Lesson 10 - zkEVM Solutions

zkEVM Solutions

Rollup Recap

Rollups are solutions that have

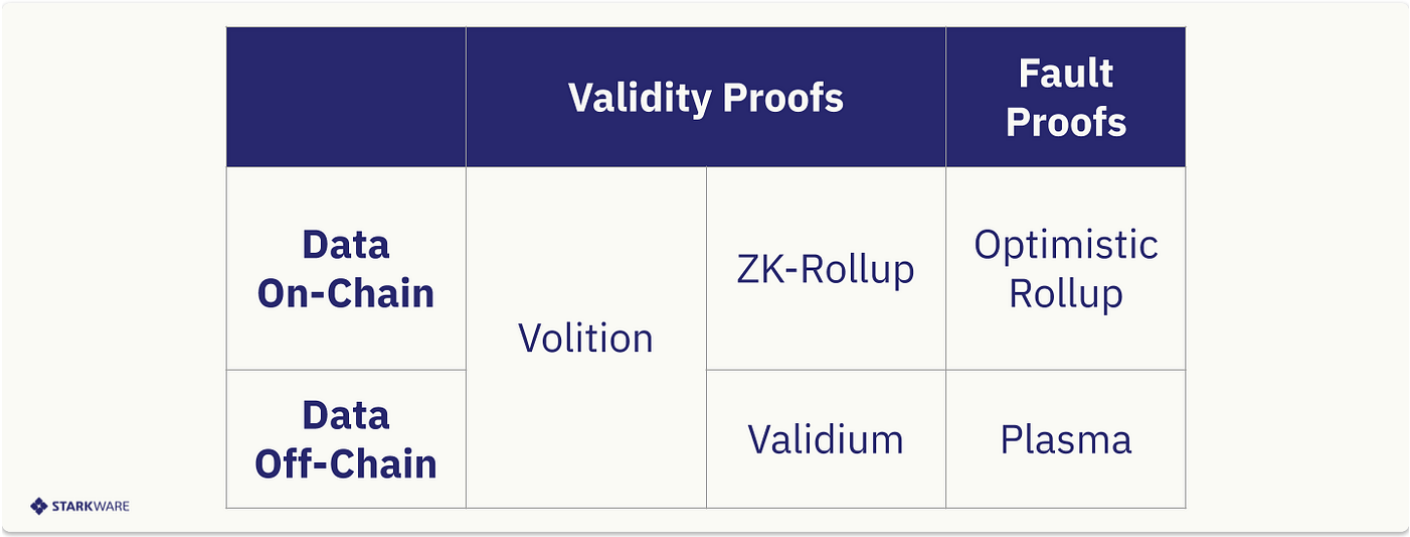
- transaction execution outside layer 1
- transaction data and proof of transactions is on layer 1
- a rollup smart contract in layer 1 that can enforce correct transaction execution on layer 2 by using the transaction data on layer 1

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)

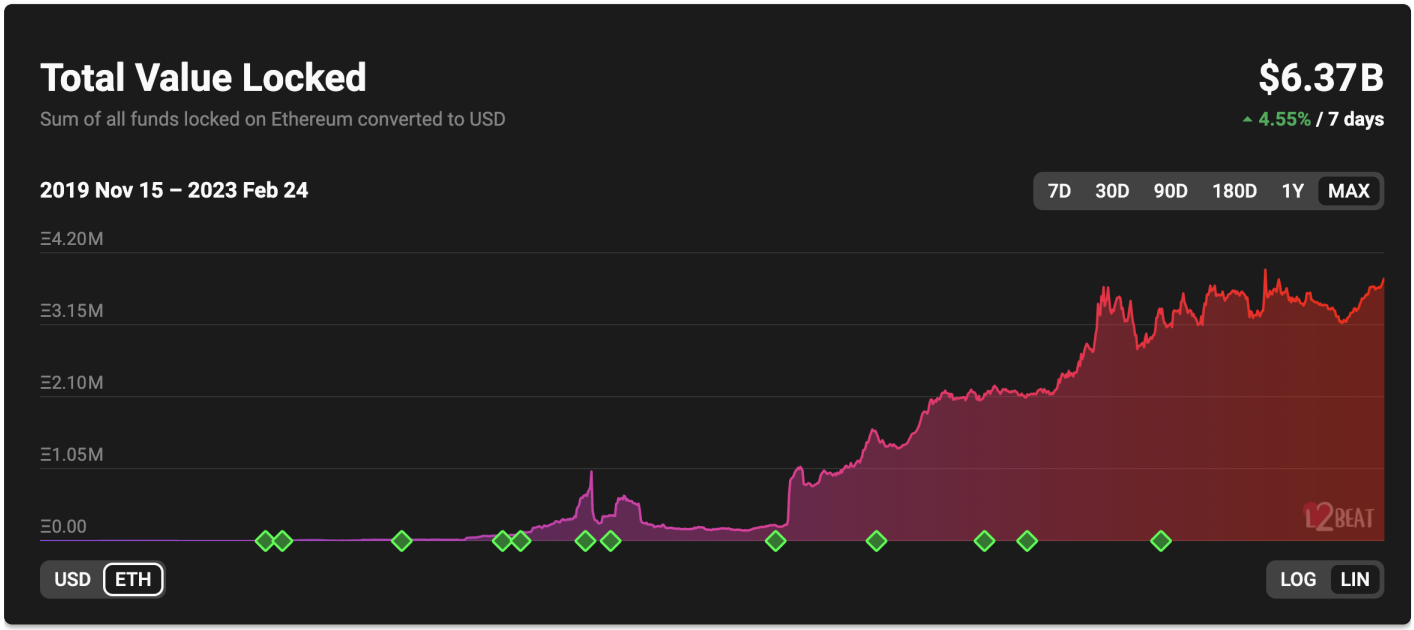
Rollups require “operators” to stake a bond in the rollup contract. This incentivises operators to verify and execute transactions correctly.

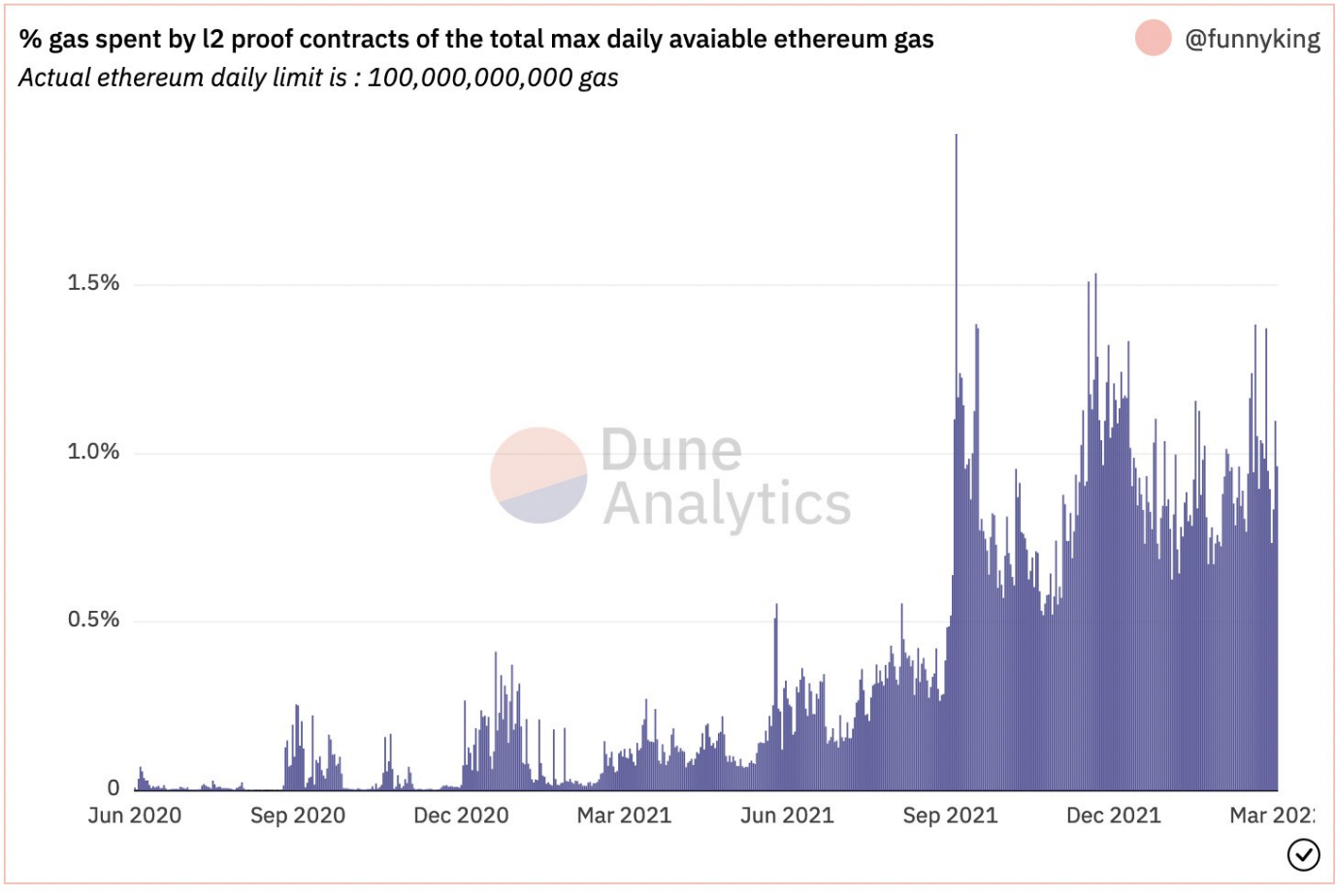
Data Availability in general

In order to re create the state, transaction data is needed, the data availability question is where this data is stored and how to make sure it is available to the participants in the system.



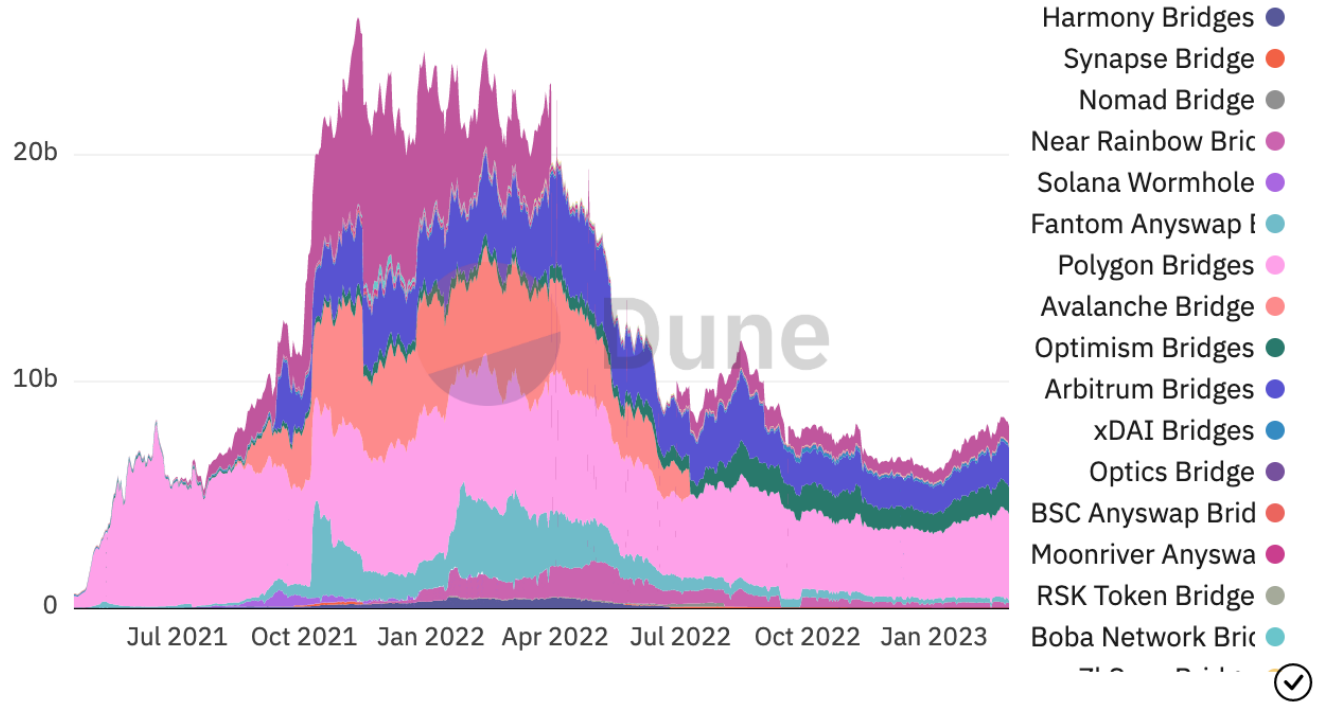
L2 Statistics





Ethereum bridges TVL over time 📈

@eliasimos



Transaction compression and costs

For zk rollups it is expensive to verify the validity proof.
For STARKs, this costs ~5 million gas, which when aggregated is about 384 gas cost per transaction.
Due to compression techniques, the calldata cost is actually only 86 gas.
This is further reduced by the calldata EIP4488 to 16.1 gas.

The batch cost is poly-log, so if activity increases 100x, the batch costs will decrease to only 4–5 gas per transaction, in which case the calldata reduction would have a huge impact.

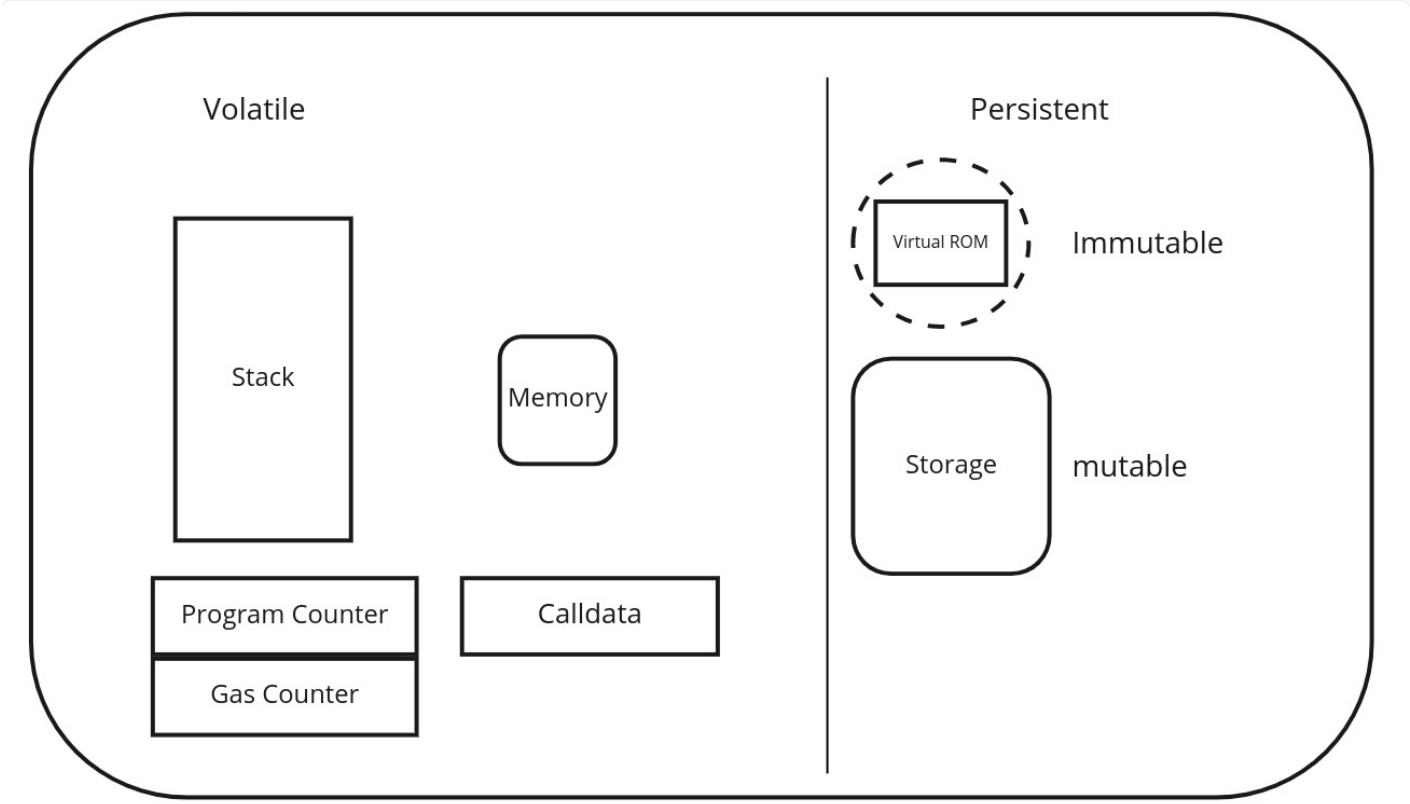
At 100x the TPS today on dYdX, the total on-chain cost will reduce to only 21 gas
At this point, the bottleneck becomes prover costs for the rollup as much as on-chain gas fees, see Polygon Zero and recursive proofs

Approaches to zkRollups on Ethereum

1. Building application-specific circuit (although this can be fairly generic as in Starknet)
2. Building a universal “EVM” circuit for smart contract execution

---

zkEVM Solutions in general



The opcode of the EVM needs to interact with Stack, Memory, and Storage during execution. There should also be some contexts, such as gas/program counter, etc. Stack is only used for Stack access, and Memory and Storage can be accessed randomly.

AppliedZKP zkEVM

AppliedZKP divides proofs into two types:

1. State proof, used to check the correctness of the state transition in Stack/Memory/Storage.
2. EVM proof, used to check that the correct opcode is used at the correct time, the correctness of the opcode itself, the validity of the opcode, and all the abnormal conditions (such as out_of_gas) that may be encountered during the execution of the opcode.

EVM processing

In general the EVM will

1. Read elements from stack, memory or storage
2. Perform some computation on those elements
3. Write back results to stack, memory or storage

So our circuit has to model / prove this process, in particular

- The bytecode is correctly loaded from persistent storage
- The opcodes in the bytecode are executed in sequence
- Each opcode is executed correctly (following the above 3 steps)

Design challenges in designing a zkEVM

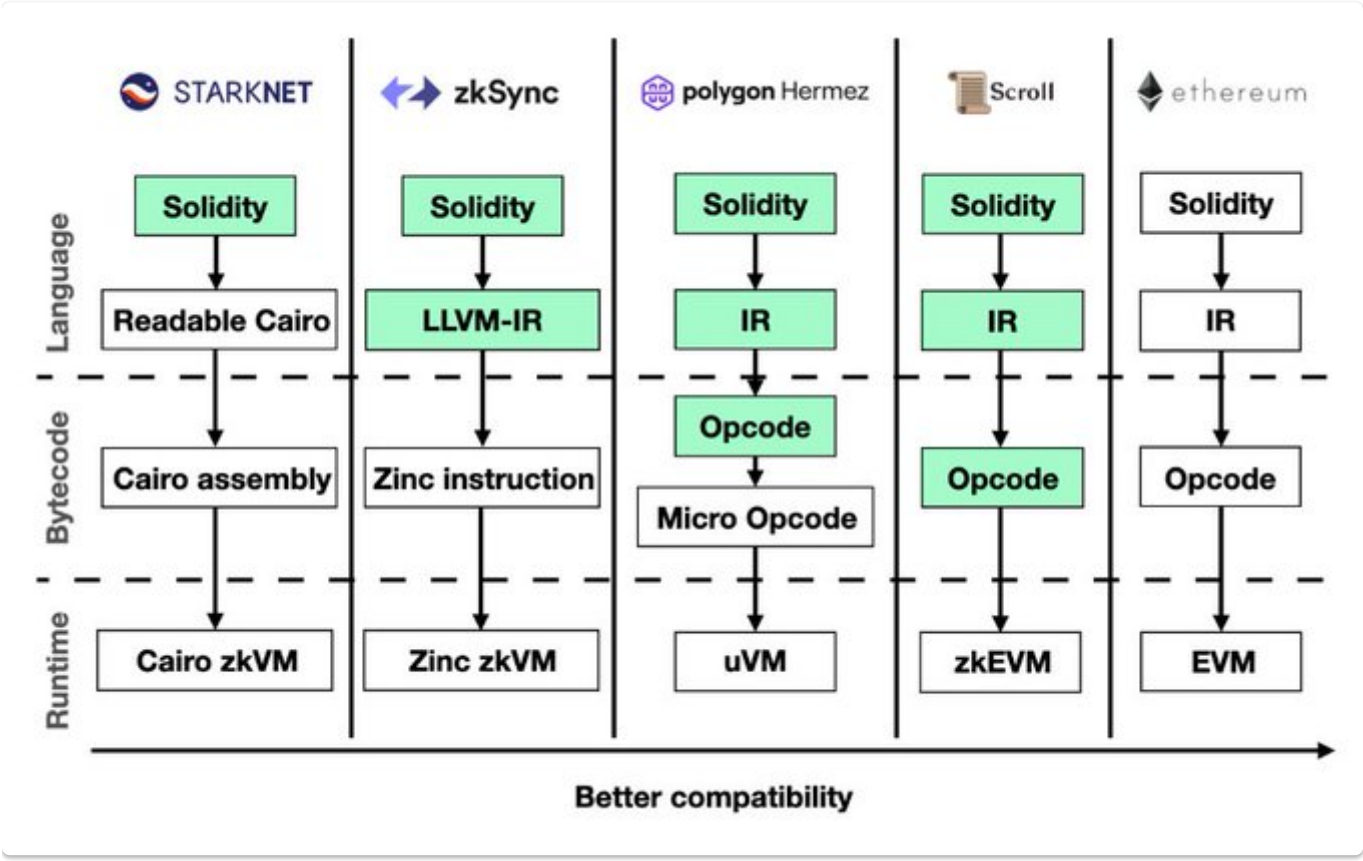
1. We are constrained by the cryptography (curves, hash functions) available on Ethereum.
2. The EVM is stack based rather than register based
3. The EVM has a 256 bit word (not a natural field element size)
4. EVM storage uses keccak and Merkle Patricia trees, which are not zkp friendly
5. We need to model the whole EVM to do a simple op code.

zkEVM taxonomy

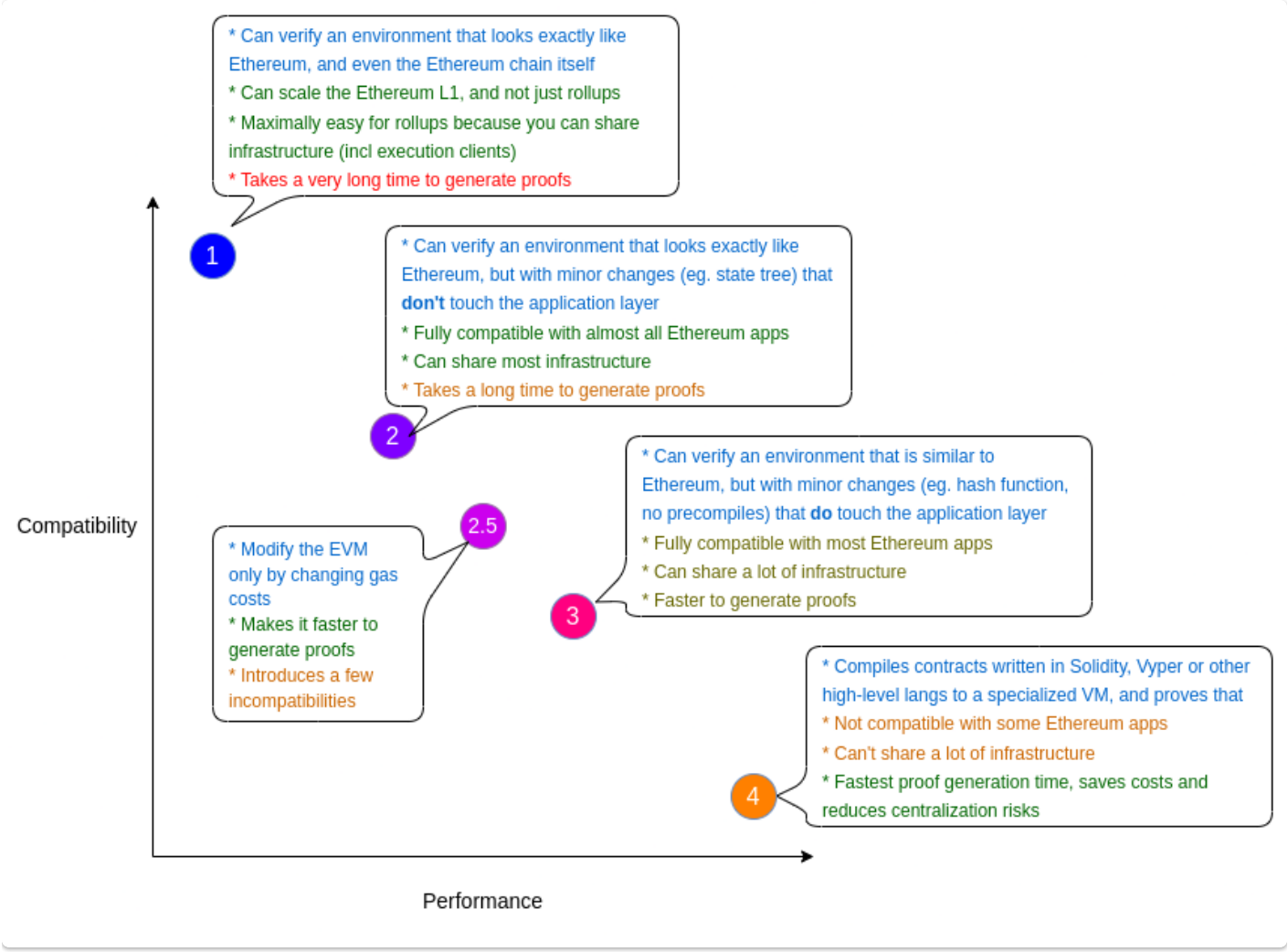
From article and article

Approaches to zkRollups on Ethereum

1. Building application-specific circuit (although this can be fairly generic as in Starknet)
2. Building a universal “EVM” circuit for smart contract execution



See Vitalik article



Type 1 (fully Ethereum-equivalent)

See zkEVM research team

Type 2 (fully EVM-equivalent)

(not quite Ethereum-equivalent)
Scroll
Hermez

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

Type 3 (almost EVM-equivalent)

Scroll and Hermez in their current form

Type 4 (high-level-language equivalent)

ZKSync
Starknet + Warp

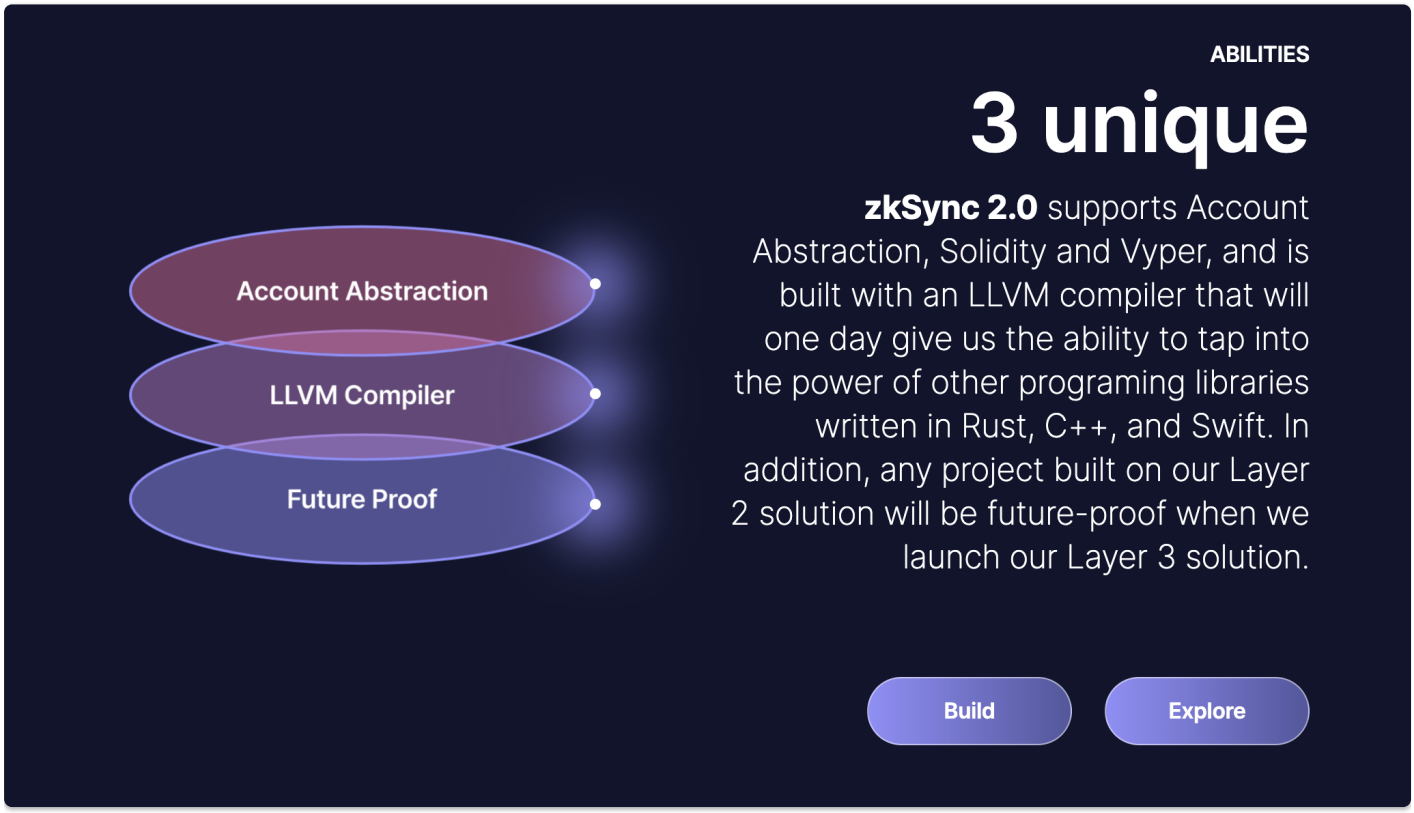
Also see
The ZK-EVM Community Edition (bootstrapped by community contributors

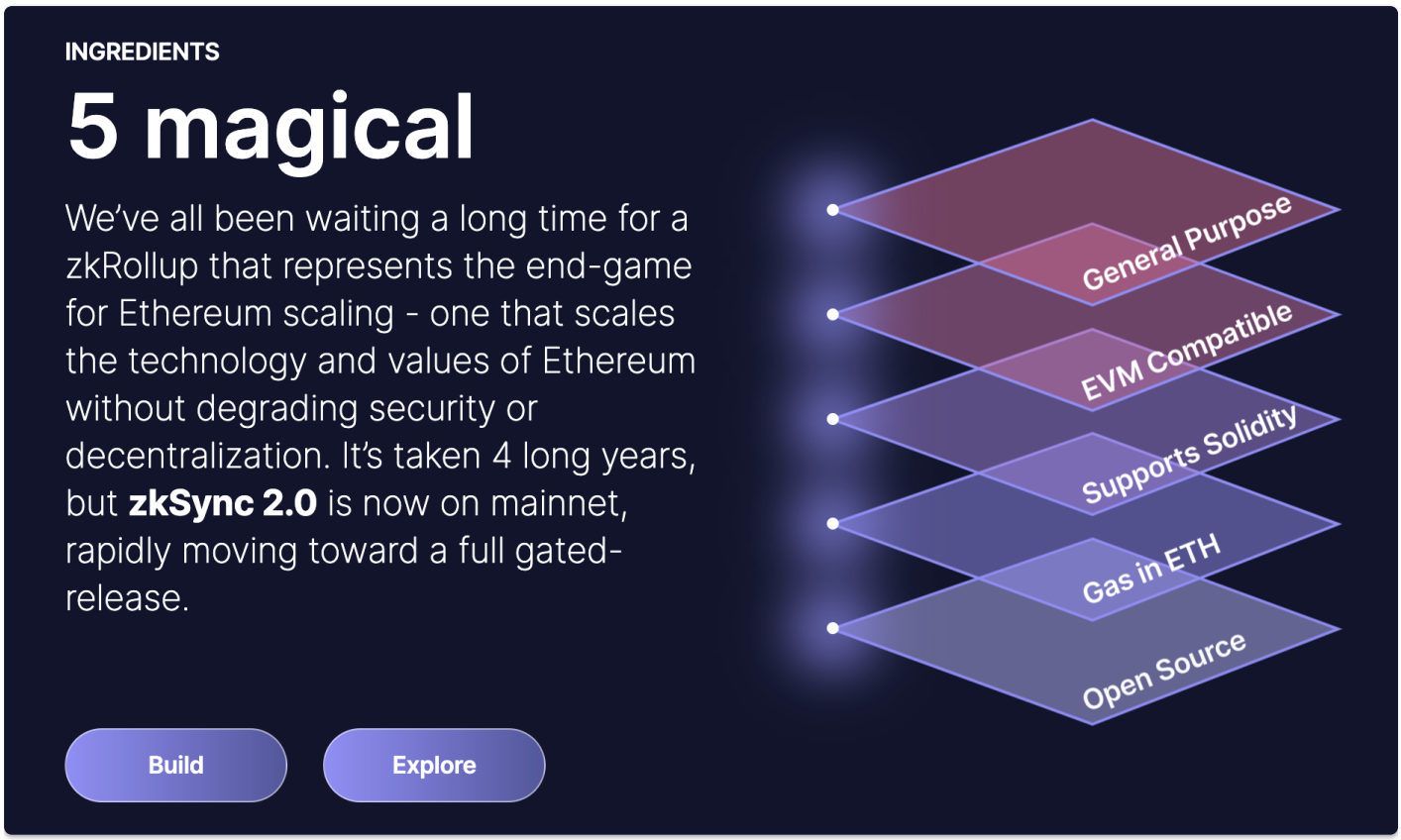
> including Privacy and Scaling Explorations, the Scroll team, Taiko and others) is a Tier 1 ZK-EVM.

zkSync

Resources

[Web]
[Twitter]
[Medium]
[Docs]





Overview

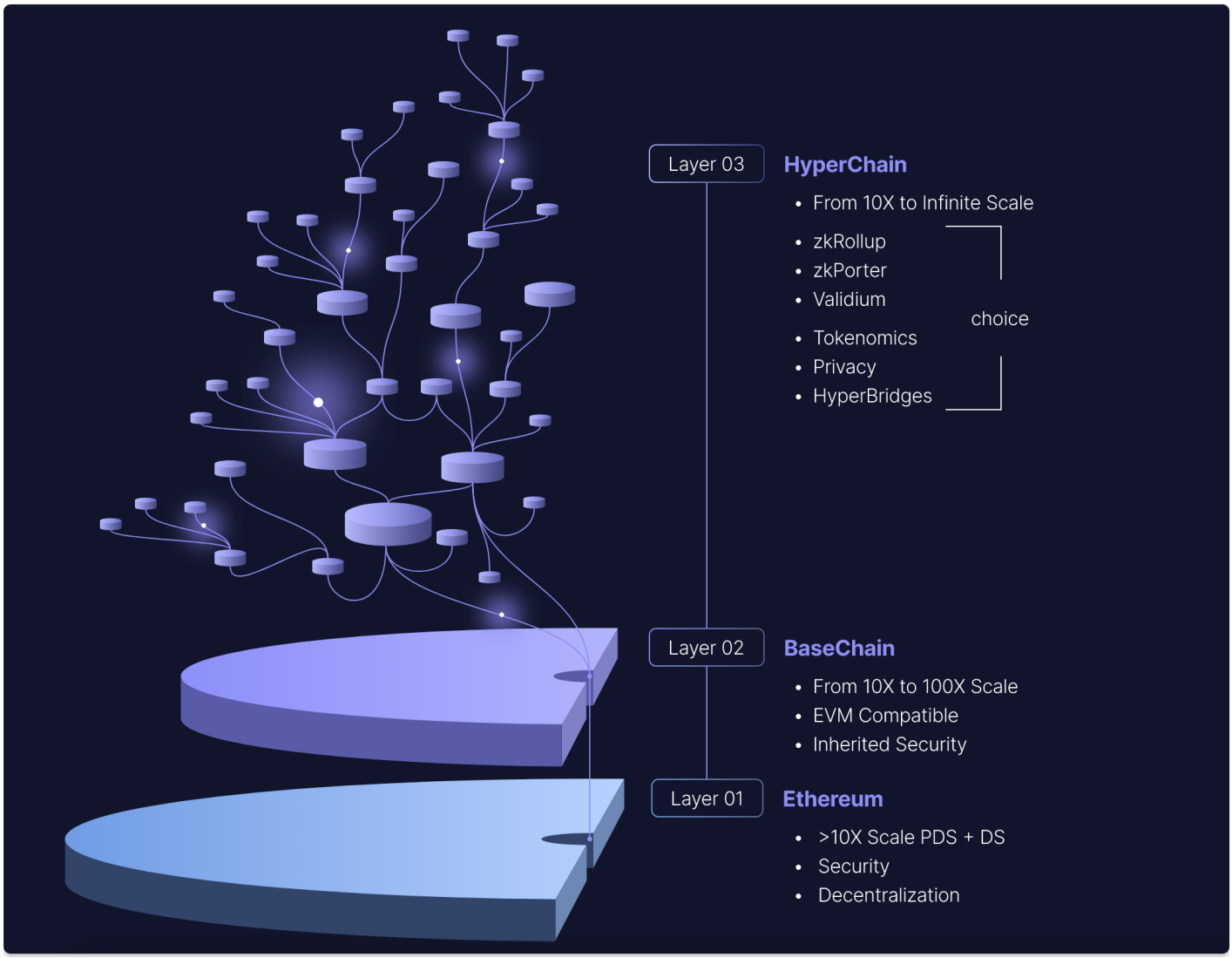
zkSync is built on ZK Rollup architecture. ZK Rollup is an L2 scaling solution in which all funds are held by a smart contract on the mainchain, while computation and storage are performed off-chain.
For every Rollup block, a state transition zero-knowledge proof is generated and verified by the mainchain contract.
This SNARK includes the proof of the validity of every single transaction in the Rollup block. Additionally, the public data update for every block is published over the mainchain network in the cheap calldata.

Hyperscaling

Hyperchains are fractal-like instances of zkEVM running in parallel and with the common settlement on the L1 mainnet.

The Basechain is the main Hyperchain instance of zkSync Era (the L2 instance). It serves as the default computation layer for generic smart contracts and as a settlement layer for all other Hyperchains (L3 and above).

The Basechain is not special in any particular way except that it settles its blocks directly on the L1


Layer 03
HyperChain
From 10X to Infinite Scale
zkRollup
zkPorter
Validium
choice
Tokenomics
Privacy
HyperBridges
Layer 02
BaseChain
From 10X to 100X Scale
EVM Compatible
Inherited Security
Layer 01
Ethereum
>10X Scale PDS + DS
Security
Decentralization

Sequencing transactions

Different modes are available

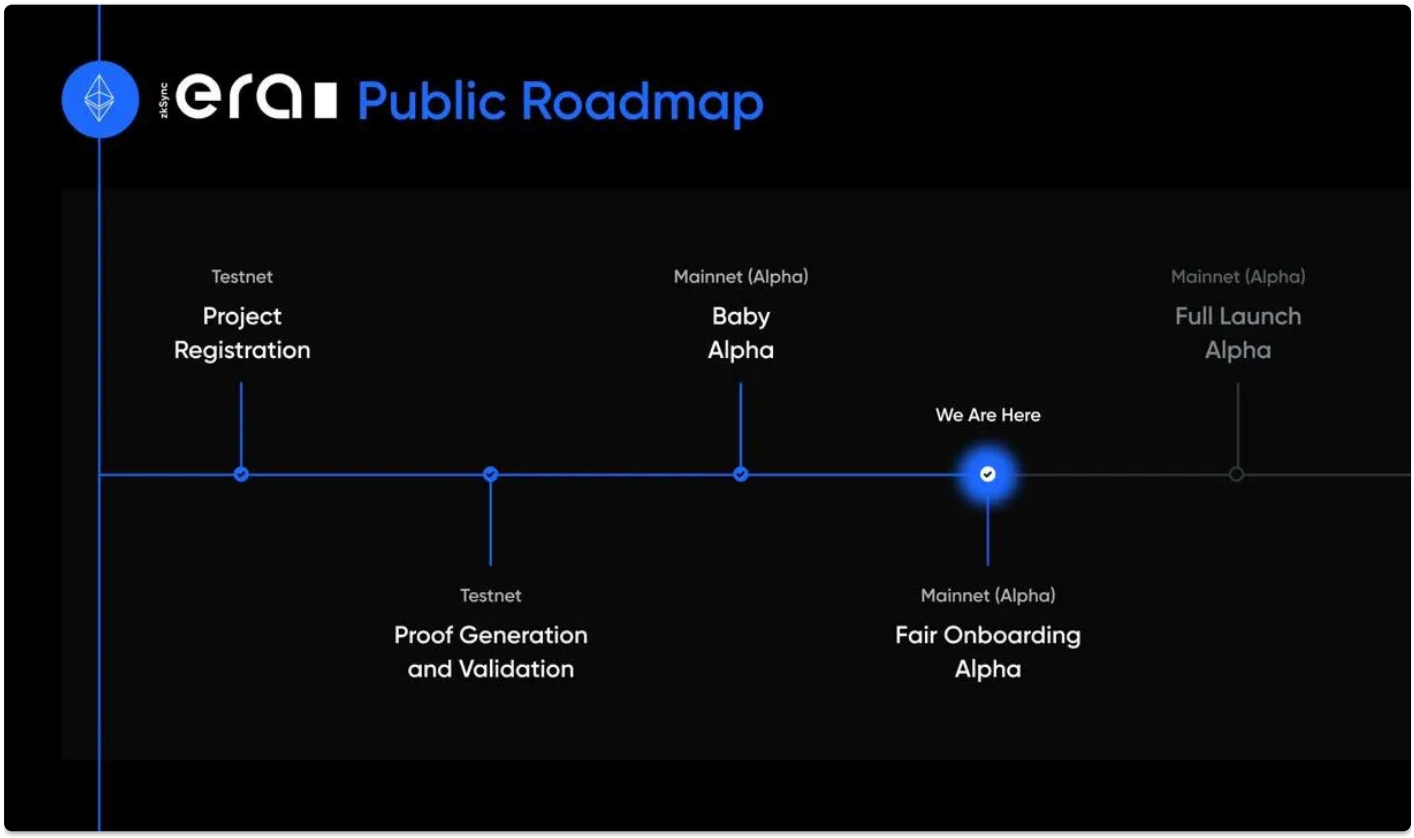
Centralized sequencer - In this mode, there will be a single centralized operator with a conventional REST API to accept transactions from users.

Decentralized sequencer - In this mode, a Hyperchain will coordinate on what transactions are included in a block using a consensus algorithm.
Any algorithm is possible, such as Tendermint or HotStuff with permissionless dPoS.

Priority queue - The goal of the priority queue is to provide a censorship-resistant way to interact with zkSync in case the operator becomes malicious or unavailable. all transactions can be submitted in batches via the priority queue from an underlying L2 or even L1 chain.

zkSync Roadmap

zkSync 2.0 is now zkSync Era



From blog post

We've arrived at Fair Onboarding Alpha, where we welcome projects that have registered to launch on zkSync Era. Fair onboarding is designed to be equitable, so we are welcoming registered projects at the same time, giving all teams an equal opportunity to deploy.

During Fair Onboarding Alpha, zkSync will

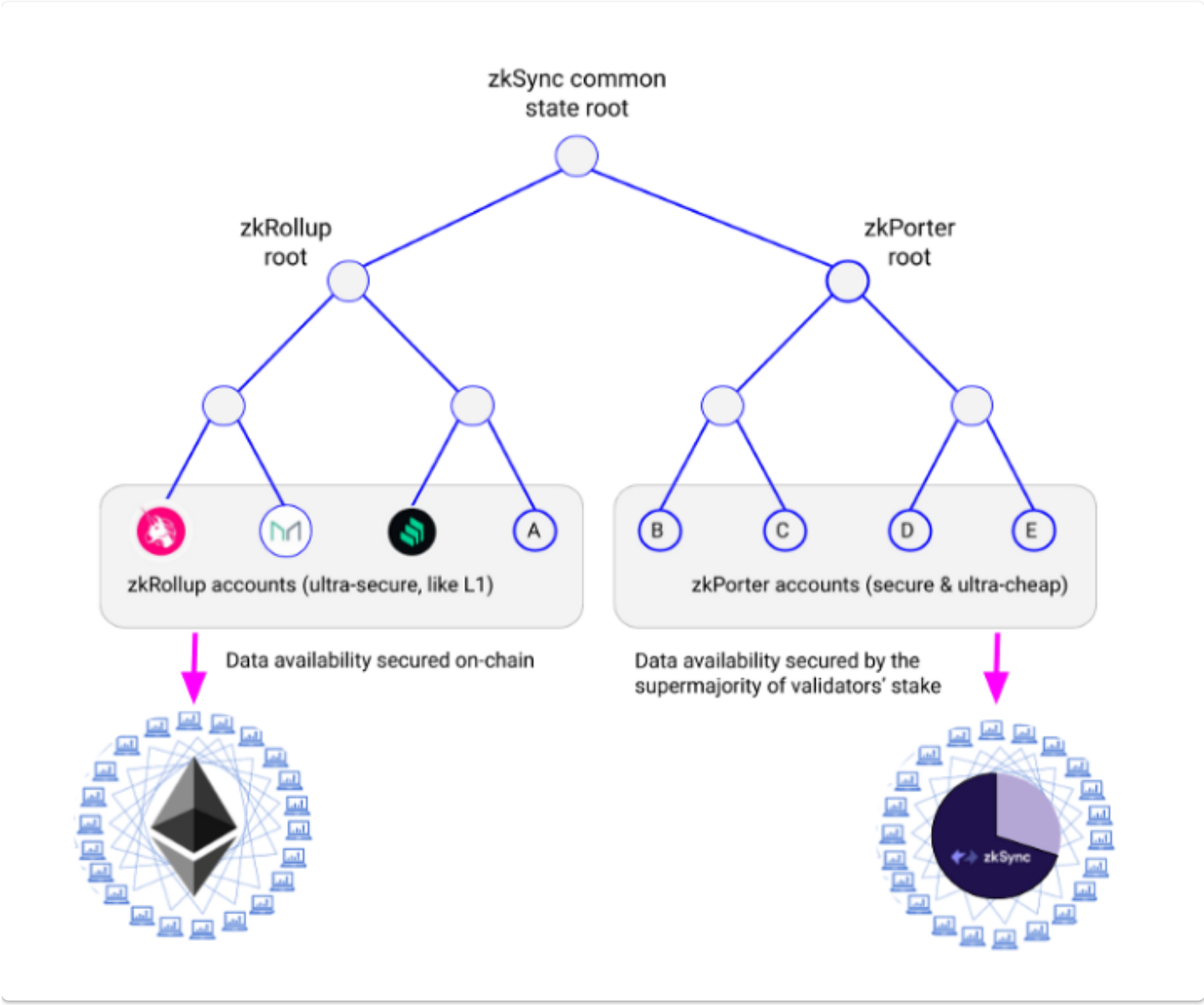
- allow registered projects to deploy and test their dapps on zkSync Era

- enable token bridging, with limited use for testing and deployment purposes
- continue running security audits, contests, and bug bounty programs
- improve developer tools, including plugins, documentation, tutorials and FAQs.

Data availability options on zkSync

- zkRollup - The default policy
  The values of every changed storage slot at the end of the block must be published as calldata on L1.
  Note that it is the end state change that is published, if there were multiple transactions contributing to the change, then the cost can be amortised.
  zkPorter - off chain data availability, see this article

contracts and accounts on the zkRollup side will be able to seamlessly interact with accounts on the zkPorter side.



From the article
"Uniswap deploys their smart contract on the zkRollup side, and retail users on a zkPorter account can swap for <$0.03 in fees.
The overwhelming majority of rollup fees are due to the costs of publishing data on Ethereum. zkPorter accounts can make thousands of swaps on the Uniswap contract, but only a single update needs to be published to Ethereum."

Underlying Cryptography

"Screenshot 2022-03-09 at 21.53.17.png" is not created yet. Click to create.

zkSync uses PLONK with custom gates and lookup tables (most commonly referred to as UltraPLONK) and Ethereum's BN-254 curve.

zkSync Infrastructure

zkSync operates several pieces of infrastructure on top of Ethereum. All infrastructure is currently live and operational, including the zkEVM.

- Full Node
  - Executes zkEVM bytecode using the virtual machine
  - Filters incorrect transactions
  - Executes mempool transactions
  - Builds blocks
- Prover
  - Generates ZK proofs from block witnesses
  - provides an interface for parallel proof generation
  - Scalable (can increase # of provers depending on demand)
- Interactor
  - The link between L1 Ethereum and L2 zkSync
  - Calculates transaction fees
    - Fees depend on token prices, proof generation, and L1 gas costs
- Paranoid Monitor
  - Monitors infrastructure and notifies Matter Labs if incidents occur

zkSync Ecosystem

https://ecosystem.zksync.io/

Faucet for test net

New Block explorer

Implementation code on L1 at
0xd61dFf4b146e8e6bDCDad5C48e72D0bA85D94DbC

Block proving contract

```
/// @notice Blocks commitment verification.
/// @notice Only verifies block commitments without any other
processing
function proveBlocks(StoredBlockInfo[] memory
_committedBlocks, ProofInput memory _proof) external
nonReentrant {

requireActive();
uint32 currentTotalBlocksProven = totalBlocksProven;
for (uint256 i = 0; i < _committedBlocks.length; ++i) {

        require(hashStoredBlockInfo(_committedBlocks[i]) ==
storedBlockHashes[currentTotalBlocksProven + 1], "o1");

        ++currentTotalBlocksProven;
        require(_proof.commitments[i] & INPUT_MASK ==
uint256(_committedBlocks[i].commitment) & INPUT_MASK, "o"); //
incorrect block commitment in proof

}
bool success =
verifier.verifyAggregatedBlockProof(
        _proof.recursiveInput,
        _proof.proof,
        _proof.vkIndexes,
        _proof.commitments,
        _proof.subproofsLimbs
);

require(success, "p"); // Aggregated proof verification fail
```

```
require(currentTotalBlocksProven <= totalBlocksCommitted,
"q");
totalBlocksProven = currentTotalBlocksProven;

}
```

Polygon Products

See this guide
Strategy article
Recent paper on efficient zk proofs for Keccak

Polygon Zero

zkRollup solutions have a bottleneck in the time in takes to generate a proof.

Polygon Zero attempts to solve this with "recursive proofs", based on Plonky2.

Polygon Zero generates proofs simultaneously for every transaction in the batch. Theses are then aggregated into a single proof which is submitted on the Ethereum network.

This approach significantly reduces the effort it takes to generate reliable validity proofs. Polygon Zero's Plonky2 can generate a recursive proof in 0.17 seconds.

Polygon Hermez

Polygon (Hermez) team are working on a protocol Proof of Efficiency
This involves 2 permissionless roles :

- Sequencer
- Aggregator

From article

Sequencers collect transactions from users on the rollup, then select and pre-process new batches of this Layer 2 data. Finally, they send transactions to Layer 1 to be recorded. Sequencers also deposit a fee in $MATIC token as an incentive for Aggregators to include the batch in a zero knowledge proof.

Hermez 2.0

Hermez current functionality is limited to token transfers and atomic swaps, so there are plans to introduce Hermez 2.0 which will have EVM compatibility.

Polygon Miden

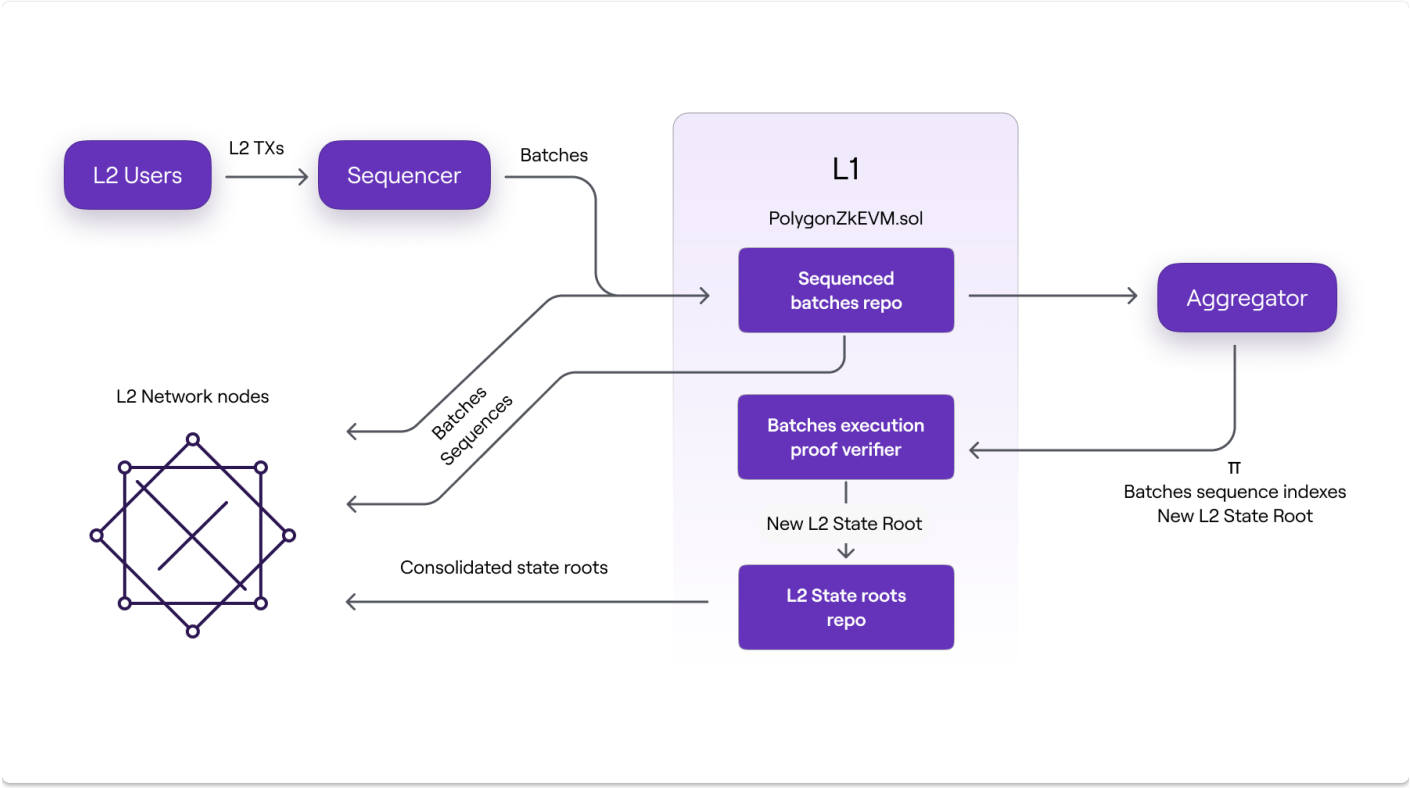
Polygon Miden is a general-purpose, STARK-based ZK rollup with EVM compatibility.
This will differ from Starknet in that it will be EVM compatible, so it should run Solidity contracts.
From their documentation
"Polygon Miden can process up to 5,000 transactions in a single block, with new blocks produced every five seconds. Although this ZK rollup exists as a prototype for now, it is expected to boost throughput to over 1,000 transactions per second (TPS) at launch."

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

Transaction / State Flow



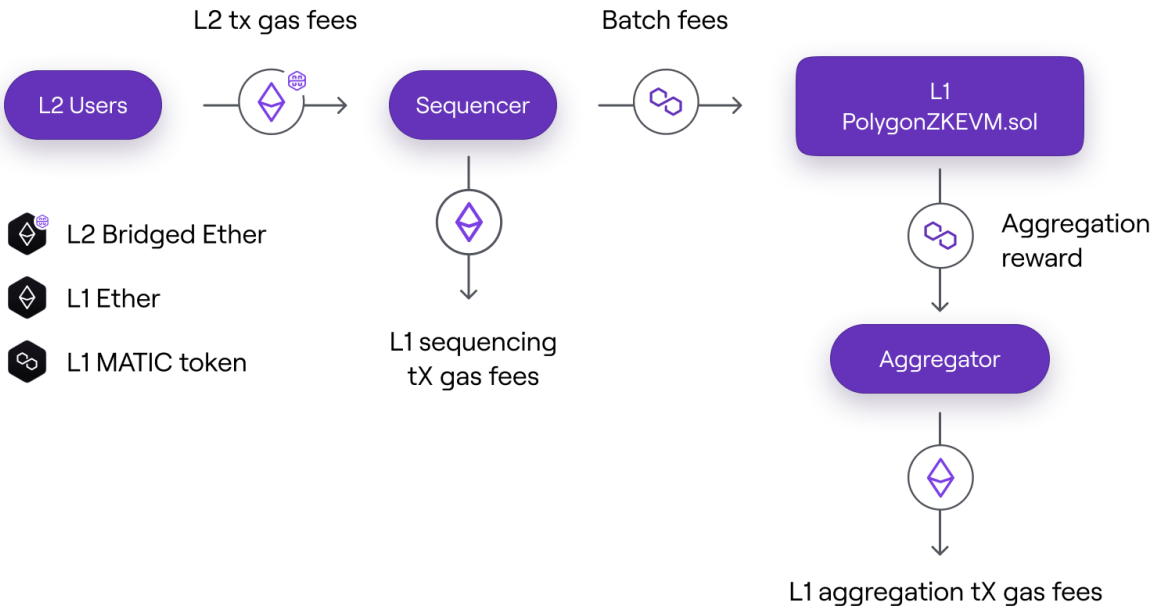
From the docs
"The Trusted Sequencer reads transactions from the pool and decides whether to discard them or order and execute them. Transactions that have been executed are added to a transaction batch, and the Sequencer's local L2 State is updated.

Once a transaction is added to the L2 State, it is broadcast to all other zkEVM nodes via a broadcast service. It is worth noting that by relying on the Trusted Sequencer, we can achieve fast transaction finality (faster than in L1). However, the resulting L2 State will be in a trusted state until the batch is committed in the Consensus Contract."

VERIFICATION ON LAYER 1

Users will typically interact with trusted L2 State. However, due to certain protocol characteristics, the verification process for L2 transactions (on Layer 1 to enable withdrawals) can take a long time, typically around 30 minutes but up to 2 weeks in rare cases.
As a result, users should be mindful of the potential risks associated with high-value transactions, particularly those that cannot be reversed, such as off-ramps, over-the-counter transactions, and alternative bridges.

Fees



The native currency used in L2 is `Bridged Ether`, which originates from L1. This is the currency that is used to pay L2 transaction fees. It can be transferred at a 1:1 exchange ratio from L1 to L2 and vice versa.

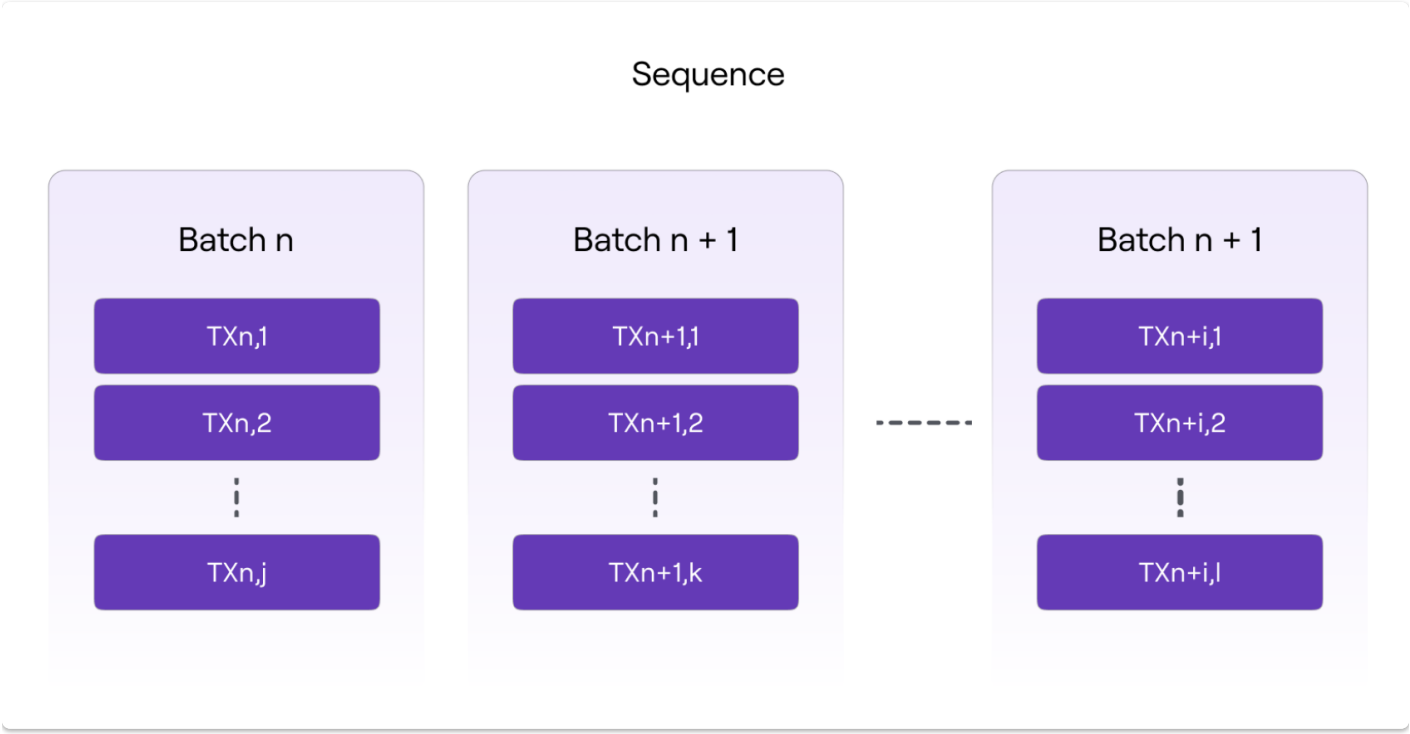
The Sequencer earns the transaction fees paid by L2 users for submitting transactions, and thus gets paid directly in `Bridged Ether`. The amount of fees paid depends on the gas price, which is set by users based on how much they are willing to pay for the execution of their transactions.

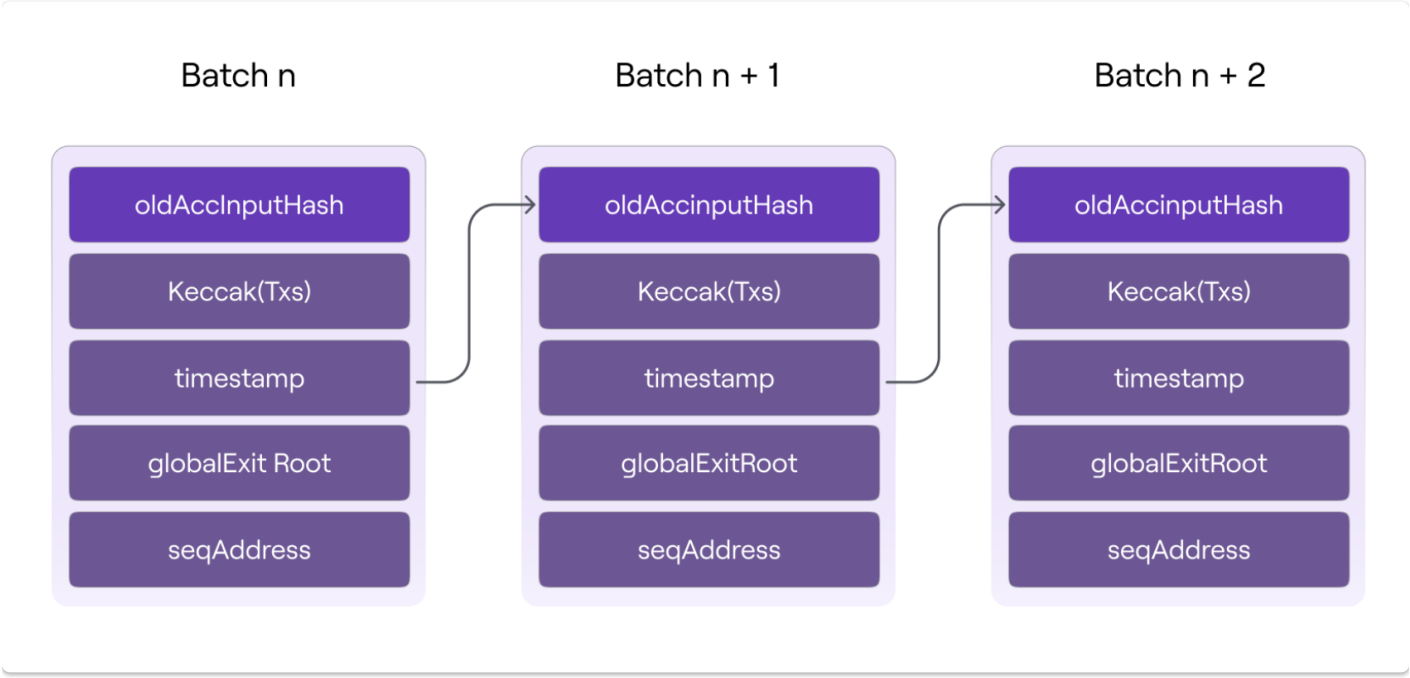
To incentivize the Aggregator for each batch sequenced, the Sequencer must lock a number of MATIC tokens in the L1 `PolygonZkEVM.sol` Contract proportional to the number of batches in the sequence. The number of MATIC tokens locked per batch sequenced is saved in the variable `batchFee`.

The Sequencer prioritizes transactions with higher gas prices. Furthermore, there is a threshold below which it is unprofitable for the Sequencer to execute transactions because the fees earned from L2 users are less than the fees paid for sequencing fees (plus L1 sequencing transaction fee).

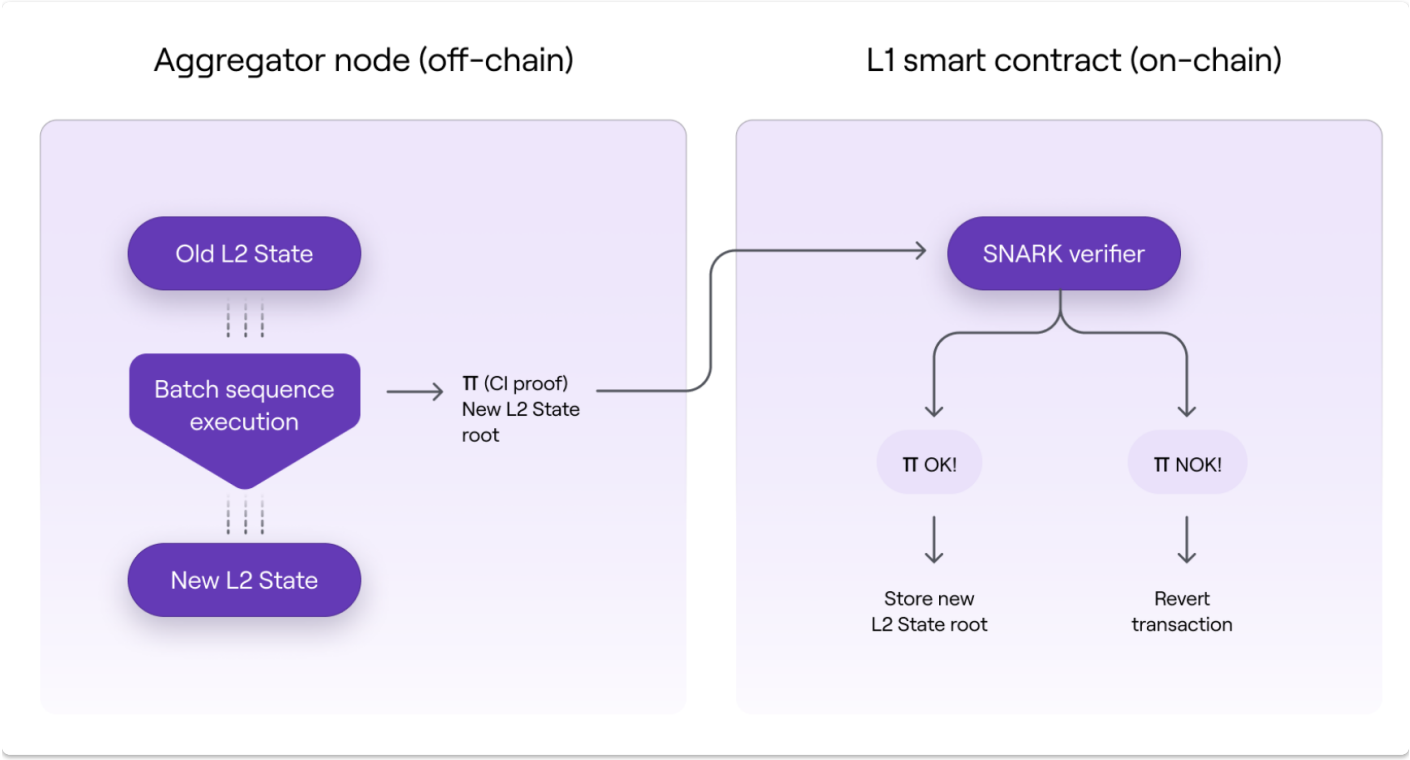
The aggregator also receives an aggregation reward.

Batch Sequencing

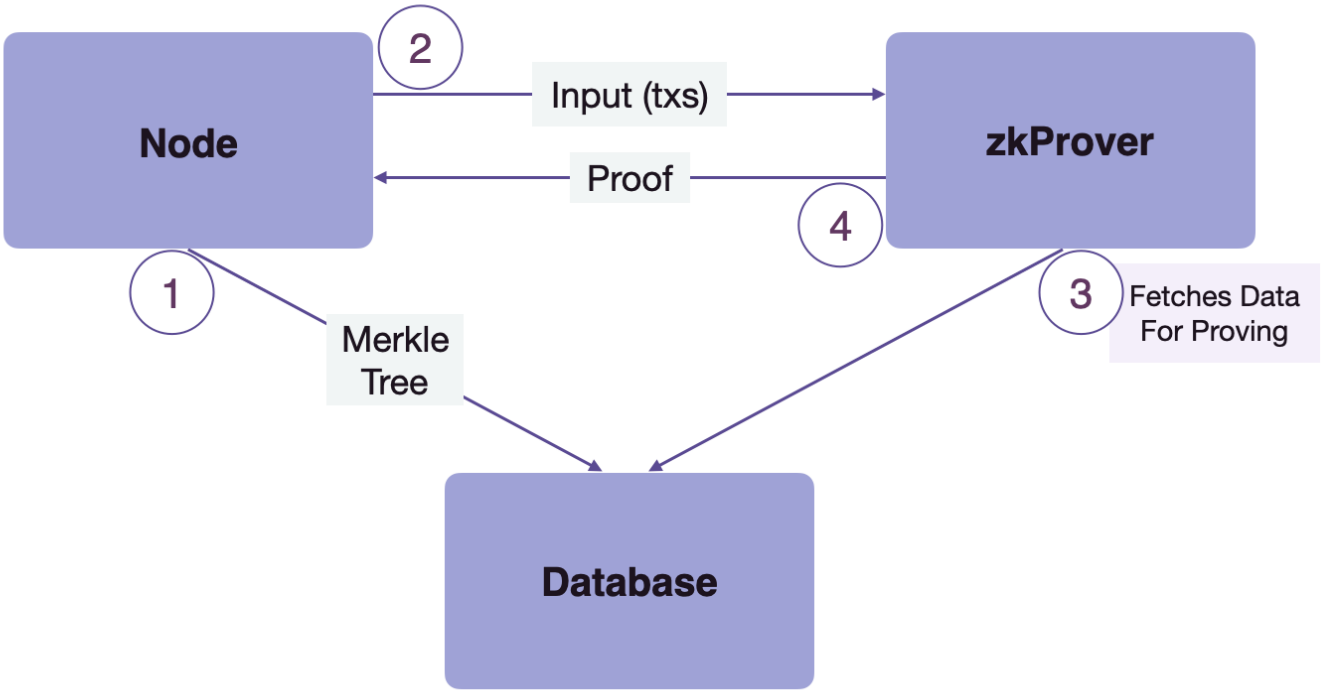




Batch Aggregation



Proof process



All the rules for a transaction to be valid are implemented and enforced in the zkProver.

In summary the process is

- The Node sends the content of Merkle trees to the Database to be stored there
- The Node then sends the input transactions to the zkProver
- The zkProver accesses the Database and fetches the info needed to produce verifiable proofs of the transactions sent by the Node. This information consists of the Merkle roots, the keys and hashes of relevant siblings, and more
- The zkProver then generates the proofs of transactions, and sends these proofs back to the Node

See Docs

Polynomial Identity Language

See Docs
From documentation
This is a novel domain-specific language useful for defining state machines.

One of the main peculiarities of PIL is its modularity, which allows programmers to define parametrizable state machines, called namespaces, which can be instantiated from larger state machines. Building state machines in a modular manner makes it easier to test, review, audit and formally verify even large and complex state machines. In this regard, by using PIL, developers can create their own custom namespaces or instantiate namespaces from some public library.

Some of the keys features of PIL are;

- Providing namespaces for naming the essential parts that constitute state machines,
- Denoting whether the polynomials are committed or constant,
- Expressing polynomial relations, including identities and lookup arguments and
- Specifying the type of a polynomial, such as bool or u32u32.

Some example PIL

```
  namespace Multiplier(2**10);

// Constant Polynomials
pol constant RESET;

// Committed Polynomials
pol commit freeIn;
pol commit out;

// Constraints
out' = RESET*freeIn + (1-RESET)*(out*freeIn);
```

Computational Model

From documentation

Many other domain-specific languages (DSL) or toolstacks, such as Circom or Halo2, focus on the abstraction of a particular computational model, such as an arithmetic circuit.

However, recent proof systems such as STARKs have shown that arithmetic circuits might not be the best computational models for all use cases. For example, given a complete programming language, computing a valid proof for a circuit satisfiability problem may result in long proving times due to the overhead of re-used logic.

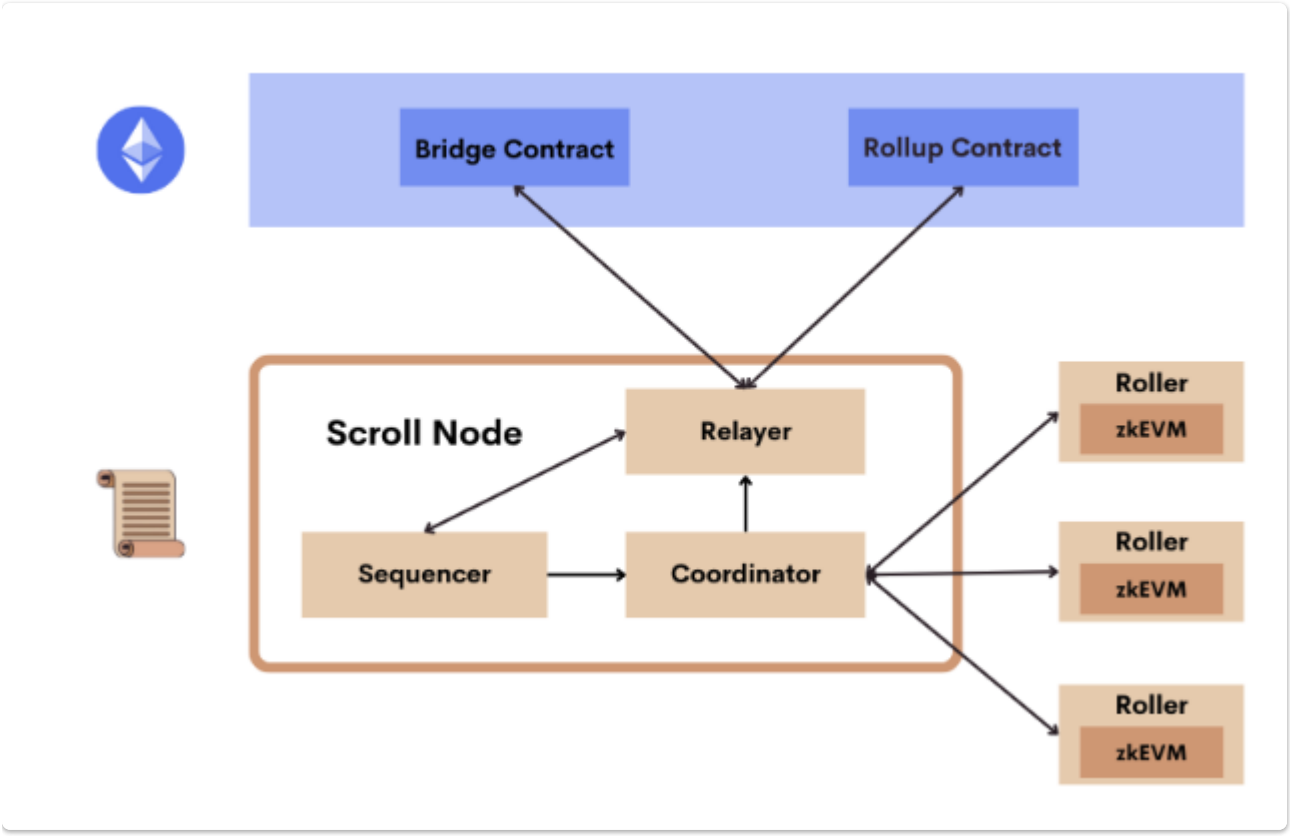
By opting for deployment of programs with their low-level programming, shorter proving times are attainable, especially with the advent of proof/verification-aiding languages such as PIL. Hence the decision to adopt state machines as the best computational model for the Polygon zkEVM.

Scroll

Scroll 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.

Scroll Architecture



Components

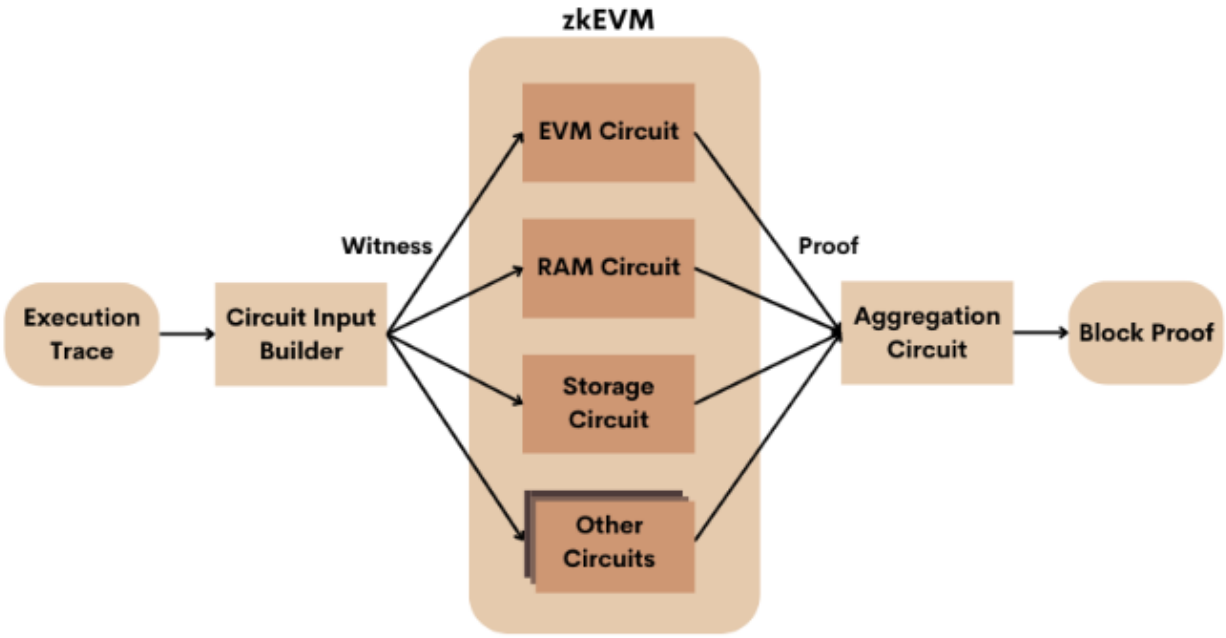
The Sequencer provides a JSON-RPC interface and accepts L2 transactions. Every few seconds, it retrieves a batch of transactions from the L2 mempool and executes them to generate a new L2 block and a new state root.

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

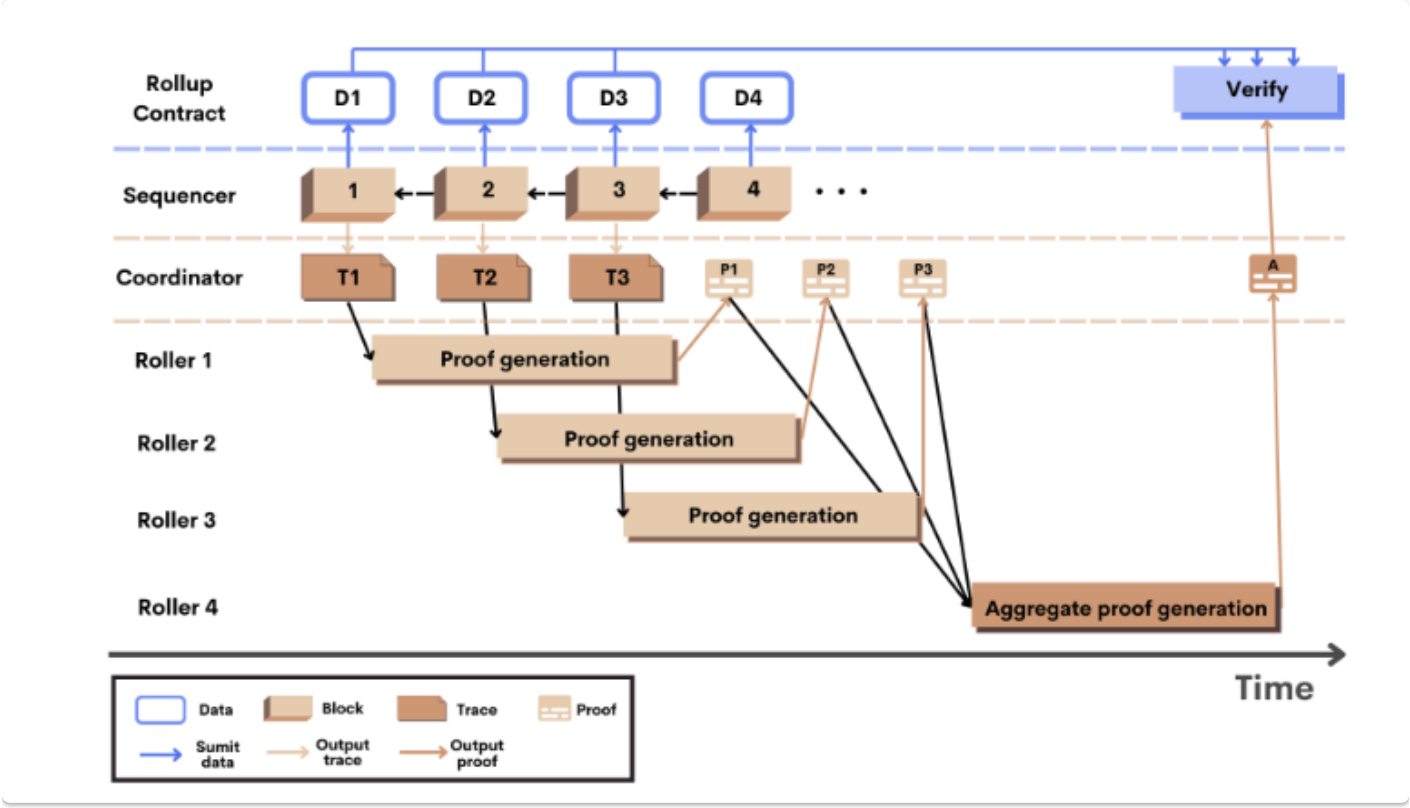


The Rollers serve as provers in the network that are responsible for generating validity proofs for the zkRollup

- 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:

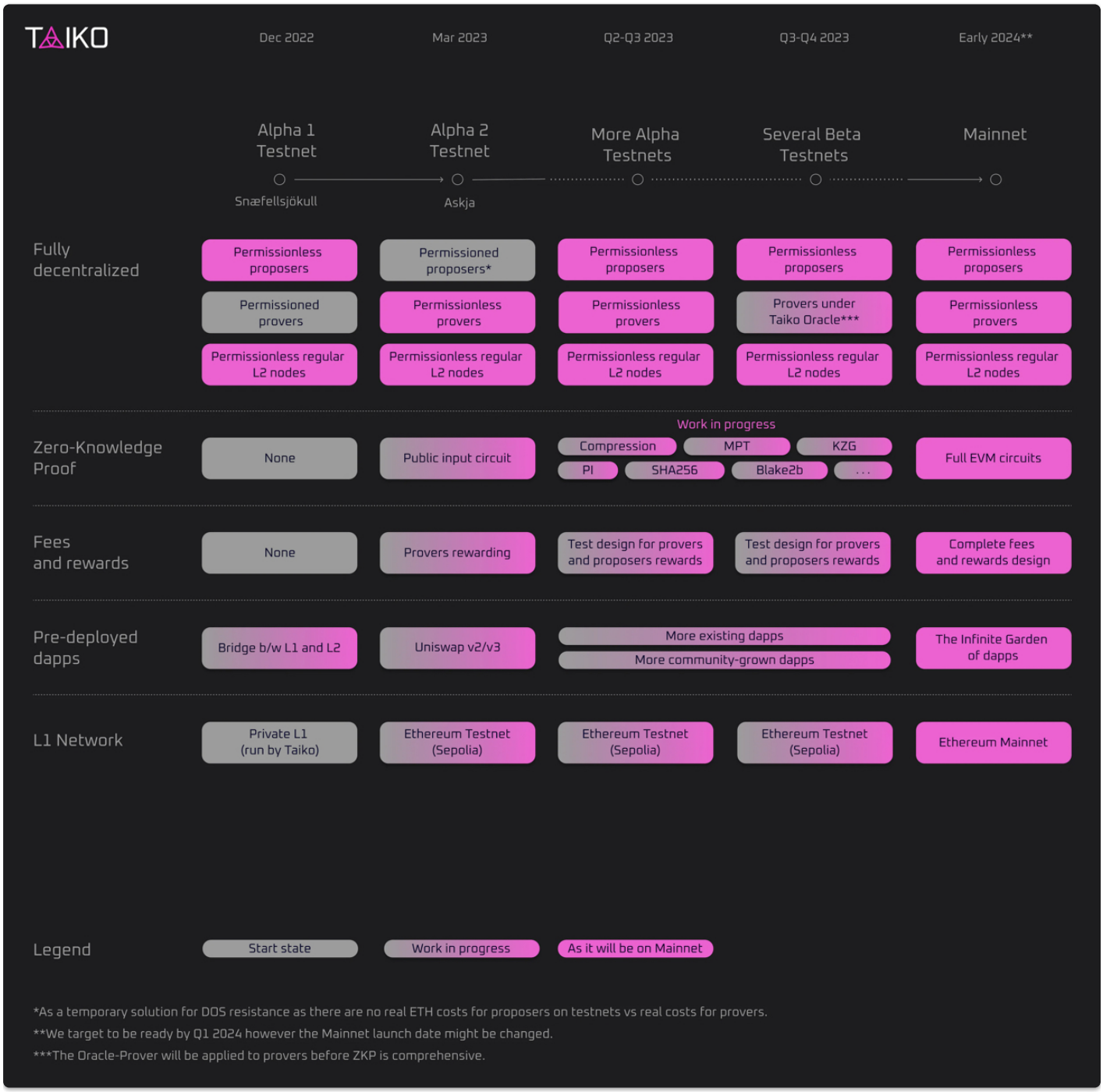
1. The Sequencer generates a sequence of blocks. For the *i*-th block, the Sequencer generates an execution trace *T* and sends it to the Coordinator. Meanwhile, it also submits the transaction data *D* as calldata to the Rollup contract on Ethereum for data availability and the resulting state roots 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.

Taiko

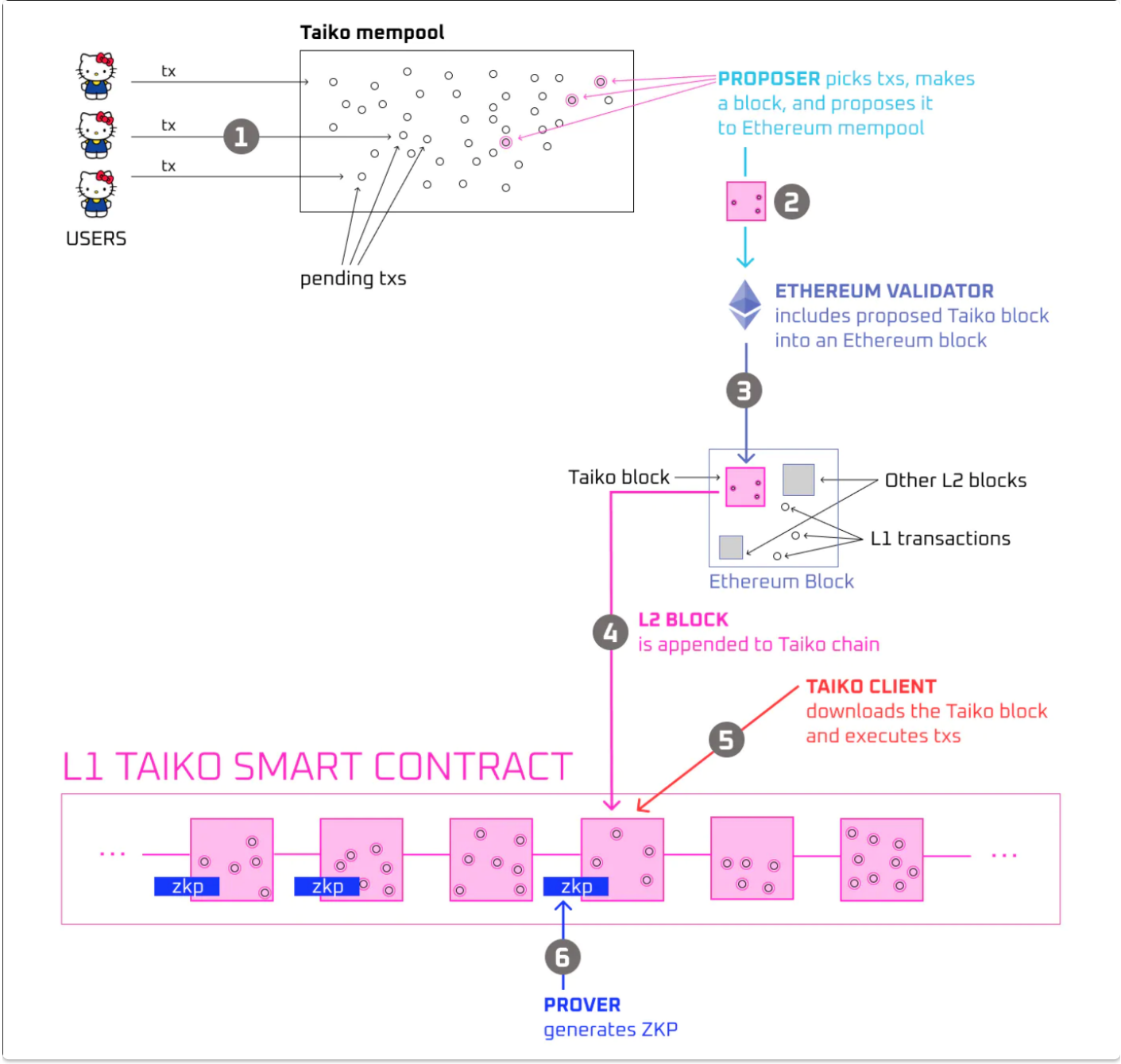
See Docs

"Taiko is a decentralized, 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"

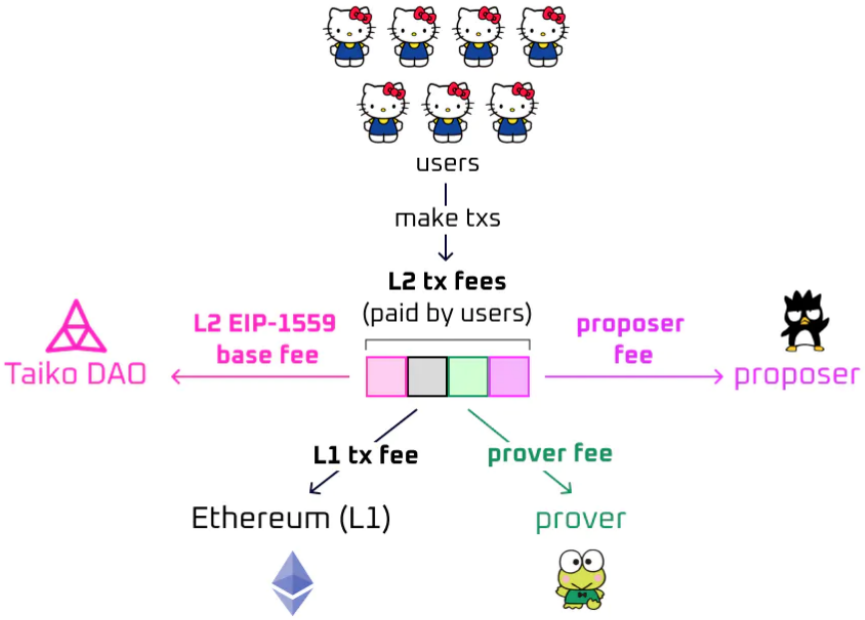
Roadmap



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.

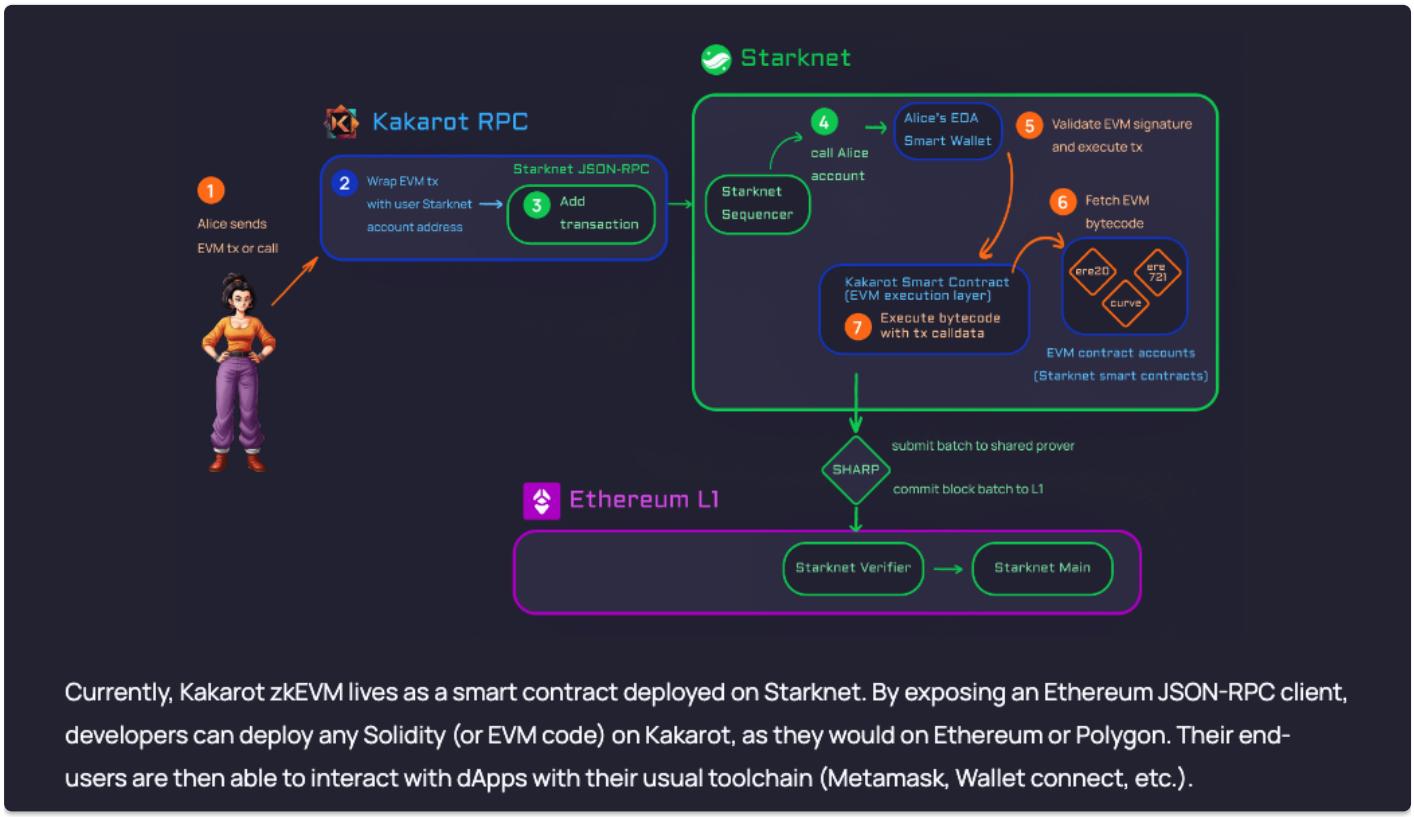
Provers can submit proofs for any block they like.
Multiple provers may work in parallel to generate ZKPs for one or multiple Taiko blocks, but only the first proof will be accepted for any given block transition (fork choice). The address receiving the reward is coupled with the proof, preventing it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

Oracle Prover

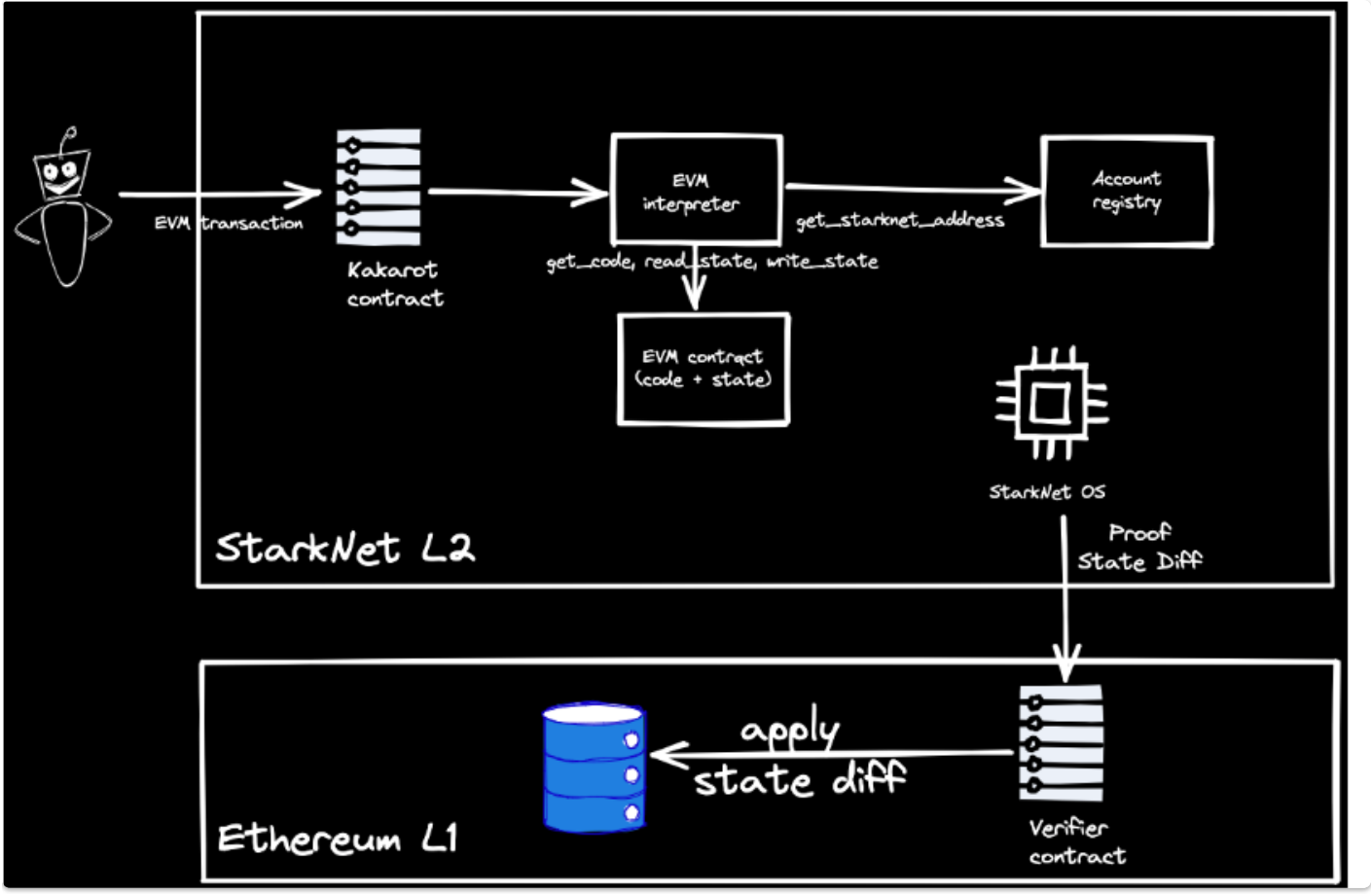
Currently Taiko also runs an 'Oracle prover', (though proof checker may be a better name) to mediate if there are any problems with proof generation.
The oracle prover checks all state transitions (by running a node to run over all transactions in the block) and may override existing state transitions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