Lesson 10 - zkEVM Solutions

Today's topics

- Introduction to zkEVM solutions 
- zkSync, architecture and development
- Polygon products and Polygon zkEVM
- Scroll
- Taiko
- Kakarot
- Linea

---

zkEVM Solutions

Rollup Recap

Rollups are solutions that have

- transaction execution outside layer 1
- transaction data and proof of transactions is on layer 1
- a rollup smart contract in layer 1 that can enforce correct transaction execution on layer 2 by using the transaction data on layer 1

The main chain holds funds and commitments to the side chains
The side chain holds additional state and performs execution
There needs to be some proof, either a fraud proof (Optimistic) or a validity proof (zk)

Rollups require “operators” to stake a bond in the rollup contract. This incentivises operators to verify and execute transactions correctly.

Introductory video
zkEVMs

zkEVM overview - Dune dashboard

https://dune.com/cryptokoryo/zk

Timeline

2013 - TinyRAM
2018 - Spice
2020 - zkSync
2021 - Cairo VM
2022 - Polygon zkEVM / Scroll / Risc Zero
2023 - Powdr

zkVM introduction

Background - Virtual Machines

See article
Virtual Machines (VMs) are designed to provide the functionality a computer, usually wholly in software (there is a subtle difference between virtualisation and emulation).
The Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software.
There are 3 main types , register based, stack based and accumulator based, most VMs are of the first 2 types.
The Ethereum Virtual Machine is stack based.

In this course we will look at zkEVMs such as Polygon zkEVM / zkSync Era, and also zkVMs such as Risc zero.

A zkVM is a zero-knowledge proof-based virtual machine that combines a zk proof and a Virtual Machine. The zkVM typically consists of two important components: a compiler that can compile high-level languages such as C++ and Rust into intermediate (IR) expressions for the ZK system to perform; the other is the ISA (Instruction Architecture) instruction set framework, which mainly executes instructions about CPU operations and is a series of instructions used to instruct the CPU to perform operations.

zkEVM Phases

Proof focussed

- Circuit creation
- Setup
  - Parameters created - gives proving key and verification key
- Proof creation
- Proof aggregation
- Proof acceptance on L1 and verification

L2 aspects

As we have seen for validity rollups we also have processes to

- Submit data to the DA layer
- Allow L1 <-> L2 messaging
- Provide an escape hatch via forced transactions

zkEVM workflow

The general workflow for an zkEVM would be

- Receive a transaction
- Execute the relevant bytecode
- Make state changes and transaction receipts
- Using the zkEVM circuits with the execution trace as input produce a proof of correct execution
- Aggregate proofs for a bundle of transactions and submit them to L1
- Submit data to the appropriate data availability layer.

Comparing the L1 and L2 processing

Even though the external workflow remains unchanged, there are significant differences in the underlying processing procedures for Layer 1 and Layer 2:

- Layer 1 relies on the re-execution of smart contracts.
- Layer 2 relies on the validity proof of zkEVM circuits.

In Layer 1, the bytecode of deployed smart contracts are stored in Ethereum storage. Transactions are broadcasted across a peer-to-peer network. For each transaction, every full node must load the corresponding bytecode and execute it on the Ethereum Virtual Machine (EVM) to reach the same state, using the transaction as input data.

In Layer 2, bytecode is also stored in storage, and users follow a similar process. Transactions are sent off-chain to a centralized zkEVM node. Instead of directly executing the bytecode, zkEVM generates a succinct proof demonstrating that the states are correctly updated after applying the transactions. Finally, the Layer 1 contract verifies the proof and updates the states without re-executing the transactions.

On L1 the EVM loads the bytecode and executes the opcodes within it one by one.
Each opcode can be broken down into three sub-steps:

- Reading elements from the stack, memory, or storage,
- Performing computations on these elements, and
- Writing back the results to the stack, memory, or storage. For instance, the "add" opcode reads two elements from the stack, adds them together, and writes the result back to the stack.

Thus, it is evident that zkEVM's proof must encompass the following aspects related to the execution process:

- Confirming that the bytecode is accurately loaded from persistent storage (ensuring the correct opcode is loaded from a specified address).
- Demonstrating that the opcodes in the bytecode are executed sequentially without missing or skipping any opcode.

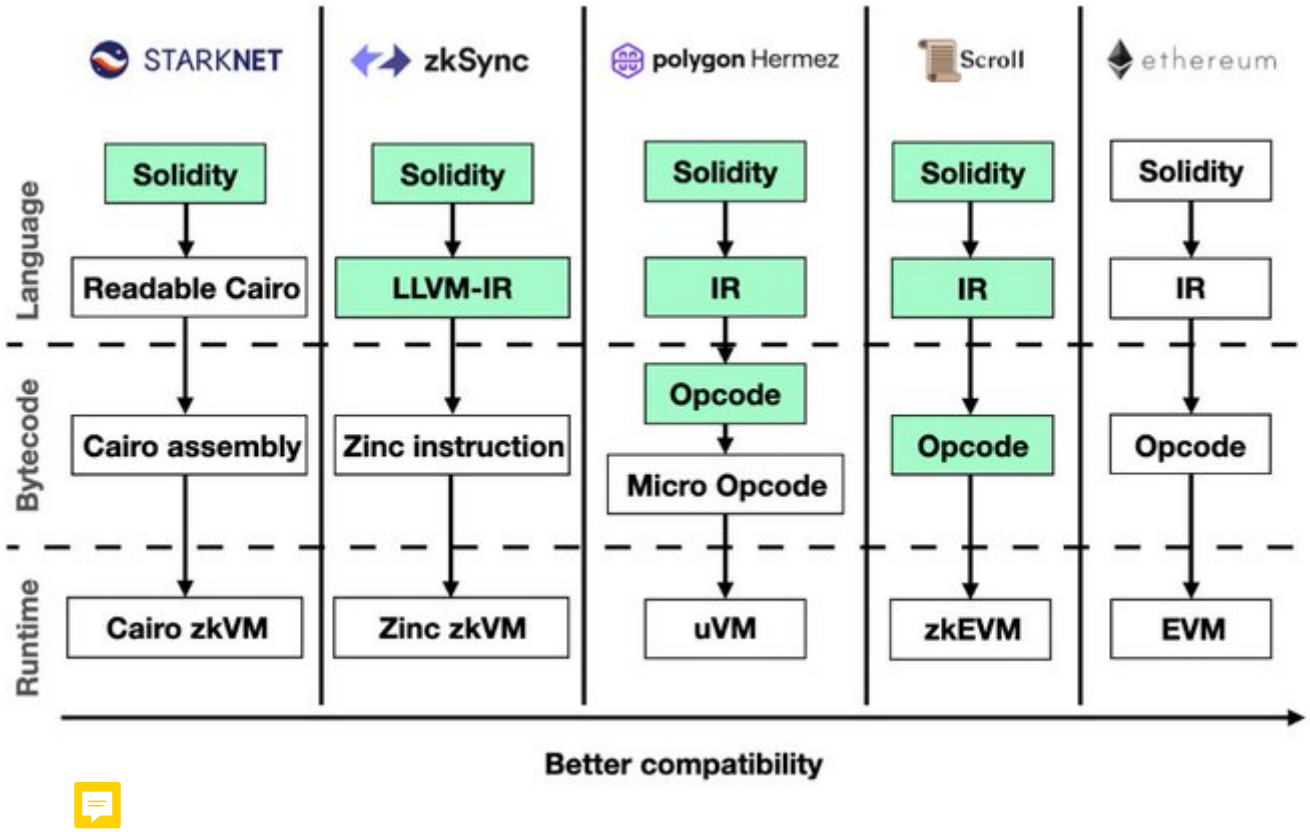
- Verifying the correct execution of each opcode, including the proper execution of the three sub-steps within each opcode (Read/Write and computation
  We will see this in much more detail in Lesson 12.

---

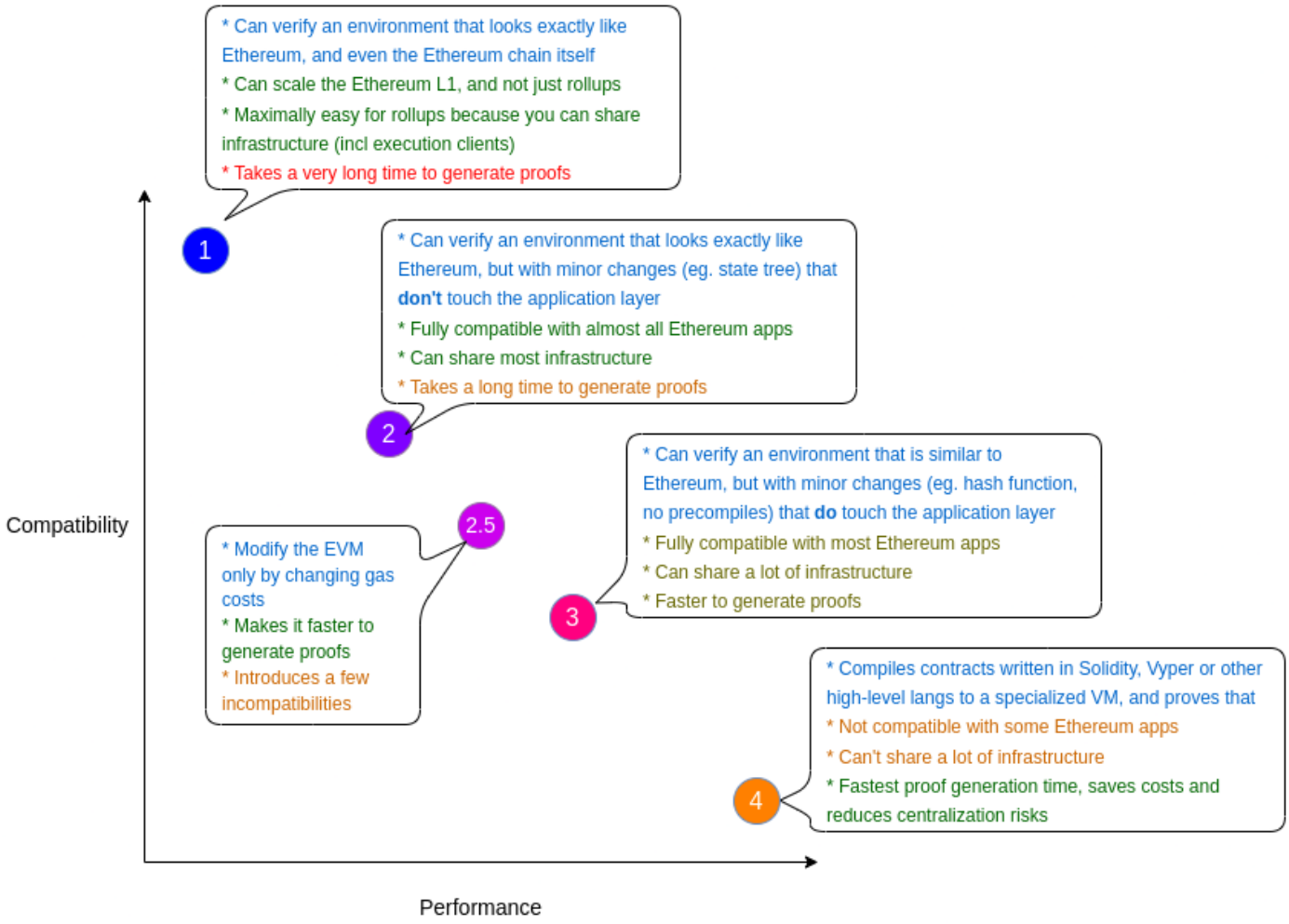
zkEVM taxonomy

From article and article

Approaches to zkRollups on Ethereum



See Vitalik's article



Type 1 (fully Ethereum-equivalent)

See zkEVM research team

Type 2 (fully EVM-equivalent)

(not quite Ethereum-equivalent)

Type 2.5 (EVM-equivalent, except for gas costs)

Type 3 (almost EVM-equivalent)

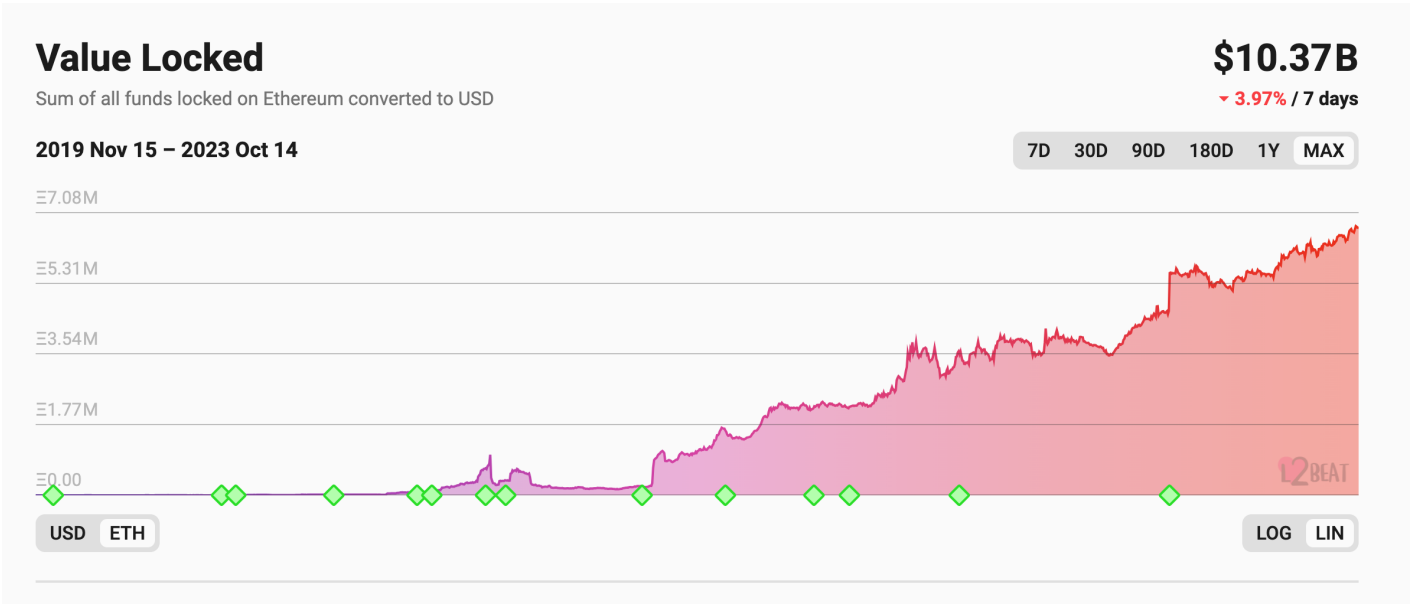
Type 4 (high-level-language equivalent)

Also see

The ZK-EVM Community Edition (bootstrapped by community contributors including Privacy and Scaling Explorations, the Scroll team, Taiko and others) is a Tier 1 ZK-EVM.

Main zkEVM Projects

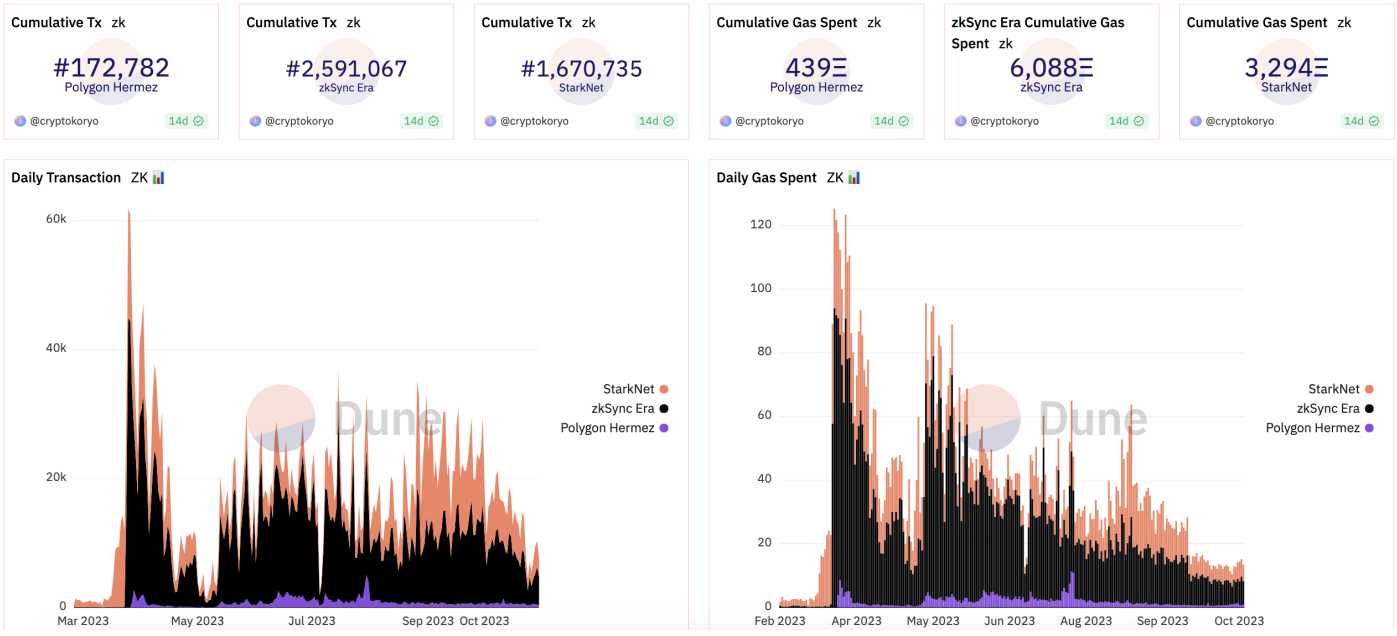
From L2Beat
TVL for all L2s



#	NAME	RISKS	TECHNOLOGY	STAGE	PURPOSE	TOTAL	MKT SHARE
1	Arbitrum One		Optimistic Rollup	STAGE 1	Universal	$5.70B 3.33%	54.94%
2	OP Mainnet		Optimistic Rollup	STAGE 0	Universal	$2.57B 6.09%	24.79%
3	Base		Optimistic Rollup	STAGE 0	Universal	$546M 2.13%	5.26%
4	zkSync Era		ZK Rollup	STAGE 0	Universal	$399M 10.60%	3.85%
5	dYdX		ZK Rollup	STAGE 1	Exchange	$325M 5.42%	3.14%
6	Starknet		ZK Rollup	STAGE 0	Universal	$131M 7.41%	1.26%
7	Loopring		ZK Rollup	STAGE 0	Tokens, NFTs, AMM	$81.31M 4.41%	0.78%
8	zkSync Lite		ZK Rollup	STAGE 1	Payments, Tokens	$66.23M 9.35%	0.64%
9	Linea		ZK Rollup	STAGE 0	Universal	$66.22M 3.79%	0.64%
10	Polygon zkEVM		ZK Rollup	STAGE 0	Universal	$52.47M 3.85%	0.51%

11	ZKSpace	ZK Rollup	STAGE 0	Tokens, NFTs, AMM	$19.56M ▼ 5.75%	0.19%
12	Manta Pacific	Optimistic Rollup	STAGE 0	Universal	$10.24M ▲ 7.78%	0.10%
13	Boba Network	Optimistic Rollup	STAGE 0	Universal	$8.54M ▼ 3.75%	0.08%
14	Aevo	Optimistic Rollup	STAGE 0	DEX	$6.71M ▼ 0.43%	0.06%
15	Zora	Optimistic Rollup	STAGE 0	Universal, NFTs	$6.25M ▼ 8.30%	0.06%
16	Aztec Connect	ZK Rollup	STAGE 0	Private DeFi	$4.32M ▼ 5.77%	0.04%
17	DeGate V1	ZK Rollup	STAGE 2	Exchange	$4.04M ▲ 31.15%	0.04%
18	Aztec	ZK Rollup	STAGE 0	Private payments	$2.03M ▼ 5.29%	0.02%
19	Scroll	ZK Rollup	IN REVIEW	Universal	$624K ▲ 0.00%	0.01%
20	Public Goods Network	Optimistic Rollup	STAGE 0	Universal	$537K ▼ 3.08%	0.01%
21	Kroma	Optimistic Rollup	STAGE 0	Universal	$266K ▼ 4.36%	0.00%
22	Honeypot (Cartesi)	Optimistic Rollup	STAGE 0	Bug bounty	$5.30K ▲ 10.49%	0.00%
23	Fuel v1	Optimistic Rollup	STAGE 2	Payments	$406 ▼ 5.38%	0.00%

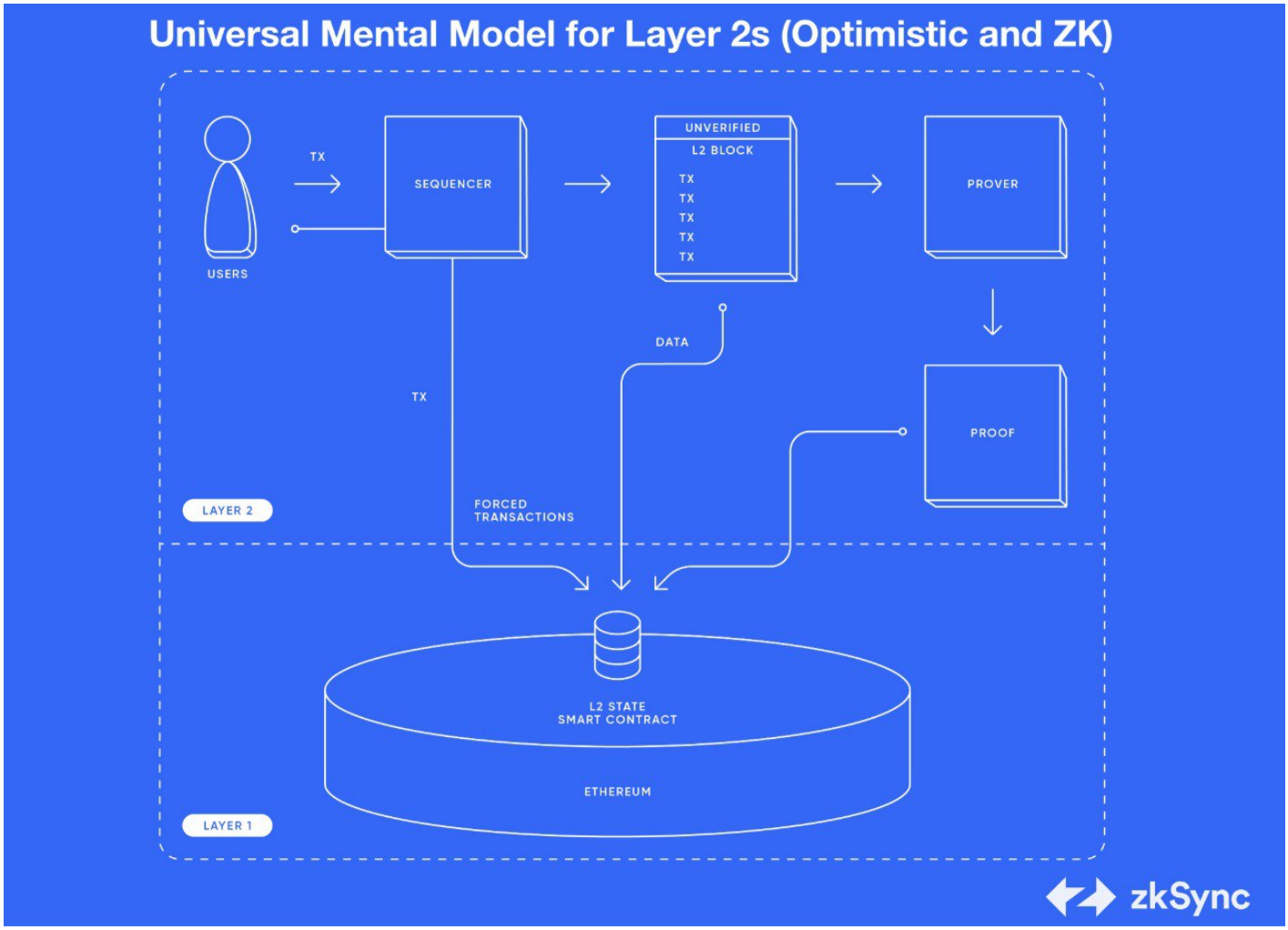
See Dune [dashboard](#)



zkSync Architecture

From zkSync Docs :
A layer 2 relies on the L1 for security guarantees :

- The Rollup validator(s) can never corrupt the state or steal funds (unlike Sidechains).
- Users can always retrieve the funds from the Rollup even if validator(s) stop cooperating because the data is available (unlike Plasma).
- Thanks to validity proofs, neither users nor a single other trusted party needs to be online to monitor Rollup blocks in order to prevent fraud (unlike payment channels or Optimistic Rollups).



zkSync Development

See General Docs
See Smart Contract Docs

Zinc

The language Zinc is being deprecated in favour of Solidity.

Dev Tools

https://era.zksync.io/docs/tools/

CLI

https://era.zksync.io/docs/tools/zksync-cli/

See tutorial using the CLI

Dev tools

Foundry

A fork of Foundry has been created - `foundry-zksync`
See Docs

Features

- **Smart Contract Deployment**: Easily deploy smart contracts to zkSync mainnet, testnet, or a local test node.
- **Asset Bridging**: Bridge assets between L1 and L2, facilitating seamless transactions across layers.
- **Contract Interaction**: Call and send transactions to deployed contracts on zkSync testnet or local test node.
- **Solidity Testing**: Write tests in Solidity for a familiar testing environment.
- **Fuzz Testing**: Benefit from fuzz testing, complete with shrinking of inputs and printing of counter-examples.
- **Remote RPC Forking**: Utilize remote RPC forking mode.
- **Flexible Debug Logging**: Choose your debugging style:
  - DappTools-style: Utilize DsTest's emitted logs for debugging.
  - Hardhat-style: Leverage the popular console.sol contract.
- **Configurable Compiler Options**: Tailor compiler settings to your needs, including LLVM optimization modes.

However be aware that some functionality found in the original Foundry may not yet be available.

The documentation gives a good list of resources.

Polygon Products

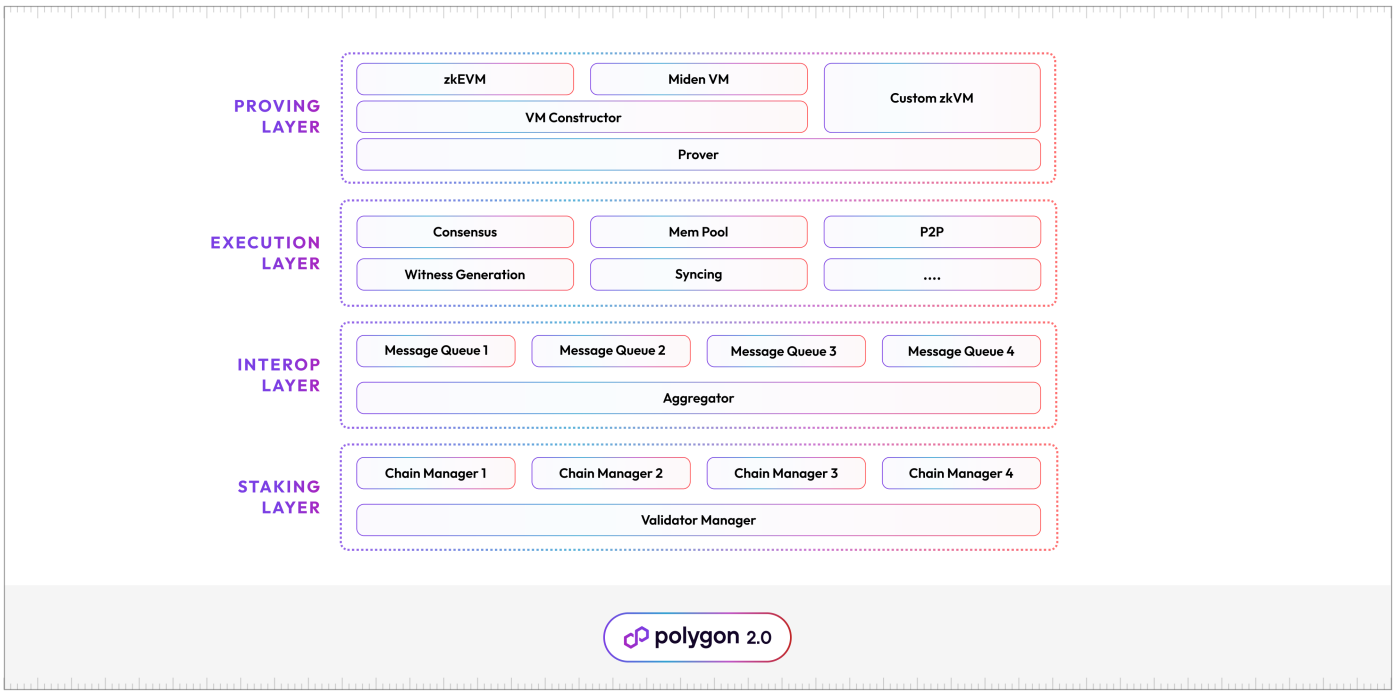
See this guide
Strategy article

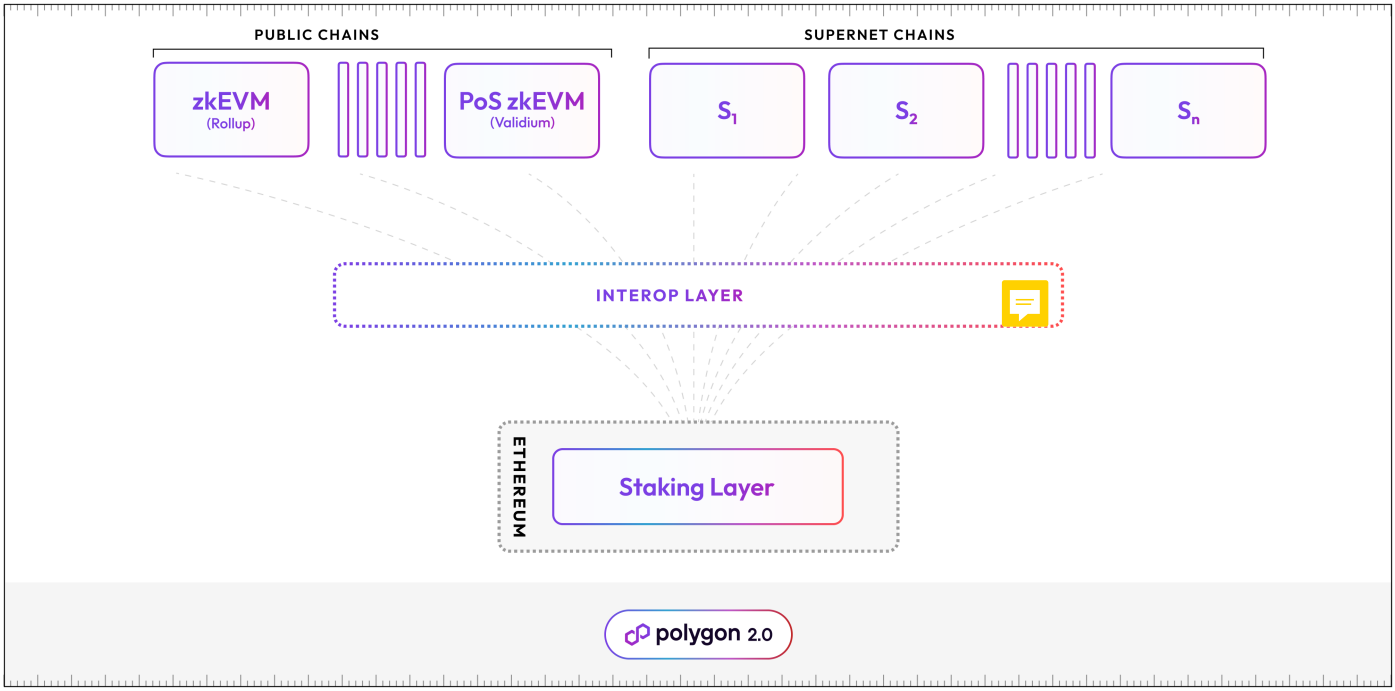
Summary

- Polygon Edge - Infrastructure
- Polygon PoS - Original side chain
- Polygon zkEVM - zkEVM based rollup
- Polygon CDK - App chain development
- Polygon Miden - zkVM using STARKs

Architecture

See blog of vision and proposed architecture





Polygon Nightfall

From a collaboration with EY it is designed to allow private transactions.
It is a combination Optimistic rollups and zero knowledge, optimistic rollups for scalability and zk for privacy.
There is a beta version available on mainnet.

Polygon zkEVM

See Repo

- `polygon zkEVM` is a new zk-rollup that provides Ethereum Virtual Machine (EVM) equivalence (opcode-level compatibility) for a transparent user experience and existing Ethereum ecosystem and tooling compatibility.
- It consists on a decentralized Ethereum Layer 2 scalability solution utilising cryptographic zero-knowledge technology to provide validation and fast finality of off-chain transaction computations.
- This approach required the recreation of all EVM opcodes for transparent deployment and transactions with existing Ethereum smart contracts. For this purpose a new set of tools and technologies were created and engineered and are contained in this organization.

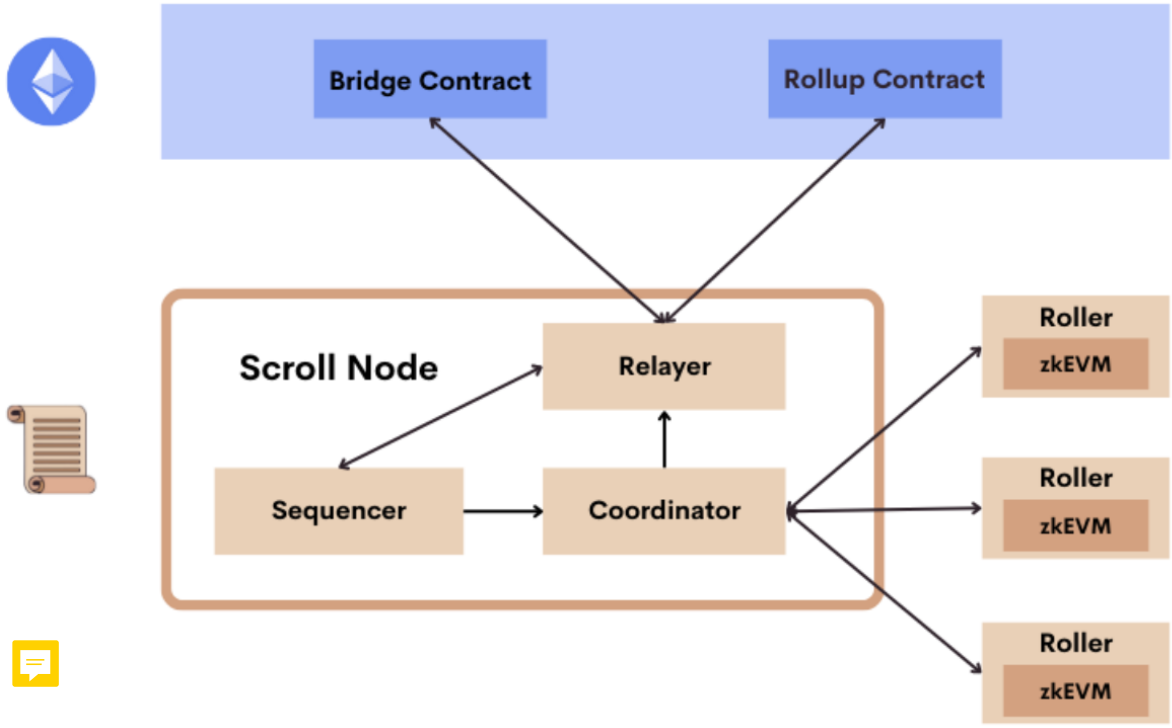
Scroll

This project is still at en early stage, there alpha testnet will be run on a private PoA fork of Ethereum (the testnet L1) operated by Scroll.
On top of this private chain, will run a testnet Scroll L2 supporting the following features:

- Users will be able to play with a few key demo applications such as a Uniswap fork with familiar web interfaces such as Metamask.
- Users will be able to view the state of the Scroll testnet via block explorers.
- Scroll will run a node that supports unlimited read operations (e.g. getting the state of accounts) and user-initiated transactions involving interactions with the pre-deployed demo applications (e.g. transfers of ERC-20 tokens or swaps of tokens).
- Rollers will generate and aggregate validity proofs for part of the zkEVM circuits to ensure a stable release. In the next testnet phase, we will ramp up this set of zkEVM circuits.
- Bridging assets between these testnet L1 and L2s will be enabled through a smart contract bridge, though arbitrary message passing will not be supported in this release.

From article and article

Scroll Architecture



Components

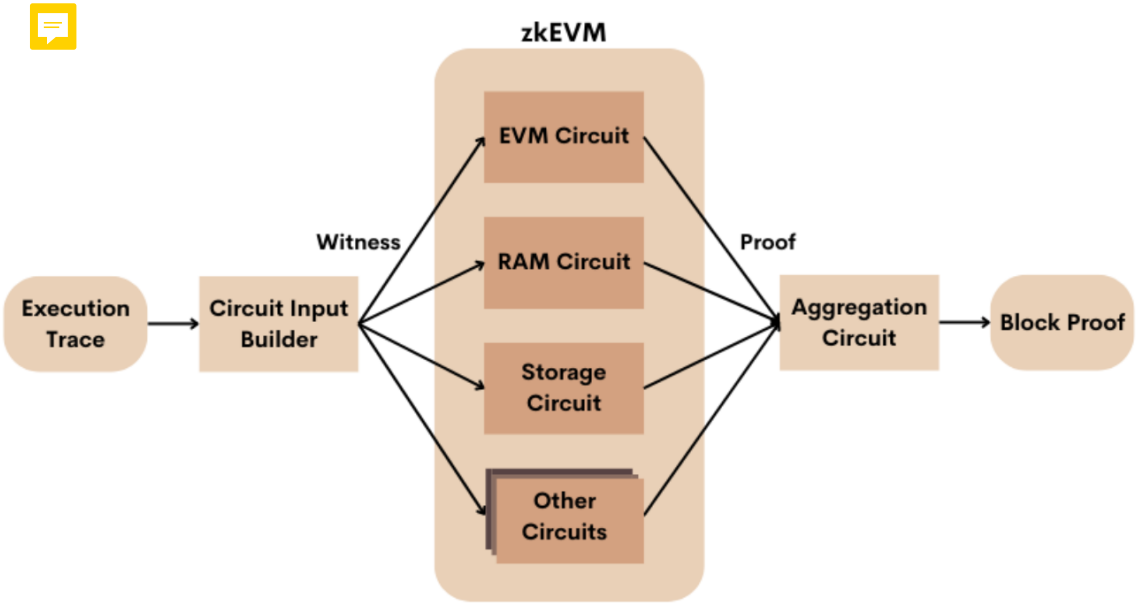
The **Sequencer** provides a JSON-RPC interface and accepts L2 transactions. Every few seconds, it retrieves a batch of transactions from the L2 mempool and executes them to generate a new L2 block and a new state root.

Once a new block is generated, the **Coordinator** is notified and receives the execution trace of this block from the Sequencer.
It then dispatches the execution trace to a randomly-selected **Roller** from the roller pool for proof generation.

The **Relayer** watches the bridge and rollup contracts deployed on both Ethereum and Scroll. It has two main responsibilities.

1. It monitors the rollup contract to keep track of the status of L2 blocks including their data availability and validity proof.
2. It watches the deposit and withdraw events from the bridge contracts deployed on both Ethereum and Scroll and relays the messages from one side to the other.

Rollers - creating proofs

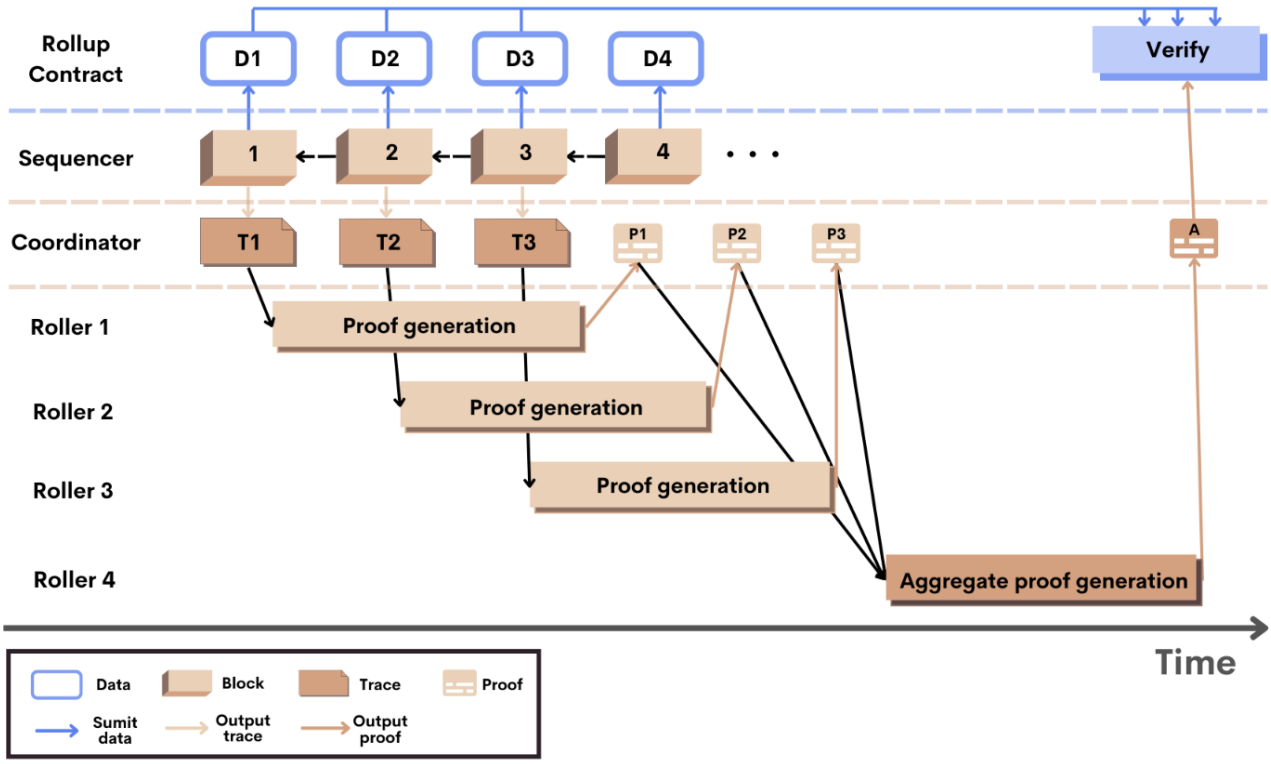


The **Rollers** serve as provers in the network that are responsible for generating validity proofs for the zkRollup

- A Roller first converts the execution trace received from the **Coordinator** to circuit witnesses.
- It generates proofs for each of the **zkEVM** circuits.
- Finally, it uses **proof aggregation** to combine proofs from multiple zkEVM circuits into a single block proof.

The **Rollup contract** on L1 receives L2 state roots and blocks from the Sequencer.
It stores state roots in the Ethereum state and L2 block data as Ethereum calldata.
This provides **data availability** for Scroll blocks and leverages the security of Ethereum to ensure that indexers including the Scroll Relayer can reconstruct L2 blocks.
Once a block proof establishing the validity of an L2 block has been verified by the Rollup contract, the corresponding block is considered finalized on Scroll.

A useful sequence diagram from the Scroll Documentation



L2 blocks in Scroll are generated, committed to base layer Ethereum, and finalized in the following sequence of steps:

1. The Sequencer generates a sequence of blocks. For the *i*-th block, the Sequencer generates an execution trace ***T*** and sends it to the Coordinator. Meanwhile, it also submits the transaction data ***D*** as calldata to the Rollup contract on Ethereum for data availability and the resulting state roots and commitments to the transaction data to the Rollup contract as state.
2. The Coordinator randomly selects a Roller to generate a validity proof for each block trace. To speed up the proof generation process, proofs for different blocks can be generated in parallel on different Rollers.
3. After generating the block proof ***P*** for the *i*-th block, the Roller sends it back to the Coordinator. Every *k* blocks, the Coordinator dispatches an aggregation task to another Roller to aggregate *k* block proofs into a single aggregate proof ***A***.
4. Finally, the Coordinator submits the aggregate proof ***A*** to the Rollup contract to finalize L2 blocks *i+1* to *i+k* by verifying the aggregate proof against the state roots and transaction data commitments previously submitted to the rollup contract.

Scroll circuit design

1. We need an accumulator to provide the proofs of storage, merkle trees can provide this
2. The execution trace is needed to show the path that the execution took through the bytecode, as this would change because of jumps. This trace is then a witness provided to the circuit.
3. Two proofs are used to show the execution is correct for each opcode
    1. Proof of fetching the data required for the opcode
    2. Proof that the opcode executed correctly.

Scroll are working with Ethereum on this, see this repo for EVM circuit design, and this design document from Ethereum.

Design Choices

See this article

Features

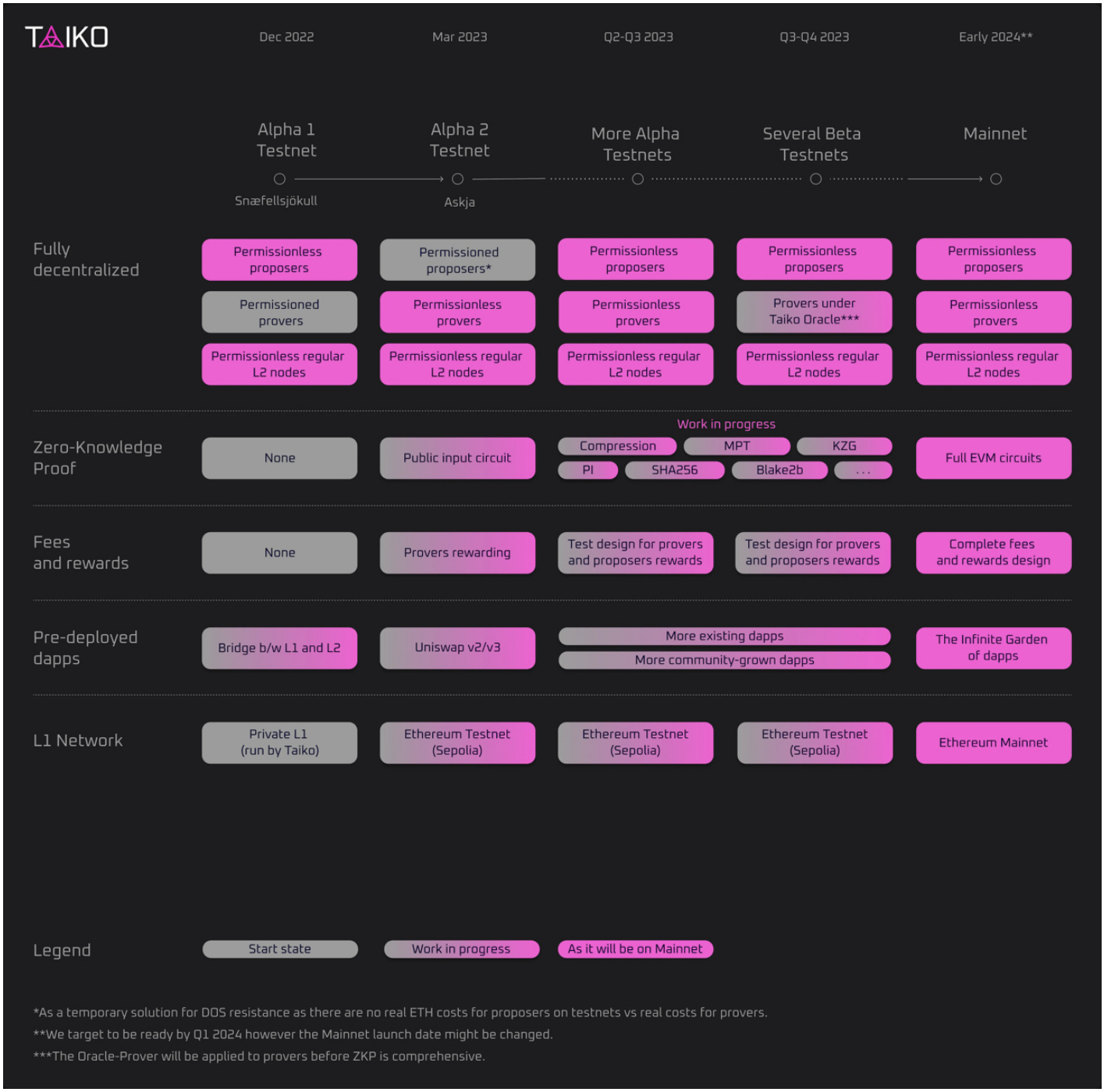
- Community driven development and broad education
- Be mindful of security and ensure a steady release
- Decentralisation at all layers is important
- Develop for Ethereum also

Taiko overview

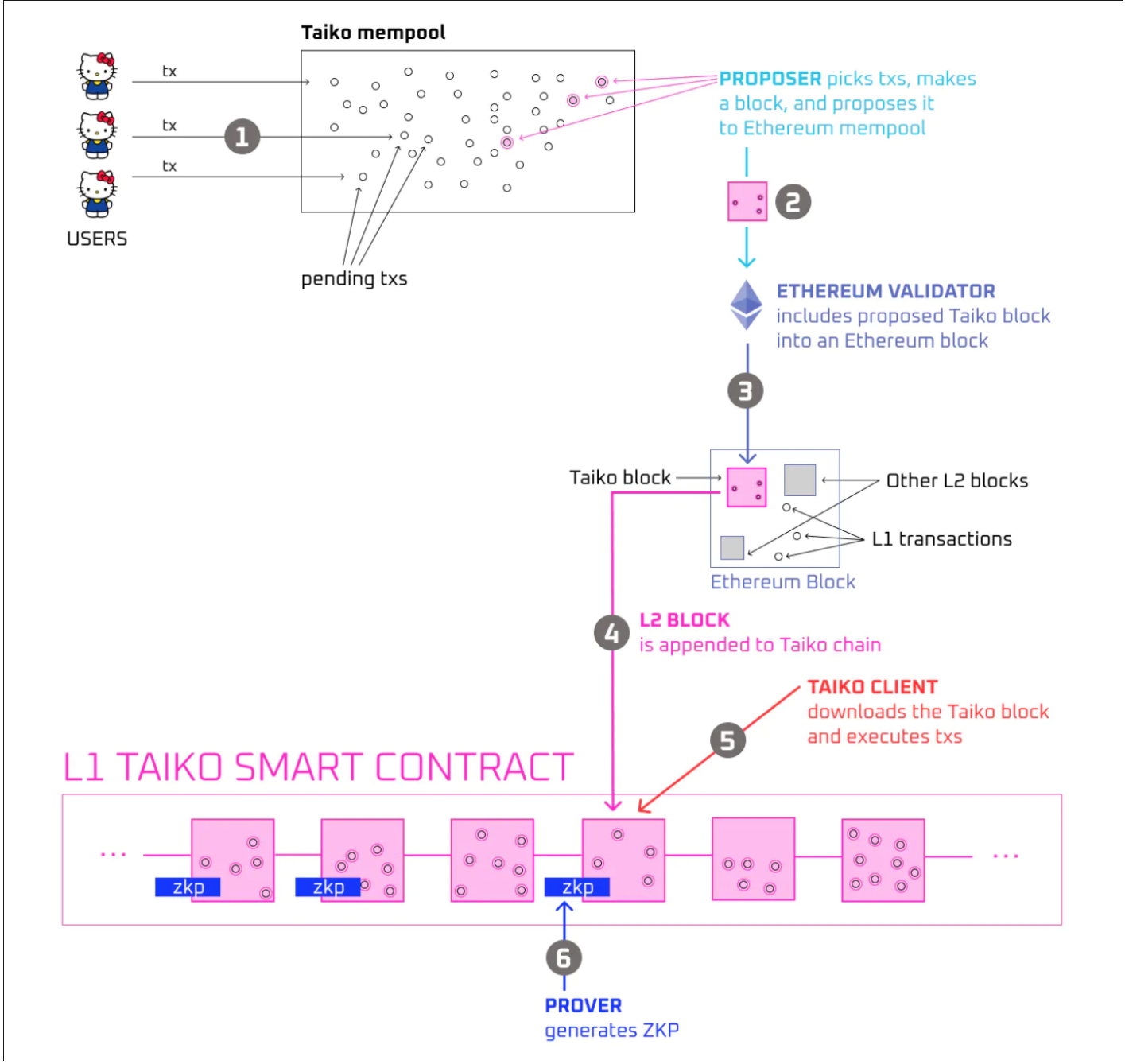
See Docs

"Taiko is a decentralised, Ethereum-equivalent ZK-Rollup (Type 1 ZK-EVM). Taiko is working on the full Ethereum ZK-EVM circuits as part of a community effort led by the EF's PSE team"

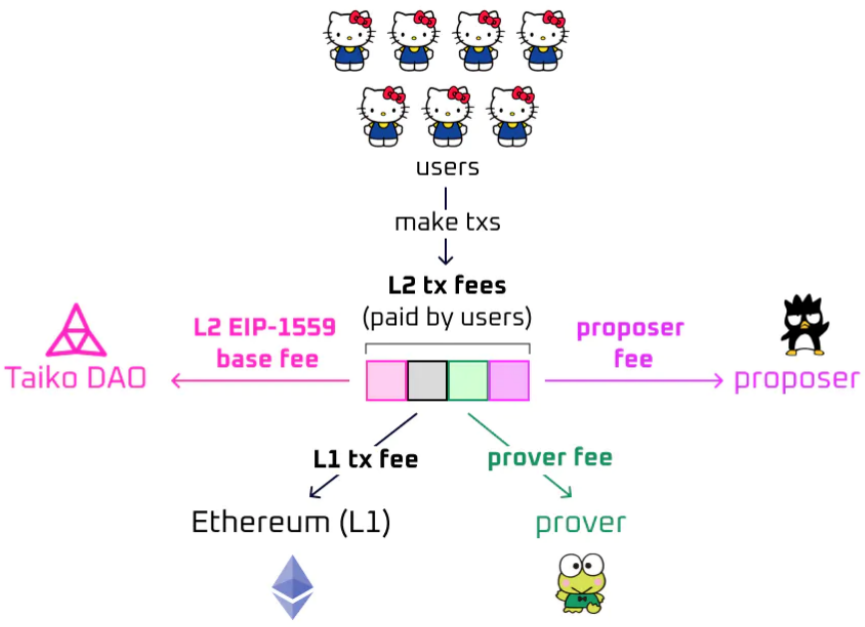
Roadmap



Process



Block Proposal



Block Validation

The block consists of a transaction list and metadata.

Once the block is proposed, the Taiko client checks if the block is decoded into a list of transactions; if the txList is not decodable, then the block is an empty block with only an anchoring tx.

The Taiko client validates each enclosed transaction and generates a tracelog for each transaction for the prover to use as witness data.
If a tx is invalid, it will be dropped.

The first transaction in the Taiko L2 block is always an anchoring transaction
The anchoring transaction contains some extra data not directly available by the ZK-EVM itself, namely :

- the 256 hashes of the latest blocks (that are not a part of Merkle Tree);
- L2's chain ID and EIP-1559 base fee.

Block Proving

The block can be proven as soon as

1. All enclosed valid transactions in this block have been executed (necessary execution trace for computing ZKP was generated);
2. Its parent block's state is known.

The proof proves a transition from the parent block state to the new block state.

Running a prover

Anyone can run a prover.

As the block execution is deterministic, all Taiko clients can calculate the post-block state.
No one knows more about the latest state than anyone else.
For L2 users, the transaction is confirmed once the block is proposed, there is no need to wait for ZKPs.

Provers can submit proofs for any block they like.
Multiple provers may work in parallel to generate ZKPs for one or multiple Taiko blocks, but only the first proof will be accepted for any given block transition (fork choice). The address receiving the reward is coupled with the proof, preventing it from being stolen by other provers.

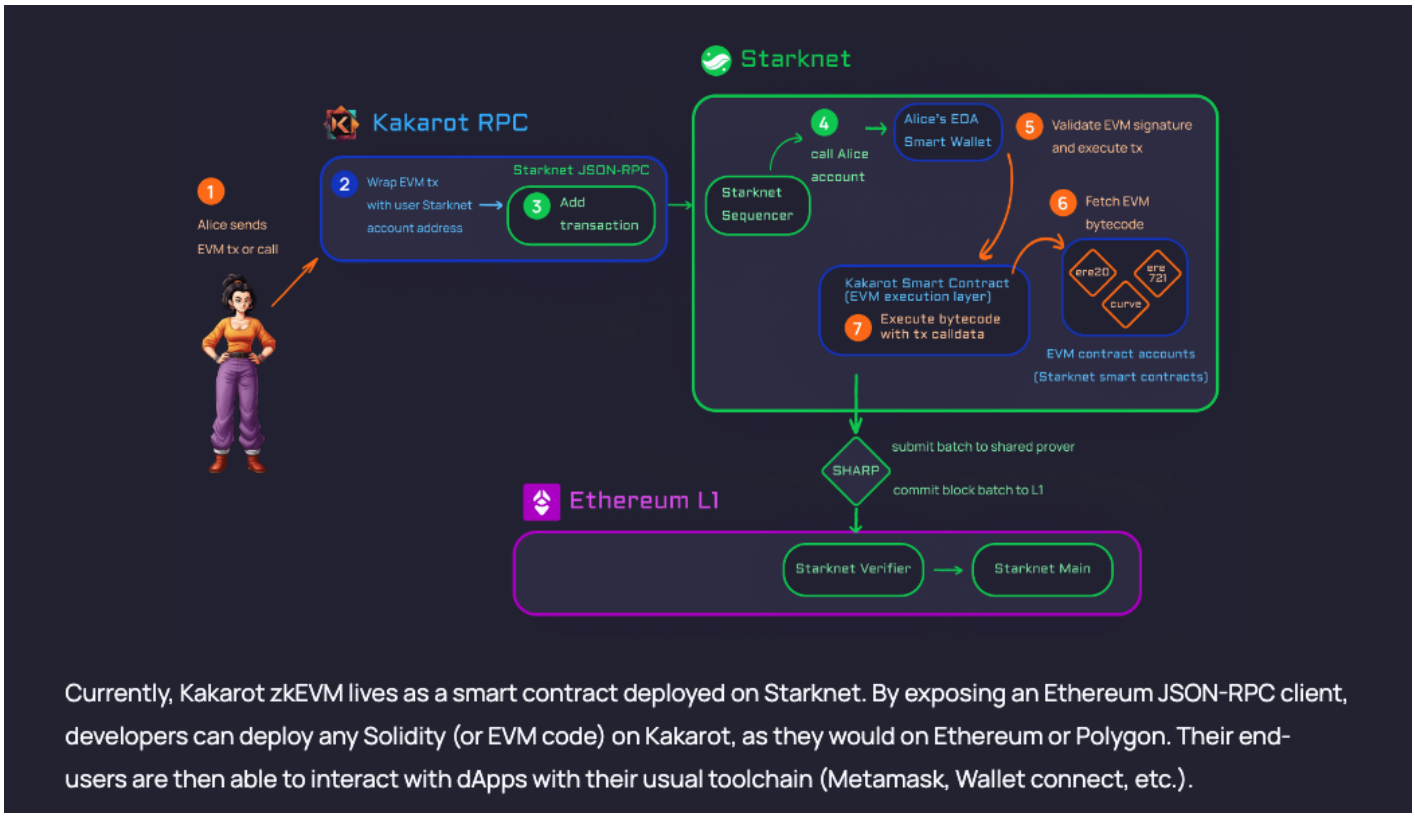
Oracle Prover

Currently Taiko also runs an 'Oracle prover', (though proof checker may be a better name) to mediate if there are any problems with proof generation.
The oracle prover checks all state transitions (by running a node to run over all transactions in the block) and may override existing state transitions (fork choices) that regular provers have proved.

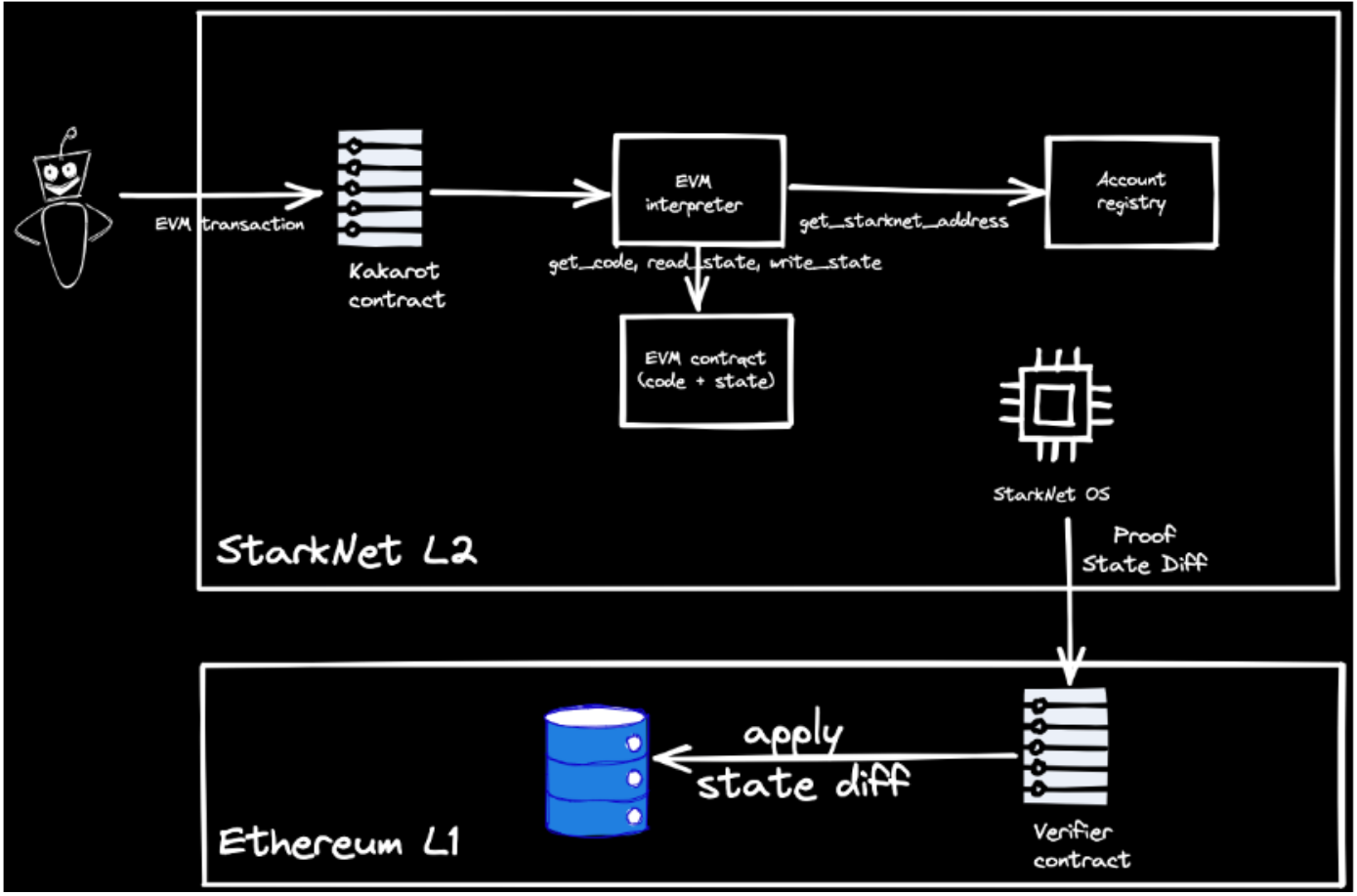
> *Note: the oracle prover cannot prove/verify blocks directly and thus cannot change the chain state. Therefore, a real ZKP is still required to prove the overridden fork choice.*

Kakarot

From their website



Currently, Kakarot zkEVM lives as a smart contract deployed on Starknet. By exposing an Ethereum JSON-RPC client, developers can deploy any Solidity (or EVM code) on Kakarot, as they would on Ethereum or Polygon. Their end-users are then able to interact with dApps with their usual toolchain (Metamask, Wallet connect, etc.).



- Kakarot can:
- (a) execute arbitrary EVM bytecode,
- (b) deploy an EVM smart contract as is,
- (c) call a Kakarot-deployed EVM smart contract's functions (views and write methods).
- Kakarot is an EVM bytecode interpreter.

Component Details

See Repo
The entire Kakarot protocol is composed of 4 different contracts:

- Kakarot (Core)
- Contract Accounts
- Account Registry
- Blockhash Registry

The main Kakarot contract is located at: `./src/kakarot/kakarot.cairo`.

This is the core contract which is capable of executing decoded ethereum transactions thanks to its `invoke` entrypoint.

Currently, Argent or Braavos accounts contracts don't work with Kakarot. Consequently, the `deploy_externally_owned_account` entrypoint has been added to let the owner of an Ethereum address get their corresponding starknet contract.

The mapping between EVM addresses and Starknet addresses of the deployed contracts is stored as follows:

- each deployed contract has a `get_evm_address` entrypoint
- only the Kakarot contract deploys accounts and provides a `compute_starknet_address(evm_address)` entrypoint that returns the corresponding starknet address

For this latter computation to be account agnostic, Kakarot indeed uses a transparent proxy.

Contract Accounts

A *Contract Account* is a StarkNet contract. However, it also acts as an Ethereum style contract account within the Kakarot EVM. In isolation it is not more than a StarkNet contract that stores some bytecode as well as some key-value pairs which were assigned to it on creation. It is only addressable via its StarkNet address and not an EVM address (which it is associated with inside the Kakarot EVM).

Externally Owned Account

Each Externally Owned Account in the EVM world has its counterpart in Starknet by the mean of a specific account contract deployed by Kakarot.

This contract is a regular account contract in the Starknet sense with `__validate__` and `__execute__` entrypoint. However, it does decode and validate an EVM signed transaction and redirect it only to Kakarot.
Further development will allow the user to have one single Starknet account for both Starknet native and Kakarot deployed dApp.

Blockhash Registry

The BLOCKHASH opcode is a particular opcode that requires the EVM to be aware of past blocks. Since this is not feasible from within Starknet, we deployed a block hash registry contract on Starknet to make this data accessible on-chain.

The blockhash registry enables this by holding a `block_number -> block_hash` mapping that admins can write to and Kakarot core can read from.

Supported Opcodes

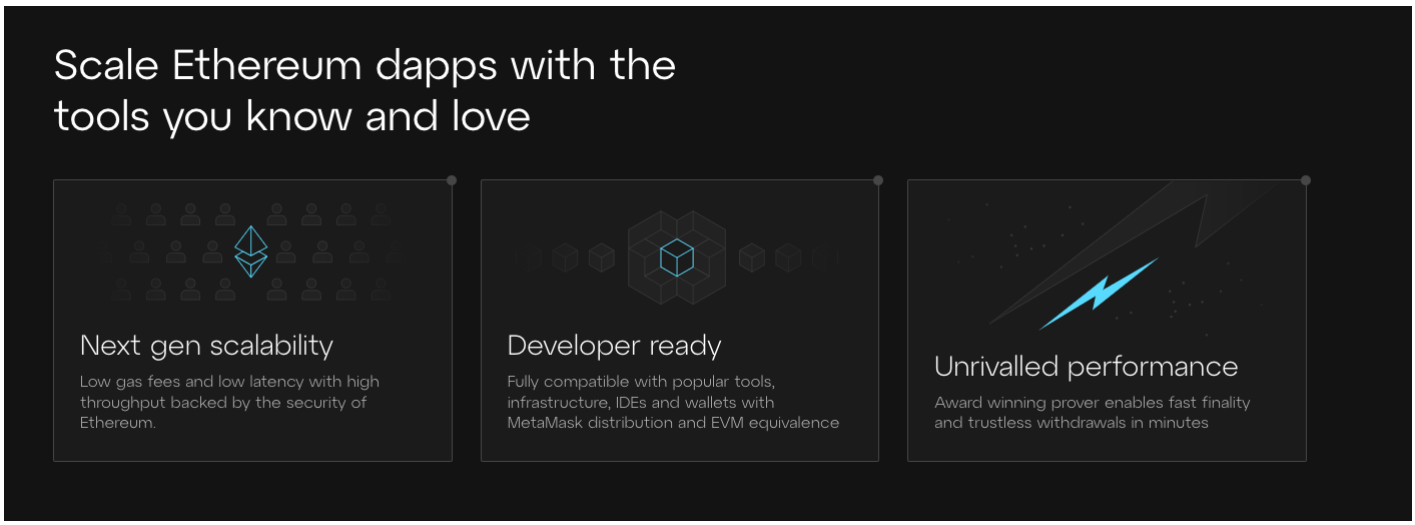
Kakarot currently supports 100% of EVM opcodes and 8 out of 9 precompiles.

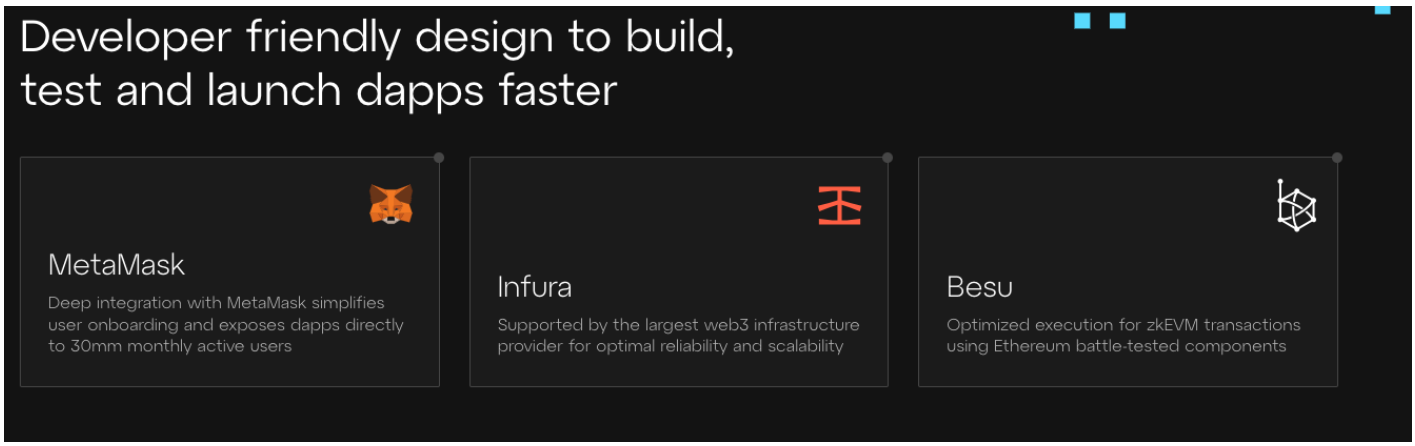
Resources

Presentation at Starkware Session
Repo

Linea

A zkEVM from Consensys
See site
See Documentation





Current progress from documentation

Linea's ideal state

Linea has the goal of being a fully decentralized, permissionless network. To that end, we are building towards an architecture made up of three main elements:

- Sequencer
- Prover
- Bridge Relayer

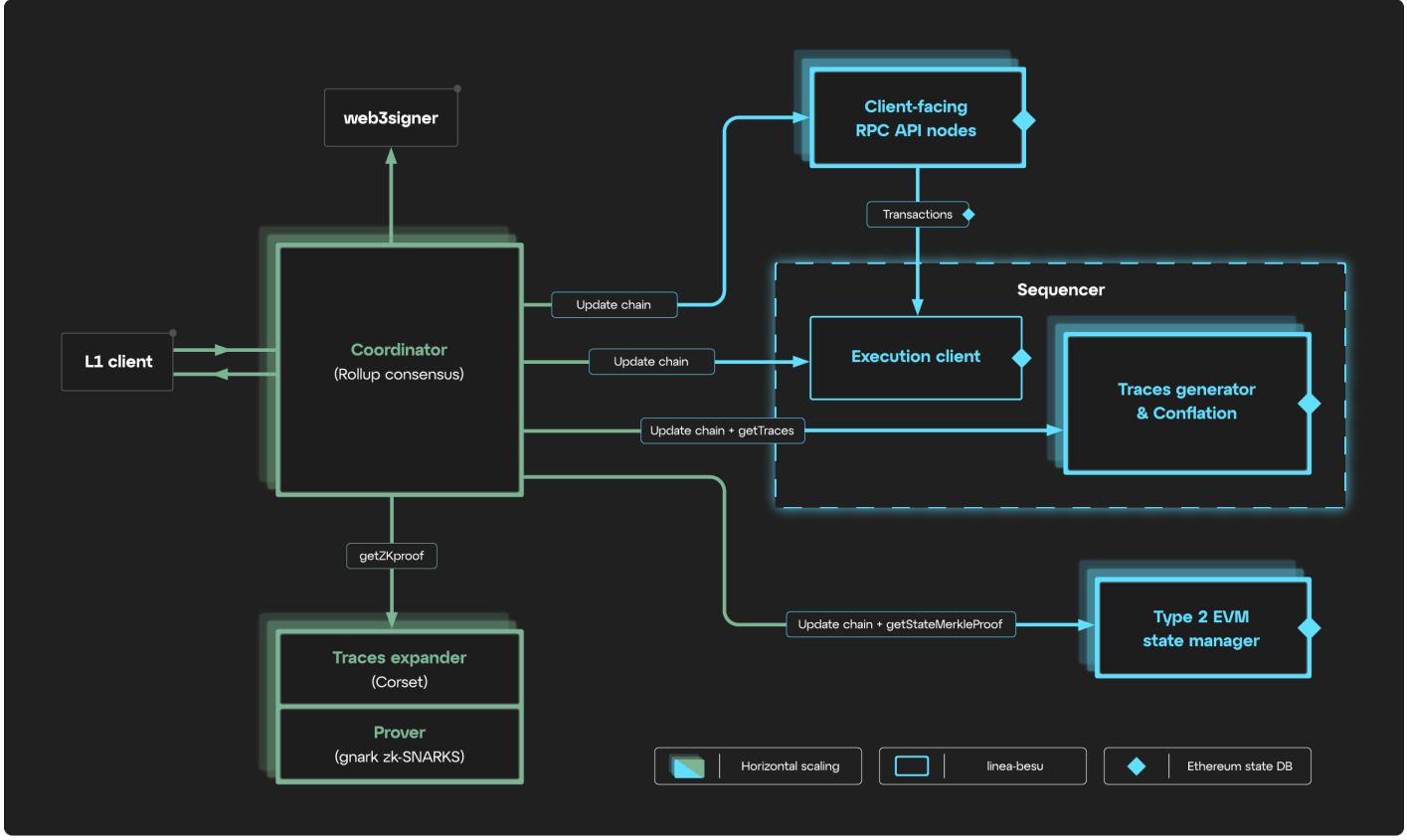
(Don't worry, if this isn't making sense to you yet, we'll explain further below 😎)

Current state

As Linea is still a beta testnet, and there's lots of development to be done, we're not quite there yet. Currently, the first two sections above are kind of rolled into one:

- Centralized Sequencer & Prover
- Bridge Relayer

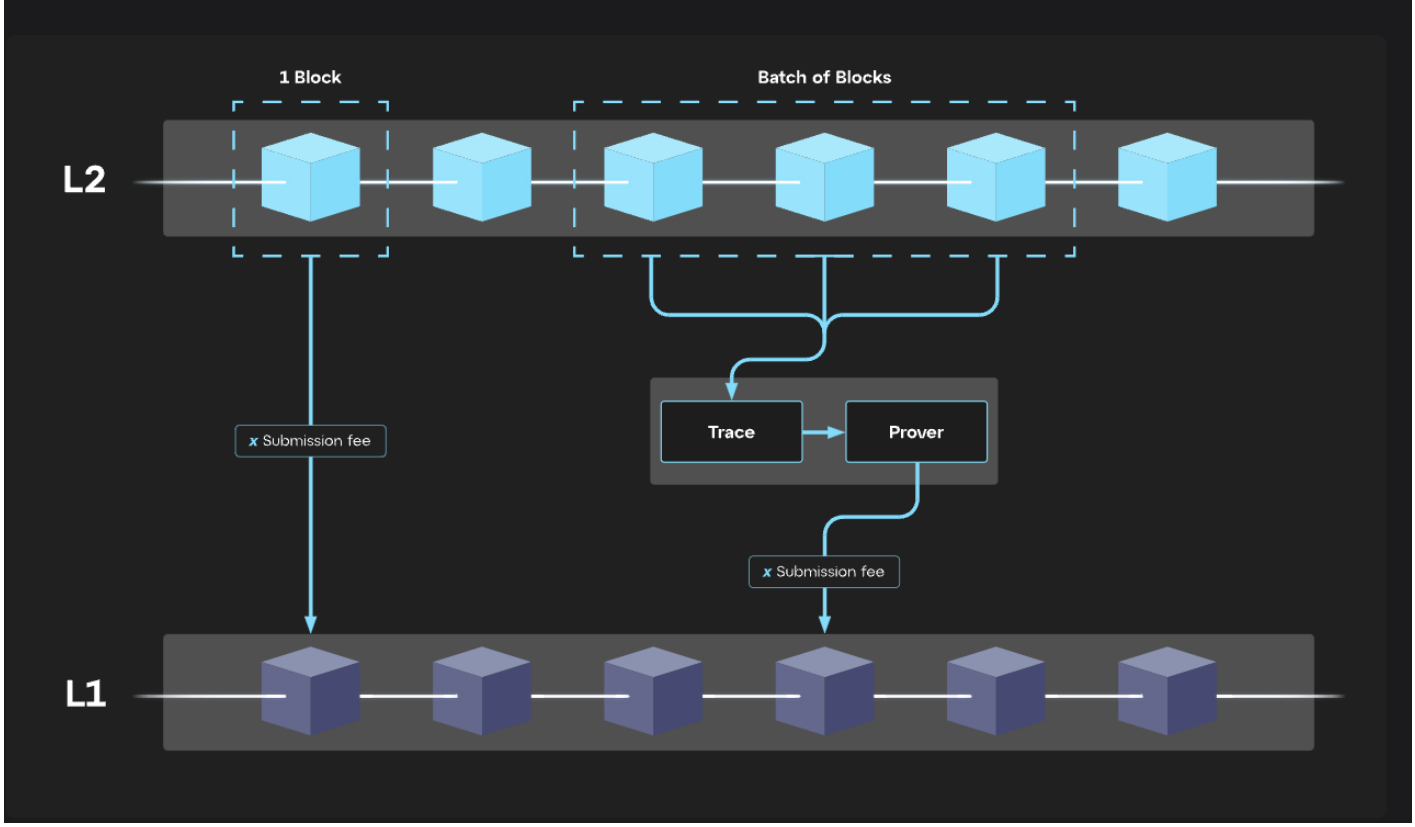
Linea Architecture



Sequencer

See Docs

Currently, Linea’s execution client is zkGeth, a version of geth that has been modified to work with zk-proving technology.
However, Linea is building linea-besu: leveraging the full power of the Consensys stack by using the same Besu client software that is used to execute blocks on Ethereum
Linea's sequencer takes transactions from the Linea memory pool, and builds them into blocks
It uses the 'Traces generator' and prover to create a trace and generate a proof and 'conflation' to combine state transitions from multiple blocks.



Linea SDK and Postman

See Docs
The SDK focuses on interacting with smart contracts on both Ethereum and Linea networks and provides custom functions to obtain message information.
The Linea SDK allows:

1. Getting contract instances and addresses
2. Getting message information by message hash
3. Getting messages by transaction hash
4. Getting a message status by message hash
5. Claiming messages (use one of the get message methods to grab all the parameters values)

EVM compatibility

Linea is based on the London fork of the EVM, more recent forks were more concerned with the merge and less specifically about EVM operation.

The additional opcodes supported by Linea are given in this table