Running an Espresso Node

Information on the different ways to run an Espresso node. The Espresso node is referred to as a 'sequencer node' from here on forth.

Note: during Mainnet 0 only a fixed set of preregistered operators can run a node. The Espresso Network will upgrade to proof-of-stake in a later release.

Basic usage

Visit the espresso-sequencer repository for instructions on how to run an Espresso node natively or with Docker. Find the latest Docker images here.

Usage:

Copy

Run a node natively

target/release/sequencer [options] -- <module>

Run a node with the sequencer Docker image

docker run -it \

--name sequencer1 \

-e <env_variable> \

ghcr.io/espressosystems/espresso-sequencer/sequencer:main \

sequencer [options] -- <module>

For brevity, we will omit the full sequencer path from here and simply refer to the executable as sequencer going forward.

The sequencer docker-compose file is a great reference point for configuring an entire local sequencer network, including a few Espresso nodes, an L1 client, and the HotShot data availability and orchestrator servers.

Required parameters

Environment variable

CLI flag

Description

ESPRESSO_SEQUENCER_L1_PROVIDER

--l1-provider-url

JSON-RPC URI of the L1 provider (e.g. http://localhost:8545.

ESPRESSO_SEQUENCER_ORCHESTRATOR_URL

--orchestrator-url

URL of the HotShot orchestrator. This service will be provided by Espresso.

ESPRESSO_SEQUENCER_CDN_ENDPOINT

--cdn-endpoint

The CDN's entry point in host:port form. This service will be provided by Espresso.

ESPRESSO_STATE_RELAY_SERVER_URL

--state-relay-server-url

URL of the state relay web server. This service will be provided by Espresso.

ESPRESSO_SEQUENCER_STATE_PEERS

--state-peers

Comma-separated list of peer URLs to use for catchup. This may include the archival query service operated by Espresso as well as URLs of your own nodes (see catchup)

ESPRESSO_SEQUENCER_KEY_FILE

--key-file

Path to file containing private signing keys. See key management.

ESPRESSO_SEQUENCER_GENESIS_FILE

--genesis-file

Path to file containing genesis state. See genesis file.

ESPRESSO_SEQUENCER_LIBP2P_BIND_ADDRESS

--libp2p-bind-address

The address to bind libp2p to in host:port form. Other nodes should be able to access this. This should be a UDP port.

ESPRESSO_SEQUENCER_LIBP2P_ADVERTISE_ADDRESS

--libp2p-advertise-address

The address we should advertise to other nodes as being our libp2p endpoint (in host:port form). It should resolve a connection to the above bind address. This should be a UDP port.

Optional parameters

Environment variable

CLI flag

Description

ESPRESSO_SEQUENCER_PRIVATE_STAKING_KEY

--private-staking-key

The private staking key to use for signing consensus messages. This parameter *replaces* the required --key-file, and must be provided alongside --private-state-key. Sometimes it is more convenient to configure keys directly instead of via a file.

ESPRESSO_SEQUENCER_PRIVATE_STATE_KEY

--private-state-key

The private key to use for signing finalized consensus states. See also --private-staking-key.

ESPRESSO_SEQUENCER_IS_DA

--is-da

Whether or not to register for and participate in the the DA committee.

ESPRESSO_SEQUENCER_L1_EVENTS_MAX_BLOCK_RANGE

--l1-events-max-block-range

Maximum number of blocks allowed in a single eth_getLogs call to the L1 RPC provider

ESPRESSO_SEQUENCER_LIBP2P_BOOTSTRAP_NODES

--libp2p-bootstrap-nodes

A comma separated list of well-known peers to use to bootstrap the initial Libp2p connection. If supplied, it will override values from the orchestrator as well as those persistently saved to any config. Addresses need to be supplied in Multiaddress Format.

ESPRESSO_SEQUENCER_IDENTITY_NODE_NAME

--node-name

This is for use in Espresso's node validator dashboard. This is a string that if supplied, this value will be used to identify this particular node's display name on the Espresso's node validator dashboard. Example: espresso-sequencer-01

ESPRESSO_SEQUENCER_IDENTITY_COMPANY_NAME

--company-name

This is for use in Espresso's node validator dashboard. This is a string that if supplied, this value will be used to populate the node's organization/company name on Espresso's node validator dashboard. Example: Espresso Systems

ESPRESSO_SEQUENCER_IDENTITY_COMPANY_WEBSITE

--company-website

This is for use in Espresso's node validator dashboard.

This is a valid URL that points to the target homepage for the organization that is operating the sequencer.

Example:

https://www.espressosys.com

ESPRESSO_SEQUENCER_IDENTITY_OPERATING_SYSTEM

--operating-system

This is for use in Espresso's node validator dashboard. This is a string that is meant to represent the operating system that the sequencer is running on. Ideally it is meant to be the primary name of the underlying operating system plus a version number. If not supplied, this has a default value that comes from supplied system constant values. By default the value will just be the operating system name itself. The values supplied here will show as node distribution statistics on Espresso's node validator dashboard. On 'nix-based systems that can be populated with the uname command: uname -sr Example: Darwin 23.6.0 Linux 5.15.0-101-generic

ESPRESSO_SEQUENCER_IDENTITY_NODE_TYPE

--node-type

This is for use in Espresso's node validator dashboard. This is a string value that is meant to represent which binary that this node is being run with. This has a default value and doesn't need to be supplied at all, unless you wish to ovewrite the value here. The values supplied here will show as node distribution statistics on Espresso's node validator dashboard. Example: espresso-sequencer 1.0.0

ESPRESSO_SEQUENCER_IDENTITY_NETWORK_TYPE

--network-type

This is for use in Espresso's node validator dashboard. This is a string that is used to identify the type of network that the sequencer is run on. The values supplied here will show as node distribution statistics on Espresso's node validator dashboard. This value is expected to be a simple representation of the values used for distribution purposes. Ideally it should be one of the following values: - Residential - Hosted - AWS - GCP - Azure If a specific cloud provider would rather not be specified then Cloud Provider can suffice. If it is a well known cloud provider like AWS, GCP, or Azure and availability zone could be added to the end for further distinction. Example: AWS AWS us-west-2

ESPRESSO_SEQUENCER_IDENTITY_COUNTRY_CODE

--country-code

This is for use in Espresso's node validator dashboard. This is a string that is meant to contain the two letter Alpha 2 ISO-3166 country code of the sequencer being run. The values supplied here will show as node distribution statistics on Espresso's node validator dashboard. Example: US DE CN

ESPRESSO_SEQUENCER_IDENTITY_LATITUDE

--latitude

This is for use in Espresso's node validator dashboard. This is a value that is a decimal representation of the approximate latitude of the node. This value should only be supplied when the longitude value is also supplied, or should omitted. This value is meant to represent the approximate sequencer's latitude location. If concerned about being too specific about the

exact location of the node, then a location that indicates a higher administrative level is recommended.

This value is used to locate the node on the World Map display on Espresso's node validator dashboard. Example: 40.417300

ESPRESSO_SEQUENCER_IDENTITY_LONGITUDE

--longitude

This is for use in Espresso's node validator dashboard.

This is a value that is a decimal representation of the approximate longitude of the node. This value should only be supplied when the latitude value is also supplied, or should omitted. If concerned about being too specific about the exact location of the node, then a location that indicates a higher administrative level is recommended. This value is used to locate the node on the World Map display on Espresso's node validator dashboard. Example: -82.907100

[libp2p] Multiaddress Format

Multiaddresses are a self-describing format that allow us to support a wide variety of protocols and address types. More information can be found here.

Here is an example of an address that represents a node on localhost on port 2000:

Copy

/ip4/127.0.0.1/udp/2000/quic-v1/p2p/12D3KooWDtGECieXrqKoVxfDhU7afYnS6toj1GqWXuEDfcaGPDxa

You can also represent a fully qualified domain name using this format:

Copy

/dns/mynode.example.com/udp/2000/quic-v1/p2p/12D3KooWDtGECieXrqKoVxfDhU7afYnS6toj1GqWXuEDfcaGPDxa

Notice how for both we have appended the node's public key as part of the address scheme. You can get your own libp2p node's public key by using the dockerized public key tool:

Copy

docker run ghcr.io/espressosystems/espresso-sequencer/sequencer:main pub-key -l "YOUR_PRIVATE_STAKING_KEY"

Your libp2p key will then be printed to the console.

Genesis file

The genesis file is used to store settings that must be identical between all nodes, like the genesis state and parameters that impact the (in)validity of proposed blocks. Genesis files for all officially supported networks are bundled with the official Docker image, making it easy to distribute the configuration and ensure it is the same across all nodes:

Copy

% docker run ghcr.io/espressosystems/espresso-sequencer/sequencer:main ls /genesis

cappuccino.toml

demo.toml

staging.toml

This Docker image can be configured to run a node for any supported network simply by setting the genesis file path appropriately. For example, the local docker-compose demo uses ESPRESSO_SEQUENCER_GENESIS_FILE=/genesis/demo.toml. It is also possible to run a node for a custom network by defining your own genesis, mounting it via a volume, and pointing the node at the mounted path.

Format

The genesis file is a TOML file with the following sections:

- [chain_config] — Parameters of the state transition function of the Espresso chain.
    - chain_id — Identifier for this instance of an Espresso chain.
    - base_fee — Fee per byte of data sequenced.
    - max_block_size — Maximum block size allowed to be sequenced.
    - fee_recipient — Address used to track amount of fees paid.
    - fee_contract — Address of L1 contract used to deposit fee tokens.
- [header] — Inputs to the genesis header.
    - timestamp — The timestamp to use for the genesis header.
- [l1_finalized] — Description of the finalized L1 block from which to process L1 events.
    - number — The L1 block number.
    - timestamp — The L1 block timestamp.
    - hash — The L1 block hash.
- [stake_table] — Parameters for the Espresso stake table.
    - capacity — The maximum number of distinct public keys.
- [network] — Configuration for connecting to the peer-to-peer network.
    - bootstrap_nodes — List of peers to bootstrap a P2P connection.
- [accounts] — Prefunded fee accounts, for testing purposes only.

The [chain_config] section

This section defines parameters which affect how proposed blocks are validated and how they affect the Espresso state.

Copy

[chain_config]

chain_id = 999999999

base_fee = '1 wei'

max_block_size = '1mb'

fee_recipient = '0x0000000000000000000000000000000000000000'

fee_contract = '0xa15bb66138824a1c7167f5e85b957d04dd34e468'

All fields except fee_contract are required.

The chain_id field

The Espresso chain ID is a unique identifier for a given instance of the Espresso network. Since it is part of the chain config, and the chain config is hashed into each block header, an untrusting client can check whether a block it is looking at belongs to the chain with a certain ID, which prevents attacks where a malicious server provides valid blocks from the wrong chain.

The chain ID is also used for certain protocol transactions to prevent cross-chain replay transactions, as in other blockchains like EVM-based chains. Note, however, that the most common type of transaction—rollups submitting data to be sequenced—does not reference an Espresso chain ID, since rollups will have their own globally unique chain IDs to prevent cross-chain attacks.

The base_fee field

The base fee is the amount of Ether required to sequence a byte of data. These sequencing fees are paid by builders submitting blocks to be finalized by consensus, and are intended to offset the cost of operating the network. Note that fees are not currently distributed to node operators, but are collected in a burner address.

The base fee can be given as a number, decimal string, or hexadecimal string (with 0x prefix), indicating an amount in WEI. It may also be specified as a string containing a decimal number and a unit. Allowed units are wei, gwei, and eth.

The max_block_size field

The maximum size of a block. Honest nodes will reject a proposed block whose payload exceeds this size. This is intended as a failsafe in case the base_fee does not adequately disincentivize DOS attacks from submitting very large blocks. This parameter may be removed in the future, when the network is upgraded to use a more sophisticated, dynamic pricing model.

The maximum block size can be given as an integer (indicating a number of bytes) or a string with a number and a unit suffix. Allowed units consist of an SI prefix k, m, g, t, p, e (though you will likely never have use of prefixes larger than mega), an optional i indicating a binary system (factors of 1024 instead of 1000) and an optional b suffix. All of these are valid units: KiB, kb, KB, k.

The fee_recipient field

An address that receives all sequencing fees.

The fee_contract field

The address of a contract on the L1 chain which allows users and builders to deposit ETH into the Espresso state. The deposited tokens can then be used for paying sequencing fees.

This field may be omitted, in which case deposits are disabled. In this case, builders can still propose if

- They are proposing an empty block, or
- The base_fee is set to 0, or
- They have been funded in the genesis block

The [header] section

This section provides some inputs that will be used to construct the genesis header.

Copy

```
[header]

timestamp = "1970-01-01T00:00:00Z"
```

All fields are required.

The timestamp field

The timestamp which will be included in the genesis header. For all blocks after genesis, the timestamp is calculated dynamically, based on the system time of the proposing node. But the genesis header, and thus its timestamp, must be known to all nodes to even start proposing and validating blocks, so it must be configured.

The timestamp is given as an RFC 3339 string.

The [l1_finalized] section

This section determines the finalized L1 block from which Espresso will start processing events (like deposit events). The first block proposed by Espresso will process all events from the genesis l1_finalized block to the current finalized block, so this block should be recent enough that all thoes events can be processed within the consensus view timeout. Subsequent blocks will process events incrementally from the previous finalized L1 block to the current one.

If the L1 finalized block has not yet been finalized when a node is started, the node will wait for it to be finalized before starting consensus.

Copy

```
[l1_finalized]

number = 6001199

timestamp = "0x66575298"

hash = "0xcbc41e0637d18c879a9d6f09fb5046a37bae99d5637d12716b70fdd027a9135e"
```

If all fields are specified, the node will only accept the exact block described. If only number is specified, the node will fetch the details of that block using its L1 provider.

The number field

The block number of the finalized L1 block to use at Espresso genesis.

This field is required.

The timestamp field

The timestamp of the finalized L1 block to use at Espresso genesis.

This field may be omitted, in which case the timestamp will be fetched from the block indicated by number. Note that if timestamp is omitted, all other details must be omitted as well, and all details will be fetched based on number.

The hash field

The hash of the finalized L1 block to use at Espresso genesis.

This field may be omitted, in which case the hash will be fetched from the block indicated by number. Note that if hash is omitted, all other details must be omitted as well, and all details will be fetched based on number.

The [stake_table] section

This section configures the layout of the Espresso stake table. Note that this does not actually *define* the stake table. Currently the stake table is defined via the orchestrator service that nodes register with when they come online. Eventually, it will be defined by an L1 contract. This section of the genesis file only configures how the stake table is represented as a Merkle tree for proving purposes.

Copy

```
[stake_table]

capacity = 200
```

The capacity field

The maximum number of distinct nodes which can be represented by the stake table. After genesis, this can only be expanded with a network upgrade.

The [accounts] section

Use in testing environments only.

This section allows the definition of some prefunded accounts, for paying sequencing fees. This is useful for easily setting up a testing environment where a builder has enough funds to build blocks. However, this section must *only* be used in testing. Since the resulting funds are created by fiat instead of being bridged from L1, they do not actually correspond to tokens locked in the bridge contract on L1, and thus use of this section can cause the bridge contract to become insolvent.

The body of this section is a list of key-value pairs, where the keys are addresses and the values indicate the balance of those accounts at Espresso genesis. The balances can be given as numbers, decimal strings, or hexadecimal strings (representing an amount in WEI), or as strings containing a decimal number and a unit. Allowed units are wei, gwei, and eth.

Copy

"0x23618e81E3f5cdF7f54C3d65f7FBc0aBf5B21E8f" = 100000

"0x184ba627DB853244c9f17f3Cb4378cB8B39bf147" = "0xabcdef"

"0x184ba627DB853244c9f17f3Cb4378cB8B39bf147" = "1 eth"

The [[upgrade]] section

This section details the parameters and settings for performing a consensus protocol upgrade. Upgrades ensure that nodes move to a new version, applying changes such as fee adjustments or applying new features

The [[upgrade]] section specifies the version for which the upgrade should be applied. It also includes the hotshot configuration parameters. Hotshot provides two modes for upgrades: time-based and view-based

View:

Copy

[[upgrade]]

version = "0.2"

start_proposing_view = 5

stop_proposing_view = 400

Time:

Copy

[[upgrade]]

version = "0.2"

start_proposing_time = “2024-09-17T16:00:00Z”

stop_proposing_time = “2024-09-18T16:00:00Z”

View based:

- start_proposing_view: the earliest view in which the node can propose an upgrade. This should be set to when an upgrade is intended.
- stop_proposing_view: view after which the node stops proposing an upgrade

Time based:

- start_proposing_time: the earliest UNIX timestamp in which the node can propose an upgrade.
- stop_proposing_time: UNIX timestamp after which the node stops proposing an upgrade.

The window between start_proposing_view/time and stop_proposing_view/time should provide sufficient time e.g 300 views or 5-10 minutes for nodes to continue proposing the upgrade until successful.

For each upgrade, the upgrade type along with its associated parameters needs to be defined. Different upgrade types can be defined by creating separate sections within the within the [[upgrade]] array of tables. Currently only two types are supported: Fee and marketplace

Note: currently we only support one upgrade per run. To perform multiple upgrades, the sequencer binary needs to be restarted after each upgrade.

Fee upgrade example:

Copy

[upgrade.fee]

[upgrade.fee.chain_config]

chain_id = 999999999

base_fee = '1 wei'

max_block_size = '1mb'

fee_recipient = '0x0000000000000000000000000000000000000000'

fee_contract = '0xa15bb66138824a1c7167f5e85b957d04dd34e468'

click here for more details on upgrades.

Optional modules

The Espresso node supports a set of optional modules that extend the node with useful APIs (e.g. transaction submission or query functionality). Here we describe what these modules do and how to enable them.

In general, modules are enabled in the following way:

Copy

Run an Espresso node with a couple of modules enabled

sequencer -- <module1_name> --<arg1_name> <arg1_value> -- <module2_name>

HTTP

This module runs a basic HTTP server that comes with healthcheck and version endpoints. Additional endpoints can be enabled with the modules listed below.

Usage:

Copy

sequencer -- http --port 50000

Parameters:

Env Variable

CLI Flag

Description

ESPRESSO_SEQUENCER_API_PORT

--port

Port that the HTTP API will use.

ESPRESSO_SEQUENCER_MAX_CONNECTIONS

--max-connections

Maximum number of concurrent HTTP connections that the server will allow. E.g. 100.

Status

This module extends the HTTP API with telemetry and consensus metrics (e.g. an endpoint to retrieve the latest block height).

Usage:

Copy

sequencer -- http -- status

This will add a Prometheus endpoint GET /status/metrics containing useful metrics for monitoring the performance of the node and network. See also monitoring.

Catchup

This module extends the HTTP API with a module that serves queries for pending consensus state. Other nodes can connect to this API to quickly sync with the latest state in the event that they fall out of sync with consensus.

Usage:

Copy

sequencer -- http -- catchup

Config

This module extends the HTTP API with a module that provides the node config information. This will add two GET endpoints : /config/hotshot and /config/env.

Usage:

Copy

sequencer -- http -- config

The /hotshot endpoint retrieves the Hotshot config for the current node, excluding the private keys.

The /env endpoint outputs all Espresso environment variables set for the current node. The keys for these variables are defined in the public-env-vars.toml file in the crates/sequencer/api directory.

Query

This module enables a HotShot query service API that connects to a persistent store containing the history of the blockchain. This API provides endpoints that rollups can use to integrate with the sequencer.

This module must be enabled alongside the http module. The query API can be accessed at the port specified by the http module. This option also requires a storage module, which defaults to storage-fs (see below for more details on storage options).

Usage:

Copy

sequencer -- http -- query

Env Variable

CLI Flag

Description

ESPRESSO_SEQUENCER_API_PEERS

--peers

A comma-separated list of peer query service URLs to fetch missing data from.

Filesystem Storage

This module enables a local storage backend for the query service and consensus state. Eventually, the backend will also store DA blocks and VID shares. This setting is useful for testing and debugging, but is not recommended for production nodes because it is currently not very stable or performant. Long term, we hope to improve this storage option.

Usage:

Copy

sequencer -- http -- query -- storage-fs --path ./storage-path

Parameters:

Env Variable

CLI Flag

Description

ESPRESSO_SEQUENCER_STORAGE_PATH

--path

Storage path for persistent data.

SQL storage

This module enables a postgres storage backend for the query service and consensus state. Eventually, the backend will also store DA blocks and VID shares. This setting is recommended for production nodes.

Usage:

Copy

sequencer -- http -- query -- storage-sql

Parameters:

Environment variable

CLI flag

Description

N/A

--uri

This is a shorthand for setting a number of other options all at once in URI form. Components of this URI can overridden by the parameters below. **Example:** postgres[ql]://[username[:password]@][host[:port],]/database[?parameter_list].

ESPRESSO_SEQUENCER_POSTGRES_HOST

--host

Hostname for the remote Postgres database server.

ESPRESSO_SEQUENCER_POSTGRES_PORT

--port

Port for the remote Postgres database server.

ESPRESSO_SEQUENCER_POSTGRES_DATABASE

--database

Name of database to connect to.

ESPRESSO_SEQUENCER_POSTGRES_USER

--user

Postgres user to connect as.

ESPRESSO_SEQUENCER_POSTGRES_PASSWORD

--password

Password for Postgres user.

ESPRESSO_SEQUENCER_POSTGRES_USE_TLS

--use-tls

Use TLS for an encrypted connection to the database.

ESPRESSO_SEQUENCER_POSTGRES_PRUNE

--prune

Use this flag or set variable to true to enable pruning of historical data

ESPRESSO_SEQUENCER_PRUNER_PRUNING_THRESHOLD

--pruning-threshold

If storage usage exceeds this threshold (in bytes), data younger than the target retention will be pruned (possibly up to the minimum retention)

ESPRESSO_SEQUENCER_PRUNER_MINIMUM_RETENTION

--minimum-retention

The minimum time which data must be retained, regardless of storage usage

ESPRESSO_SEQUENCER_PRUNER_TARGET_RETENTION

--target-retention

The desired amount of time to retain data, storage permitting

ESPRESSO_SEQUENCER_PRUNER_BATCH_SIZE

--batch-size

The number of objects to delete at once when pruning

ESPRESSO_SEQUENCER_PRUNER_MAX_USAGE

--max-usage

The maximum fraction of pruning-threshold to use. If storage usage exceeds pruning-threshold, it will be pruned back to this fraction of pruning-threshold. Expressed as an integer on a scale of 1 to 10000.

ESPRESSO_SEQUENCER_PRUNER_INTERVAL

--internval

Interval for running the pruner.

ESPRESSO_SEQUENCER_FETCH_RATE_LIMIT

--fetch-rate-limit

Maximum number of simultaneous requests allowed when fetching missing data from peers. Setting this can limit the load placed on peers by catchup and in turn make catchup more efficient. E.g. 25.

ESPRESSO_SEQUENCER_ACTIVE_FETCH_DELAY

--active-fetch-delay

The minimum delay between requests to fetch data from another node, when syncing a DA or archival node. This can be used to ensure this node complies with the upstream node's rate limit. E.g. 50ms.

ESPRESSO_SEQUENCER_CHUNK_FETCH_DELAY

--chunk-fetch-delay

The minimum delay between loading chunks of data in a sequential stream. This can limit the load a node places on its own database, especially for streaming from old blocks, where lots of data is loaded from the database eagerly. E.g. 100ms.

Submit

This module extends the HTTP API with a POST endpoint to submit a transaction for sequencing.

Usage:

Copy

sequencer -- http -- submit

Key management

Each Espresso node needs two signing key pairs to run:

- The **staking key** is a BLS key used to sign consensus messages (votes, proposals), and it supports efficient signature aggregation, important for consensus performance.
- The **state key** is a Schnorr key used to sign finalized consensus states, which in turn drives the onchain Espresso light client on L1.

These keys are typically stored in a .env file within the sequencer container, and the sequencer is configured via ESPRESSO_SEQUENCER_KEY_FILE to load private keys from this file. While you are welcome to generate these keys however you like, as long as they have the right performance, Espresso provides a utility program keygen which is distributed with the sequencer image. The simplest way to generate keys is to run the following command:

Copy

docker run $IMAGE keygen -o /keys

$IMAGE is the ID or name of the sequencer node Docker image, such as ghcr.io/espressosystems/espresso-sequencer/sequencer:main. You can use Espresso's pre-built images or build the image yourself from the source code. This command will generate a file called /keys/0.env in the Docker container containing the private keys. It will also print the generated public keys in the terminal. You can then pass this file to the sequencer by setting ESPRESSO_SEQUENCER_KEY_FILE=/keys/0.env.

This method of generating keys is nice because the keys never leave the Docker container where they will be used. If, however, you want to store the keys on the host machine as well, or if you want to use a different container to generate the keys than you will use to run the sequencer (such as a one-off container from docker run) you need only create a Docker volume to store the keys in a host directory, such as

Copy

docker run -v ./keys:/keys $IMAGE keygen -o /keys

This will store the generated keys at ./keys on the host. You can then pass them into the sequencer by mounting the same volume in the sequencer container.

The keygen utility has some additional options which you can view by running with --help. One of the most useful is --seed <SEED>, to use a seed for generating the keys deterministically, instead of using entropy from the OS. This is particularly useful if you want to use your own entropy instead of the default entropy source: you can generate a randomized seed however you like and then pass it to the keygen program. The seed is a 32-byte integer encoded as hex (with no 0x prefix).

Monitoring

When running the status API, the performance of an Espresso node can be monitored using Prometheus tools, by monitoring the endpoint /status/metrics. Some of the most important metrics to monitor include:

- consensus_current_view: should be incrementing once every 1-2 seconds. In rare cases it is acceptable for this metric to remain static for up to 1 minute. If not increasing, it may mean the network has lost liveness, or your node has fallen out of sync with the network.
- consensus_last_decided_view: should be increasing mostly in tandem with consensus_current_view. If current_view is increasing but last_decided_view is not, it indicates a network-wide problem with consensus state, or a recurring problem with builders or proposers.
- consensus_number_of_timeouts_as_leader - The number of slots (views) this leader has failed to propose a block for. If this is increasing frequently, there is potentially an issue with your node.
- consensus_libp2p_num_connected_peers - The number of connected peers on the Libp2p network. This should be nonzero.
- consensus_outstanding_transactions: smaller is better, and this metric should not show any trend over time. If it is especially large (relative to volume) or increasing over a long period of time, it may indicate that your node is not garbage collecting the public mempool properly.

Here are some important logs to look out for as a secondary measure:

- Vote sending timed out in ViewSyncTimeout - We entered a view synchronization phase and did not immediately leave. We expect to see this in some cases (e.g. some leaders are failing in a row), but it's indicative of an issue if it does not eventually resolve.
- Failed to publish proposal - As the consensus leader, this prints if we fail to propose a block for some reason. It may be fine to happen occasionally (e.g. we just restarted and are still catching up) but lots of these are a problem.
- Progress: entered view - Every 100 views, we print this log with the view number. With the current block builder we expect to see this every ~13 minutes or less.

Modes

It is possible to run an Espresso node in three modes, differentiated by how long the nodes store historical data.

Lightweight node

A lightweight node stores only the data needed to run consensus. It does not keep any historical data, and it is not eligible to be on the consensus DA committee. It has negligible storage requirements on the order of kilobytes. A lightweight node is any node running without the optional query module.

Archival node

An archival node stores all historical data in perpetuity, and is thus able to serve queries for arbitrary historical state. It is eligible to be on the DA committee. Its storage requirements depend on how much data the network is processing, which in turn depends on how much the network is being used. In testnets, this has been on the order of tens of gigabytes per month, but in mainnet this may be more.

To run an archival node, simply enable the optional query module without any of the pruning options.

DA node

Pruning of old data is not yet supported with filesystem storage, and thus it is only possible to run a DA node with Postgres storage at this time.

A DA node provides data availability for recently finalized data. It is eligible to be one the DA committee, because it will make data available for long enough for an archival node to fetch it and persist it, before the data is pruned from the DA node. Unlike the archival node, the DA node has bounded storage requirements.

The storage requirements for a DA node are determined by how long we want it to retain data in the worst case. Typically, we want DA nodes to retain data for 1 week under average load, and a minimum of 1 day under worst case load, which give archival nodes time to ensure the data is persisted long term. Detailed hardware requirements for DA nodes are given below.

To run a DA node, enable the optional query module as you would for an archival node, but additionally set pruning parameters:

- ESPRESSO_SEQUENCER_POSTGRES_PRUNE=true
- ESPRESSO_SEQUENCER_PRUNER_MINIMUM_RETENTION=1d
- ESPRESSO_SEQUENCER_PRUNER_TARGET_RETENTION=7d
- ESPRESSO_SEQUENCER_PRUNER_PRUNING_THRESHOLD set to the worst case storage usage, in bytes, based on the hardware requirements
- ESPRESSO_SEQUENCER_IS_DA="true"

Hardware requirements

Hardware requirements are still in flux as we refine our testnets and add new features, but for now we recommend the following:

RAM: 16-32 GB.

CPU: 2-4 Cores.

Storage (DA node): 20 GB minimum, ability to scale to 1 TB on demand.

Storage (non-DA Node): Negligible, kilobytes