - Deposit - - - Sign - - - Borrow - - - Sign

DeFi dapps tomorrow (with account abstraction)



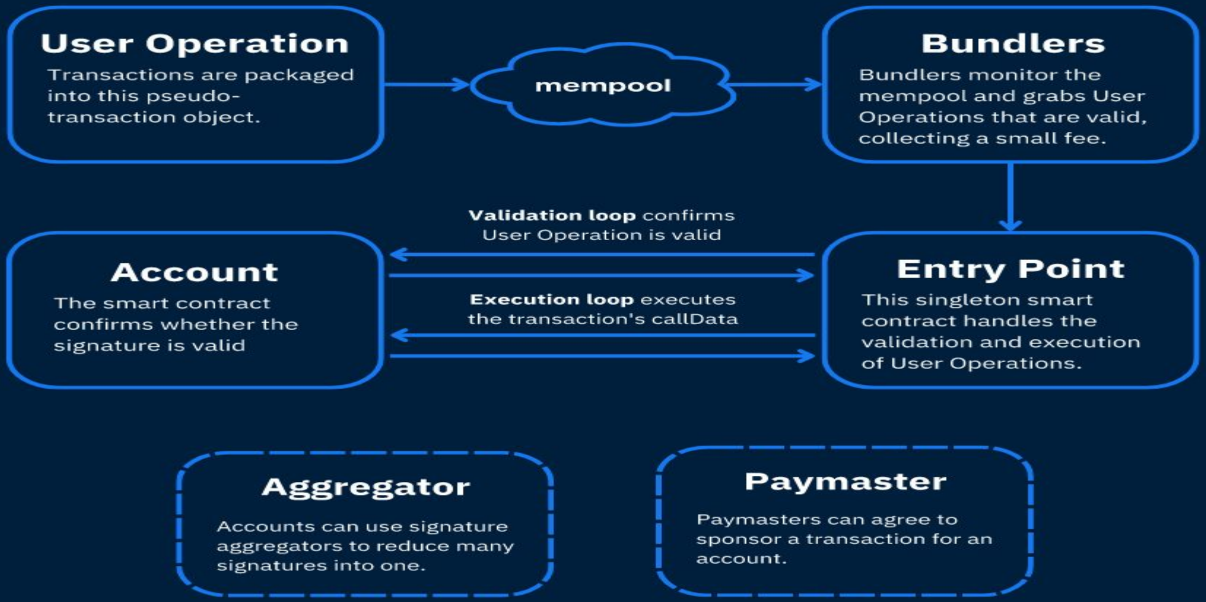
Approve - - - Deposit - - - Borrow - - - Sign All

METAMASK



- *UserOperations* are pseudo-transaction objects that are used to execute transactions with contract accounts. These are created by your app.
- *Bundlers* are actors that package UserOperations from a mempool and send them to the EntryPoint contract on the blockchain.
- *EntryPoint* is a singleton smart contract that handles the verification and execution logic for transactions.
- *Contract Accounts* are smart contract accounts owned by a user.
- *Paymasters* are optional smart contract accounts that can sponsor transactions for Contract Accounts.
- *Aggregators* are optional smart contracts that can validate signatures for multiple Contract Accounts.

How ERC-4337 works



User Operation

Transactions are packaged into this pseudo-transaction object.

mempool

Bundlers

Bundlers monitor the mempool and grabs User Operations that are valid, collecting a small fee.

Validation loop confirms User Operation is valid

Account

The smart contract confirms whether the signature is valid

Execution loop executes the transaction's callData

Entry Point

This singleton smart contract handles the validation and execution of User Operations.

Aggregator

Accounts can use signature aggregators to reduce many signatures into one.

Paymaster

Paymasters can agree to sponsor a transaction for an account.

Stackup

Existing Libraries and frameworks

- Zerodev
- Stackup
- Biconomy
- UserOp.js

Zerodev

ZeroDev is a developer framework for creating, using, and extending smart wallets powered by account abstraction (ERC-4337).

- Create: ZeroDev integrates with all popular onboarding and authentication solutions, so that you can easily create *smart wallets* for your users, no matter they are Web2 or Web3.
- Use: ZeroDev provides SDKs and APIs for the most popular smart wallet *features*, including gas sponsoring, transaction batching, automated transactions, session keys, and more.
- Extend: if our smart wallet features don't cover your needs, you can easily build custom smart wallet *plugins* using our plugin framework.

No matter if you are building a wallet or a DApp, you can use ZeroDev to dramatically improve Web3 UX for your users.

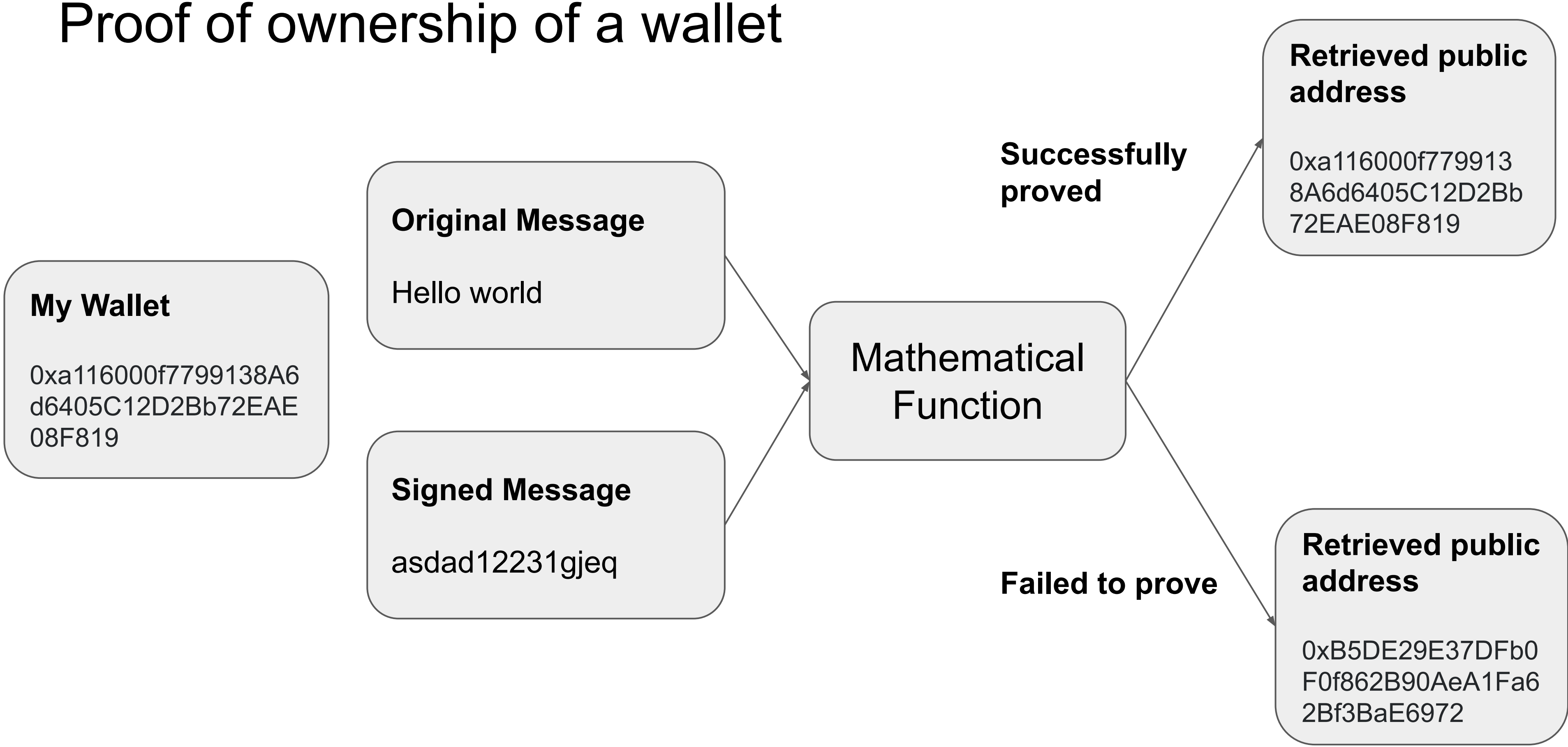
APIs

1. Get-user-nonce -> return nonce if exists from an existent user
2. Signup -> if nonce doesn’t exists in above case then create a user in database and get a nonce.
3. Login -> send signature and get a token

What does authentication requires?

1) Unique Identifier
2) Proof of ownership of that unique identifier

Proof of ownership of a wallet



My Wallet

0xa116000f7799138A6
d6405C12D2Bb72EAE
08F819

Original Message

Hello world

Signed Message

asdad12231gjeq

Mathematical
Function

Successfully proved

Retrieved public address

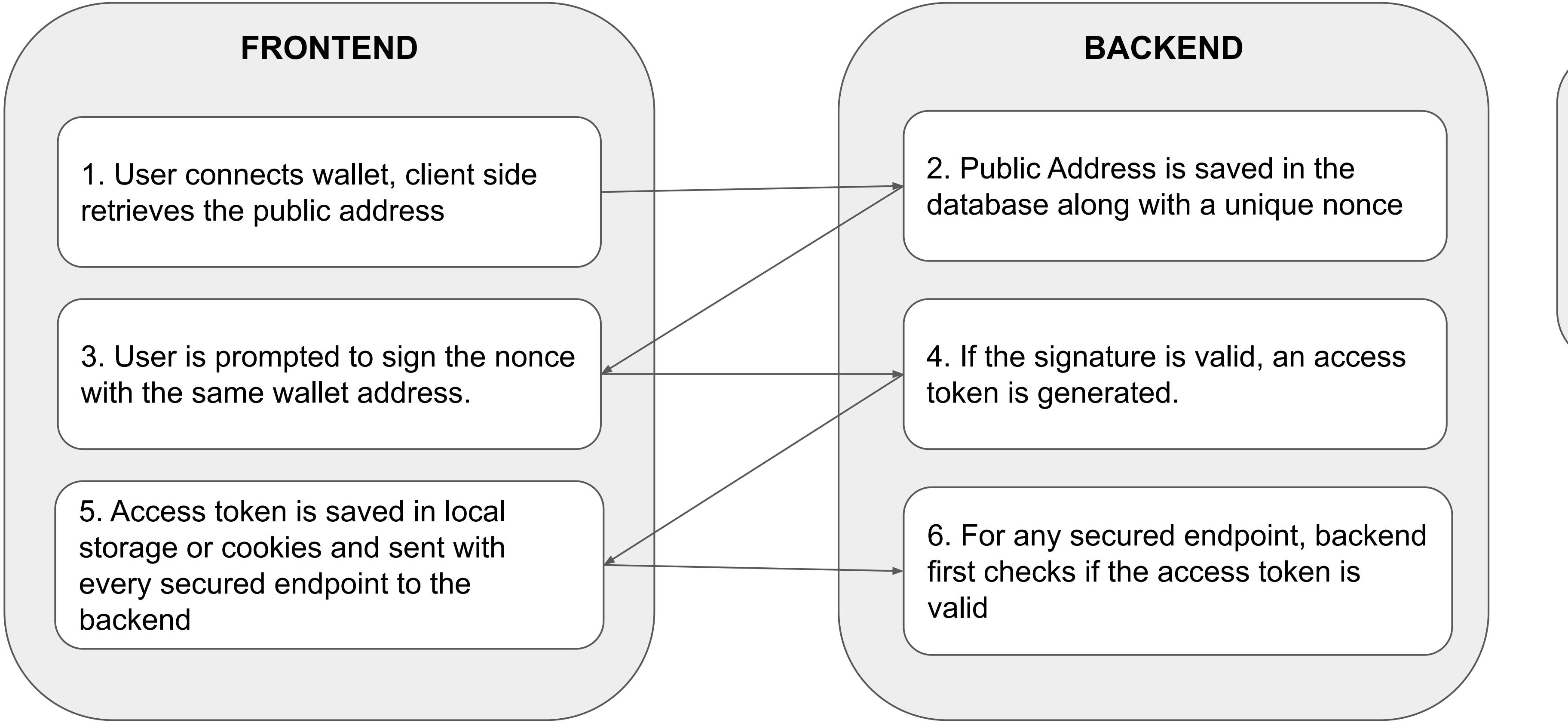
0xa116000f779913
8A6d6405C12D2Bb
72EAE08F819

Failed to prove

Retrieved public address

0xB5DE29E37DFb0
F0f862B90AeA1Fa6
2Bf3BaE6972

Authentication flow with wallet



FRONTEND

1. User connects wallet, client side retrieves the public address

3. User is prompted to sign the nonce with the same wallet address.

5. Access token is saved in local storage or cookies and sent with every secured endpoint to the backend

BACKEND

2. Public Address is saved in the database along with a unique nonce

4. If the signature is valid, an access token is generated.

6. For any secured endpoint, backend first checks if the access token is valid