Rollup Stacks

Disclaimer: Espresso has launched Mainnet 0, our first production release. While the network is now live and being used in production, we are actively developing and upgrading both the network and its underlying technology. We welcome rollup developers to integrate with Espresso, but please note that this documentation may be updated frequently and integration processes may evolve as we enhance the system.

The Espresso Network is a global confirmation layer that enables faster bridging, decentralized and shared sequencing, and low-cost data availability for rollups. As a rollup developer, you can permissionlessly integrate your own rollup with Espresso by making certain code changes to interface with Espresso. Espresso is also working with several rollup stacks and rollup-as-a-service options to provide easy deployment options.

This section describes the general architecture of an integration between Espresso and a typical rollup. It covers integrations for both ZK rollups and optimistic rollups. There may be some specific details which vary between individual rollups. If you'd like more information on a possible integration with your specific rollup, you can contact us. As supplementary material, you can also check out the demo integration of a minimal, illustrative ZK rollup.

Integrating a ZK Rollup

This section will develop a model of the architecture of a typical ZK rollup and describe how this architecture can be adapted to use Espresso. It can serve as a guide both for adapting existing rollups to replace their current sequencer with Espresso and for designing new rollups intended to use Espresso from genesis. For a more in-depth overview please see this GitHub page.

If you've already familiarized yourself with the Espresso architecture and you just want a quick integration checklist, you can skip ahead to the summary of changes.

ZK Rollup Architecture

This section develops a model of the architecture of a ZK rollup by abstracting away most of the internal complexity, and focusing mainly on the interactions between the sequencer and other components. Thus, this model should be sufficient to guide the integration of a wide variety of ZK rollups with Espresso.

Background

In this model, a **ZK rollup** is a distributed **state machine** in which one or more agents that compute state transitions (the **executors** or **provers**) use zero-knowledge proofs to convince additional, non-computing agents (such as the end users) of the validity of state updates. The state transitions are determined by applying the deterministic function which defines the rollup to a sequence of transactions, or state transition requests, starting from a known initial state. This sequence is usually represented as a sequence of **blocks**, where each block consists of an ordered list of many transactions.

The architecture model will aim to isolate the **sequencer**, the component of the system which determines this sequence. Modeling the sequencer's interactions with the rest of the system can shed light on how to integrate with Espresso.

Components

We model the rollup as a collection of three components:

- The **sequencer** is responsible for batching transactions from various users into ordered *blocks*, and then committing to an order of those blocks. The order can be arbitrary or subject to rollup-specific constraints. The important thing is that the order is public and immutable: all rollup users at all times should agree on the relative ordering of blocks.
- The **executor and prover** is an abstract component which may in reality consist of several components. Its primary job is to execute the rollup's **state transition function**, usually some abstract machine in which transactions represent programs to execute. By executing this function, the executor computes and stores a VM **state** which is a deterministic result of the ordering of blocks produced by the sequencer.

By virtue of maintaining and storing the state, this component is equipped to perform some additional functions, including

  - Running a web server which allows users to interact with the VM state (such as an Ethereum JSON-RPC server, if the rollup VM is Ethereum-compatible).
  - Running a **ZK prover** which produces zero-knowledge proofs of the correctness of the state stored by the executor. This component proves to other, untrusting participants that the executor has correctly computed the state transition function on the sequence of blocks produced by the sequencer.

- The **rollup contract** validates ZK proofs produced by the prover and stores recent, certified **state roots** in the storage of a lower-layer blockchain (like Ethereum). This allows clients who neither trust the executor nor want to do the work of validating ZK proofs to authenticate the state of the VM, by trusting a majority of the L1 validators to have correctly verified the ZK proof. The state roots stored in the contract are usually succinct digests of the VM state, like Merkle roots, so that clients can fetch large pieces of the state from an untrusted, off-chain source and then validate them against the state root.

This is an optional component. Some L2s (so-called **sovereign rollups**) do not verify their state with any other blockchain, and instead rely on each peer either computing the correct state themselves or validating a ZK proof directly, rather than trusting the L1 validators to check the proof.

This model is designed to make it easy to understand what changes when the sequencer is replaced by Espresso. However, in a real rollup one or more of these components may be combined. For example, the sequencer and executor may be part of the same service.

Transaction Flow

The figure below shows the interactions between these components as a transaction flows through the system. The dashed arrows indicate places where more than one design is possible, and different ZK rollups may make different choices.

Generalized ZK Rollup architecture

1. The user submits a transaction. This will often be submitted to a rollup-specific service of the user's choice (such as a JSON-RPC server) before being forwarded to the sequencer. However, many rollups also allow the user to submit directly to the sequencer. In some rollups, the sequencer may even be combined with the rollup service, so that it can execute transactions before sequencing them and filter out invalid transactions.
2. The sequencer eventually includes the transaction in a block.
3. The block is sent to the rollup contract, where it is stored. This serves two purposes:
    - The contract can act as a source of truth for the order of blocks, so that if the sequencer later tries to equivocate, and report a different order to different participants, all parties can take the order saved in the contract as authoritative.

    - The layer 1 blockchain hosting the contract provides **data availability** for the rollup. Even if the sequencer goes down, users and executors/provers can read the data for each sequenced block from storage in the layer 1 blockchain, which is presumed to be highly available. This means that the sequencer cannot stop any participant from reconstructing the state of the layer 2 blockchain.

4. The executor is notified of the new block. They may get it from the rollup contract, or directly from the sequencer. In the latter case, they will authenticate the block provided by the sequencer against the ordering (or a succinct commitment to the ordering) stored in the contract.
5. The executor executes the block and updates its local copy of the VM state.
6. The prover produces a zero-knowledge proof that the new state is the result of correctly applying the state transition function to the sequenced block.
7. The prover sends the proof and the new state root to the rollup contract, which verifies the proof and stores the state root.
8. The user observes the result of their transaction by querying the server provided by the executor. If the user does not trust the executor, they can check the response against the state root stored in the contract, or they can request a zero-knowledge proof of the state root and verify it themselves.

Using Espresso

This section describes the recommended architecture for a ZK rollup integrated with Espresso.

Here we describe the architecture for a ZK rollup using Espresso for confirmations, data availability, and sequencing. Rollups may choose to use a separate sequencer (centralized or decentralized) while still using Espresso for confirmations, with a similar system architecture.

Additions, deletions, and changes are highlighted in comparison with the typical ZK rollup architecture.

Components

Generalized ZK rollup architecture, modified to use Espresso

Espresso brings a few new components into the picture:

- **HotShot** finalizes blocks produced by the Espresso Network in a few seconds. This ensures that rollup transactions are confirmed quickly and in a decentralized and secure fashion, owing to the many nodes participating in HotShot.
- **Tiramisu** provides data availability for all rollup blocks. This availability is guaranteed by the same operator set that guarantees finality, so it is reliable, and rollups have the option of using Tiramisu as their standalone data availability layer if they wish. Tiramisu is designed to provide higher throughput and lower fees than using Ethereum for data availability.
- The Light client contract runs a HotShot client on the layer 1 blockchain, verifying the finality of each block and storing a binding commitment to the order of blocks decided

by HotShot. As described in the following sections, rollup contracts on the same layer 1 can use this contract to relate their state transitions to the committed block order.

Transaction Flow

The transaction flow with Espresso is very similar to the transaction flow without. The main difference is in how confirmed blocks are disseminated. When not integrated with Espresso, rollup nodes are free to choose from where they fetch sequenced blocks: from the rollup contract on layer 1, or directly from the sequencer. Espresso, however, does not send sequenced blocks to each rollup contract: doing so would be expensive and would not meet the needs of all rollups. Instead, Espresso only sends sequenced blocks to one place—the sequencer contract—and it only sends a *commitment* to each block, at that: since Espresso provides its own data availability solution, the default is *not* to store entire blocks on the layer 1.

With this new flow, rollup nodes may choose how they get *notified* of new blocks: either by streaming them directly from Espresso or streaming events from the sequencer contract. But in either case, they must retrieve the block *contents* from the Espresso Network (see Espresso's data availability API). Furthermore, rollups that still wish to use the layer 1 for data availability are responsible for sending the necessary data to the layer 1 once it has been sequenced.

It is important to note that while the transaction flow changes in terms of how transactions get sequenced (if Espresso is used for sequencing) and how rollups consume that sequence, there is no change in what a rollup does with a block *after* it is sequenced. This means that the execution layer, which makes up the bulk of the complexity of many rollups, can be used completely unmodified with Espresso.

The following sections describe how the components of the rollup must be modified to support the updated transaction flow and retain security.

Rollup Proofs in a Shared-Sequencer World

The bulk of the changes from the basic ZK rollup architecture relate to the state transition proofs, requiring additions to both the prover and the verifier (which may be part of the rollup contract, or may be an off-chain component, in the case of sovereign rollups). The difference stems from the statement which needs to be proven. In a typical ZK rollup that only has to deal with a single, centralized sequencer, the prover proves a statement of the form:

Given a block B and an initial state root S, the result of applying the state transition function is a new state root S'.

The verifier checks both that the proof is *correct*, or internally consistent, and also that it is *relevant*: that S is the current state root and B is the next block in the sequence. Only then does it accept S' as the new state root. The on-chain verifier can easily do this check because the sequencer is sending blocks (or block commitments) directly to the rollup contract, and those blocks contain only transactions that are relevant to this particular rollup. Therefore the contract simply reads the last sequenced block directly from its own storage and compares it with B (or compares a commitment to the last block with a commitment to B). The job of an off-chain verifier is similarly straightforward; it may also read the last sequenced block from the contract, or it may fetch the block directly from the sequencer along with some kind of authentication. In the case of a trusted, centralized sequencer, this authentication could be a simple signature.

Things are a bit more complicated when the confirmation layer is

1. decentralized, and
2. can be shared among multiple rollups

as is the case in Espresso. Blocks processed by HotShot may contain transactions meant for other rollups, which must be filtered out, and the process of checking that a block has been confirmed by HotShot is more complicated than verifying a simple signature.

Handling Multi-Rollup Blocks

Let's tackle the first problem first. The statement that must be proven now reads something like

Given a block commitment *C*, the block *B* contains all of the transactions belonging to rollup *R* in *C*, and only those transactions, in the same order as they appear in *C*. Furthermore, given an initial state root *S*, the result of applying the state transition function to *B* and *S* is a new state root *S'*.

The second part of the proof stays the same, which is good: this is where the zero-knowledge proof encodes the semantics of the rollup's state transition function, and, since rollup VMs tend to be fairly complex (e.g. EVM) this is where most of the proving complexity lies. Thus, the changes required of the state transition proof will mainly be additions, and will not affect the semantics of the state transition function at all.

The first part of the statement has changed substantially. Let's break it down:

Given a block commitment *C*

We work with a succinct commitment to the block, rather than the whole block, because we don't want to bring transactions from other rollups into the proof.

the block *B* contains all of the transactions belonging to rollup *R* in *C*

Since we do not have the entire block, only a commitment *C*, the state transition proof must contain a *completeness proof* which shows that the block being executed, *B*, does not omit any transactions for this rollup which were sequenced in *C*. Otherwise, two provers could execute a different subset of the relevant transactions, arrive at different state roots, and successfully verify proofs of both conflicting states. This attack essentially gives malicious actors a way to fork a ZK rollup, so the completeness proof here is essential!

To permit an efficient proof of completeness, *C* is computed according to the Savoiardi VID scheme described in Appendix A of The Espresso Sequencing Network. The Espresso SDK, also under development, will include predefined functions for working with these commitments and doing proofs about them, so the complexity will be abstracted away from rollup integrations.

and only those transactions

In addition to completeness, the block commitment scheme also defines a protocol for proving *inclusion*. This prevents a forking attack where a malicious prover executes some transactions that were not in the original block.

in the same order as they appear in *C*.

The last point is subtle. The inclusion and completeness proofs provided by the block commitment scheme enforce an ordering, which prevents a forking attack where a malicious prover executes the right transactions in the wrong order. However, some rollups may then apply a deterministic *reordering function* to ensure that the transactions are executed in an order which respects VM-specific constraints. For example, an EVM-compatible rollup may define the execution of a block *B* as *first* reordering the transactions in *B* by nonce order, and *then* executing the result. This reordering function would be encoded in the zero-knowledge proof just like other rules of the rollup's execution layer. It may be beneficial to do so, because HotShot is agnostic to the semantics of any particular rollup and does not enforce ordering constraints at the consensus level.

However, most rollups can probably consider this step optional. In practice, the sequencers elected through Espresso (and sequencers in general) will work with builders that are aware of rollup-specific semantics. These builders will seek to create valuable blocks, either by requiring users to pay fees or via arbitrage, and thus they will have an economic incentive to fill their blocks with valid transactions. In most cases, then, it will be sufficient for rollups to reject transactions that are "out of order", and in the case of EVM rollups, this is already done, as transactions with an incorrect nonce are considered invalid. Rejecting invalid transactions can be somewhat simpler than unconditionally sorting or reordering a whole block.

Checking That a Block Has Been Finalized

The last unsolved problem is how the rollup contract can check that a block commitment *C*, which has been proven to correspond to a certain list of rollup transactions, has actually been finalized at a given position in the chain.

With a centralized sequencer, it is usually straightforward to confirm whether a given block has been sequenced at a given position in the chain. The sequencer may have a known public key, for which it can produce a signature on any given block. Or, the sequencer may own the only Ethereum account authorized to call the function on the rollup contract which stores a new block on the layer 1.

With a decentralized system, things are a bit more complicated. Rollups that integrate with Espresso have their blocks processed by a decentralized **consensus protocol**, where thousands of nodes act as peers, and no one node has the privilege of unilaterally determining the status of sequenced rollup blocks. A block is considered *finalized* if this network of peers reaches consensus on the decision to include that rollup block at the next available position in the chain. Luckily, the process of reaching consensus produces artifacts which can be independently verified by non-participants of consensus, including smart contracts. These artifacts are called **quorum certificates**, or **QCs**.

A quorum certificate shows that a vote took place to include a certain block in a certain view, and that consensus nodes controlling a sufficient fraction of the total stake voted yes. The certificate contains an aggregated signature from those that voted. Due to the nature of distributed consensus, it actually requires three (in an upcoming version of HotShot this will be reduced to two) consecutive rounds of voting to finalize a block, so a chain of three consecutive valid QCs is definitive evidence that a block (and all previous blocks) has been finalized. Thus, any client, such as a rollup state transition verifier, wishing to verify that a rollup block has been sequenced must obtain and validate a chain of three consecutive QCs.

Luckily, the work required to verify finality for each block is the same across all rollups using Espresso, and can be shared. This is where the **sequencer contract** comes in. It is a single contract that receives commitments to sequenced blocks along with QCs proving the finality of those commitments, verifies the QCs, and stores a commitment to the finalized order of blocks. Anyone can append a commitment to a newly sequenced block to the contract, simply by providing a valid chain of QCs, which can be obtained from any honest consensus node.

Once the sequencer contract has done the hard work of checking QCs to verify that a block is finalized, anyone else can check finality simply by checking that the block is included in the sequence committed to by the contract. The contract uses a Merkle tree to commit to the sequence of finalized blocks, storing the root of this tree, so this check is usually done by Merkle proof.

In a typical ZK rollup, there are several parties who need to verify that a certain block has been sequenced:

- The executor/prover must verify that a block has been sequenced before executing it. It can do this easily by waiting for the sequencer contract to verify the block and emit an event confirming that the block has been finalized. However, the executor may opt to confirm the block faster than the sequencer contract (thus providing *preconfirmations* to users) by downloading and verifying the QCs itself. There are two options for verifying QCs:
    - Use the Espresso SDK to run the same QC verification algorithm that HotShot consensus uses
    - Participate in consensus as a HotShot node. The HotShot node interface exposes a stream of verified blocks that the executor can then consume.
- The rollup contract can read the sequence commitment directly from the sequencer contract. It then has two choices:
    - It can require the prover to pass in a Merkle proof showing that the block commitment *C* from the state transition proof exists at a certain position in the Merkle tree
    - It can make the Merkle root a public input to the state transition proof, and the prover can prove in zero knowledge that the commitment *C* exists at a certain position in the Merkle tree. These choices allow a tradeoff between work done by the prover (which may be slow) and work done by the rollup contract (which may be expensive). In either case, the Espresso SDK will provide functionality for verifying these Merkle proofs on-chain and for encoding them in a zero-knowledge proof.
- Rollup users may need to verify a state transition proof, especially in sovereign rollups, rather than relying on the rollup contract to do so. They can follow a very similar process as the executor/prover, either verifying QCs on their own or waiting for the sequencer contract to do so.

Transaction Format

From HotShot's perspective, a transaction is just an array of bytes with an integer attached to identify the rollup that the transaction belongs to. Therefore, rollups using Espresso can keep their existing transaction format. The only change required is that, if the rollup provides a service like JSON-RPC that accepts transaction submissions, it must be modified to attach the rollup identifier when forwarding the transaction to Espresso.

The rollup identifier works much like an EVM chain ID. Each rollup is completely free to choose whatever identifier they want. However, it is strongly recommended to choose an identifier that no other rollup is using, because the rollup identifier determines which transactions are included in the completeness proof when filtering a multi-rollup block. Therefore, if you choose a rollup identifier which is already in use, your rollup will be forced to execute not only its own transactions but also all of those intended for the rollup with the same ID.

Downloading Data

Once a block has been finalized, various rollup participants will need to download it or a subset of it from the Tiramisu data availability layer. We consider three main use cases:

- A node wants to get notified when a new block is finalized
- An end user wants a proof that a particular transaction has been included in a block, but they don't want to download the entire block. This is a way of obtaining fast finality, because once a transaction is included in a finalized block, it is guaranteed that the rollup will eventually execute it. (This follows from completeness proofs.)
- An executor wants to download just the subset of a block pertaining to the relevant rollup, with a proof that the server has provided the correct transactions in the correct order.

All of these use cases can make use of the **availability API**. Any HotShot node or client can provide this API by plugging in the modular HotShot query service.

New Block Notifications

The availability API provides several streaming endpoints, which a client can connect to using a WebSockets client. These endpoints allow clients to receive information when a new block is finalized or becomes available in the DA layer, without excessive polling. The streaming endpoints are:

- /availability/stream/leaves/:height

Stream blocks as soon as they are finalized, starting from :height (use 0 to start from genesis). The stream yields leaves, which contain metadata about finalized blocks such as the identity of the node that proposed them, the signature from nodes that voted for the block, and so on. This is the fastest way to learn of new blocks, but because Tiramisu disseminates data asynchronously, the actual *contents* of the block may or may not be included in this stream.

- /availability/stream/blocks/:height

This endpoint is similar to the leaves stream, but it waits until a block is fully available for download from Tiramisu before notifying the client. Each entry in the stream is a block with its full contents.

- /availability/stream/headers/:height

This endpoint is similar to the blocks stream, in that it will not notify the client of a new block until the full contents of that block are ready for download. However, it will not *send* the full contents of the block. It will only send the block header, which contains metadata like the block height and timestamp. This is a good way to wait until a block is finalized, at which time you can use some of the finer-grained endpoints discussed below to download only a subset of the block contents, saving bandwidth.

In the following sub-sections, it is assumed that clients of the availability API will use one of these streams to wait for more blocks to be sequenced before querying for the specific data they are interested in.

Single-Transaction Finality

The typical flow for this use case is

1. A user builds a transaction using rollup-specific client software.
2. The user saves the hash of their transaction and then submits it.
3. The user queries the availability API for proof that a transaction with the same hash has been included in a block.
4. The user checks that the resulting block has in fact been finalized.
5. The user verifies the proof, at which point it is guaranteed that the rollup will eventually execute the transaction.

The query for a proof uses the endpoint GET /availability/transaction/hash/:hash, replacing :hash with the tagged base 64 encoding of their transaction hash. If the requested transaction has in fact been sequenced, the response is a JSON object with a key proof, containing a proof of inclusion in a block, as well as metadata about the block, such as height and block_hash. It does not include the full block contents, so the bandwidth usage is minimal.

The user can check that block_hash has been sequenced as described above: either by checking for the corresponding event from the sequencer contract, or by downloading the relevant QCs and verifying them manually. The QCs can be obtained using the endpoint GET /availability/leaf/:height, and the Espresso SDK will include functionality for verifying them.

Once the user has confirmed that the block is finalized, the only thing left to do is to verify the proof that the transaction of interest was included in that block. This is a namespace KZG inclusion proof just like the ones used in the state transition proofs, and the SDK will include functionality for verifying it.

Rollup's Subset of a Block

Rollup executors must download transactions relevant to their rollup in order to execute, but it would be wasteful to download entire blocks, which may contain many transactions from other rollups. However, they do not generally want to trust the availability service to provide the correct subset of transactions. The desired flow is:

1. An executor queries for the relevant subset of the next block.
2. The availability service responds with the desired transactions and a proof of completeness and inclusion.

3. The executor verifies that the block has been finalized, verifies the completeness/inclusion proof, and then executes the transactions.

The executor's query has the form GET /availability/block/:height/namespace/:rollup-id. On success, this returns a JSON object with two keys:

- block, a commitment to the desired block
- proof, which includes within it a list of transactions and proves their inclusion and completeness in block.

The executor checks that block has been finalized at :height as described above: either by checking for the corresponding event from the sequencer contract, or by downloading the relevant QCs and verifying them manually. The QCs can be obtained using the endpoint GET /availability/leaf/:height, and the Espresso SDK will include functionality for verifying them.

proof is a KZG namespace proof just like the ones used in the state transition proofs, and the SDK will include functionality for verifying it. After verifying the proof, the executor is assured that block :height includes all the returned transactions, in the correct order, and no other transactions with ID :rollup-id. It can then execute the transactions to compute the next rollup state.

Data Availability

Tiramisu is a scalable and secure data availability solution which ensures that all sequenced blocks will be available for rollup participants to download and execute. This ensures that any participant can reconstruct the state of the rollup.

While Tiramisu with ETH restaking can be just as secure as Ethereum DA, using it as the only source of data availability technically makes a rollup into a **validium**. Some rollups in the Ethereum ecosystem place a high value on persisting all of their data to Ethereum. Espresso supports both approaches. Any rollup may continue to use Ethereum for DA in addition to Tiramisu simply by having a rollup node send each block produced via Espresso to a layer 1 contract. If your existing rollup already uses Ethereum DA, this is actually one less change you have to make!

Summary of Changes

This section summarizes the changes from the generalized ZK-rollup architecture that are required to integrate a ZK rollup with Espresso. Though the changes are presented in terms of an abstracted architecture, if you can map this abstraction onto your specific ZK rollup, you can derive a very concrete to-do list for integrating your rollup with Espresso.

1. Modify JSON-RPC or analogous server to forward transactions from users to Espresso. Choose a numeric ID for your rollup and attach it to the forwarded transactions.
2. Modify executor to stream notifications of new blocks from either:
   - The streaming availability API
   - Events emitted by the sequencer contract
3. Modify the executor to download full blocks, or rollup-specific subsets of blocks, from the sequencer availability API.

4. Change the interface of the state transition proof, replacing the rollup block or block commitment input with a HotShot block commitment.
5. Extend the proof to encode a proof of completeness and inclusion of the rollup block relative to the block commitment. For a ZK proof system formalized the usual way, in terms of constraints on an arithmetic circuit, this means adding more wires and constraints so that the circuit checks the inclusion/completeness proof, which becomes a witness to the circuit. Once released, the Espresso SDK can help define these additional constraints.
6. (Optional) Extend the proof to encode a deterministic reordering of the transactions provided by Espresso.
7. Update the prover to generate the new type of proof.
8. (If applicable) Update the on-chain proof verifier to check the new kind of proof. This generally involves replacing the preprocessed representation of the circuit. The contract will also need to accept a Merkle proof showing that the new block commitment input to the proof matches the corresponding block commitment in the Merkle tree maintained by the sequencer contract. Alternatively, the verification of this Merkle proof can be embedded in the circuit.
9. (If applicable) Update off-chain proof verifiers (validators or light clients) in the analogous way.