Cappuccino Testnet Release

Espresso testnet 5 (Cappuccino)—May 2024

With the release of Decaf, Espresso's persistent testnet, the Cortado testnet is currently paused.

In May 2024, we announced the Cappuccino release of Espresso.

Cappuccino continues the process of decentralizing Espresso, by onboarding a total of 10 operators to participate in Cappuccino. The 10 operators will be running a total of 100 geographically distributed nodes, all participating in HotShot together.

The Tiramisu DA layer is now also further upgraded and supports VID, which ensures that data is recoverable, even if the CDN and DA committee fail to be responsive.

As part of the Arbitrum tech stack integration, Espresso now supports Arbitrum Nitro fraud proofs, which enables Arbitrum chains to be fully productized on top of Espresso. We will be releasing test versions of Arbitrum Orbit chains in collaboration with Caldera and AltLayer.

We have also implemented a simple block builder for Cappuccino. This lays the foundation for shared sequencers elected through the Espresso marketplace to build valuable multi-rollup blocks and give out atomic execution guarantees for cross-chain transactions. You can track activity on the Cappuccino testnet in our new block explorer here: https://explorer.cappuccino.testnet.espresso.network/ And you can interact with Cappuccino via the public endpoint here: https://query.cappuccino.testnet.espresso.network/v0/

Running a Node

Configuration for Cappuccino nodes

Cappuccino node operators are limited to a select group. If you are interested in running a node in a future release of Espresso, contact us.

This page give the configuration used to run different types of nodes in the Cappuccino testnet. For general information on running an Espresso node, see Running an Espresso Node.

All nodes in Cappuccino use the ghcr.io/espressosystems/espresso-sequencer/sequencer:cappuccino Docker image. Depending on the type of node, the configuration varies.

Regular Node

Command

```
sequencer -- http -- catchup -- status
```

Environment

Same for all nodes

Copy

```
ESPRESSO_SEQUENCER_ORCHESTRATOR_URL=https://orchestrator.cappuccino.testnet.espresso.network

ESPRESSO_SEQUENCER_CDN_ENDPOINT=cdn.cappuccino.testnet.espresso.network:1737
```

ESPRESSO_STATE_RELAY_SERVER_URL=https://state-relay.cappuccino.testnet.espresso.network

ESPRESSO_SEQUENCER_GENESIS_FILE=/genesis/cappuccino.toml

RUST_LOG="warn,libp2p=off"

RUST_LOG_FORMAT="json"

# At least one state peer is required. The following URL provided by Espresso works.

# Optionally, add endpoints for additional peers, separated by commas.

ESPRESSO_SEQUENCER_STATE_PEERS=https://query.cappuccino.testnet.espresso.network

Chosen by operators

Copy

# JSON-RPC endpoint for Sepolia testnet

ESPRESSO_SEQUENCER_L1_PROVIDER # e.g. https://sepolia.infura.io/v3/<API-KEY>

# Port on which to host metrics and healthchecks

ESPRESSO_SEQUENCER_API_PORT # e.g. 80

# Path in container to store consensus state

ESPRESSO_SEQUENCER_STORAGE_PATH # e.g. /mount/sequencer/store/

# Path in container to keystore

ESPRESSO_SEQUENCER_KEY_FILE # e.g. /mount/sequencer/keys/0.env

Volumes

- $ESPRESSO_SEQUENCER_STORAGE_PATH
- $ESPRESSO_SEQUENCER_KEY_FILE

DA Node

Requires operator to additionally run a Postgres server

Command

sequencer -- storage-sql -- http -- catchup -- status -- query

Environment

Same for all nodes

Copy

```
ESPRESSO_SEQUENCER_ORCHESTRATOR_URL=https://orchestrator.cappuccino.testnet.espresso.network

ESPRESSO_SEQUENCER_CDN_ENDPOINT=cdn.cappuccino.testnet.espresso.network:1737

ESPRESSO_STATE_RELAY_SERVER_URL=https://state-relay.cappuccino.testnet.espresso.network

ESPRESSO_SEQUENCER_GENESIS_FILE=/genesis/cappuccino.toml

ESPRESSO_SEQUENCER_POSTGRES_PRUNE="true"

ESPRESSO_SEQUENCER_PRUNER_PRUNING_THRESHOLD="549755813888" # 0.5 TB

ESPRESSO_SEQUENCER_PRUNER_MINIMUM_RETENTION="1d"

ESPRESSO_SEQUENCER_PRUNER_TARGET_RETENTION="7d"

ESPRESSO_SEQUENCER_PRUNER_BATCH_SIZE=5000

ESPRESSO_SEQUENCER_IS_DA="true"

ESPRESSO_SEQUENCER_FETCH_RATE_LIMIT=25


RUST_LOG="warn,libp2p=off"

RUST_LOG_FORMAT="json"


# At least one state peer is required. The following URL provided by Espresso works.

# Optionally, add endpoints for additional peers, separated by commas.

ESPRESSO_SEQUENCER_STATE_PEERS=https://query.cappuccino.testnet.espresso.network

ESPRESSO_SEQUENCER_API_PEERS=https://query.cappuccino.testnet.espresso.network
```

Chosen by operators

Copy

```
# JSON-RPC endpoint for Sepolia testnet

ESPRESSO_SEQUENCER_L1_PROVIDER # e.g. https://sepolia.infura.io/v3/<API-KEY>


# Port on which to host metrics, healthchecks, and DA API

ESPRESSO_SEQUENCER_API_PORT # e.g. 80


# Path in container to keystore

ESPRESSO_SEQUENCER_KEY_FILE # e.g. /mount/sequencer/keys/0.env
```

# Connection to Postgres

ESPRESSO_SEQUENCER_POSTGRES_HOST

ESPRESSO_SEQUENCER_POSTGRES_USER

ESPRESSO_SEQUENCER_POSTGRES_PASSWORD

Volumes

- $ESPRESSO_SEQUENCER_KEY_FILE

Archival Node

Requires operator to additionally run a Postgres server

Command

sequencer -- storage-sql -- http -- catchup -- status -- query -- state

Environment

Same for all nodes

Copy

ESPRESSO_SEQUENCER_ORCHESTRATOR_URL=https://orchestrator.cappuccino.testnet.espresso.network

ESPRESSO_SEQUENCER_CDN_ENDPOINT=cdn.cappuccino.testnet.espresso.network:1737

ESPRESSO_STATE_RELAY_SERVER_URL=https://state-relay.cappuccino.testnet.espresso.network

ESPRESSO_SEQUENCER_GENESIS_FILE=/genesis/cappuccino.toml

ESPRESSO_SEQUENCER_FETCH_RATE_LIMIT=25

RUST_LOG="warn,libp2p=off"

RUST_LOG_FORMAT="json"

# At least one state peer is required. The following URL provided by Espresso works.

# Optionally, add endpoints for additional peers, separated by commas.

ESPRESSO_SEQUENCER_STATE_PEERS=https://query.cappuccino.testnet.espresso.network

ESPRESSO_SEQUENCER_API_PEERS=https://query.cappuccino.testnet.espresso.network

Chosen by operators

Copy

# JSON-RPC endpoint for Sepolia testnet

ESPRESSO_SEQUENCER_L1_PROVIDER # e.g. https://sepolia.infura.io/v3/<API-KEY>

# Port on which to host metrics, healthchecks, and query API

ESPRESSO_SEQUENCER_API_PORT # e.g. 80

# Path in container to keystore

ESPRESSO_SEQUENCER_KEY_FILE # e.g. /mount/sequencer/keys/0.env

# Connection to Postgres

ESPRESSO_SEQUENCER_POSTGRES_HOST

ESPRESSO_SEQUENCER_POSTGRES_USER

ESPRESSO_SEQUENCER_POSTGRES_PASSWORD

Volumes

- $ESPRESSO_SEQUENCER_KEY_FILE

Deploying a Rollup on Cappuccino

Users interested in deploying their own rollup on Cappuccino can make use of the following

- Query Service: https://query.cappuccino.testnet.espresso.network
- Light client contract address: 0xfdbf8b5ed2c16650aa835315a67d83eda5c98872

For more information on deploying an Arbitrum Nitro chain to Cappuccino, please see the following guide. This makes use of our Arbitrum Nitro integration with Espresso.

Benchmarks

Performance metrics for HotShot consensus and Tiramisu data availability in Espresso's Cappuccino testnet release

As a part of launching the Cappuccino testnet and releasing our implementation of HotShot under the MIT license, we are publishing benchmarks related to performance of this release. Compared to earlier benchmarks, these results benchmark the addition of Tiramisu DA's Savoiardi layer to the HotShot protocol.

In our evaluations, we progressively increased the block size from 50KB to 20MB and tested on network sizes ranging from 10 to 1000 nodes. In all settings, a subset of 10 nodes serves both as validators and the committee for Tiramisu DA's Mascarpone layer. As shown in the below figure, throughput rises with the increasing load without a corresponding increase in latency, up to a certain point of saturation. Beyond this point, latency begins to increase while throughput either remains steady or shows a slight increase. In the below table, we show the benchmark data for block sizes of 5MB block size, which is approximately the turning point.

HotShot throughput vs. end-to-end latency for varying network sizes and increasing block sizes

Network Size

Mascarpone Committee Size

Block Size (MB)

Average Latency (s)

Average View Time (s)

Throughput (MB/s)

10

10

5

3

1.08

4.58

100

10

5

2

0.85

5.76

200

10

5

4

1.21

4.04

500

10

5

9

1.97

2.48

1000

10

5

21

5.56

0.88

Experimental Setup

These benchmarks were run on HotShot version 0.5.63.

We conducted our experiments on two types of machines:

- CDN Instances: Our CDN (repository located here) is a distributed and fault-tolerant system responsible for routing messages between validators. The CDN was run across 3 Amazon EC2 m6a.xlarge instances located in the us-east-2 region. Each instance ran a broker, which is the component responsible for forwarding messages to their intended recipients. One instance also ran the marshal service, which is the service that facilitates the authentication and marshaling of validators to a specific broker. Each instance had 4 vCPUs and 16.0 GiB memory.
- Validator Instances: HotShot nodes were run on Amazon ECS tasks with 2 vCPUs and 4 GiB memory. Nodes were equally distributed among the us-east-2a, us-east-2b and us-east-2c availability zones.

Data Calculation

Each benchmark was run until 100 blocks were committed. After each benchmark run, nodes reported:

- the total time elapsed for the run
- the throughput per second
- the total latency
- the total number of blocks committed
- the total number of views it took to reach 100 commits
- the number of failed views (views that failed to make progress)

These values were collected and averaged in the final results. Note that throughput is measured in megabytes per second, not mebibytes per second.

Analysis of Results

- Our implementation of the Tiramisu data availability protocol achieves better maximum throughput in large networks than standard consensus protocols where data is sent to all nodes. That being said, this particular implementation's latency is worse in large networks. However, we've identified several implementation-specific bottlenecks to fix this issue.

- During benchmarks where the network is unsaturated with data, small network sizes (10 and 100 nodes) achieve finality in ~1s, and large network sizes (500, and 1000 nodes) achieve finality between 2-5s.
- The primary bottlenecks of this particular implementation are twofold:
  - Our current implementation of Tiramisu DA's Savoiardi layer is compute-intensive. This causes builders, leaders, and Mascarpone DA committee members to spend additional time computing Savoiardi shares during each view. This bottleneck can be addressed by more optimally parallelizing intensive compute, dynamically tuning Savoiardi parameters such as multiplicity to optimally encode block data, experimenting with different hardware such as GPUs, and having the Cocoa layer optimistically calculate Savoiardi shares.
  - The builder used in these benchmarks is a simple, naive builder. Unlike a sophisticated builder, this builder does no optimistic execution or optimistic Savoiardi calculations. The simple builder does not begin building blocks until the HotShot leader requests it to do so. This causes the builder to be slow in returning block data to the HotShot leader, thus adding unneeded latency each view. This bottleneck can be addressed by using a sophisticated builder that optimistically builds blocks.
- This implementation of HotShot uses the HotStuff-1 protocol. We plan to upgrade to the HotStuff-2 protocol in the future, which will reduce commit latency significantly.

Notes

- These benchmarks did not use a public transaction mempool. Instead, block builders were configured to build predetermined-sized blocks each view. This configuration is equivalent to block builders only building blocks with privately-sent transactions. A public mempool is part of the current HotShot implementation, however, and will be included in future benchmarks. Note that throughput and latency results will differ with the inclusion of the public mempool.
- **Multiplicity:** Tiramisu DA's Savoiardi VID scheme is inspired by Ethereum's danksharding proposal, where the block payload is viewed as a list of polynomial coefficients. Ordinarily, these coefficients are partitioned into multiple polynomials, and each storage node gets one evaluation from each of those polynomials. At the other extreme, one could instead gather these coefficients into a single high-degree polynomial, and give each storage node multiple evaluations from this polynomial. We use the word “multiplicity” to denote the number of evaluations per polynomial sent to each storage node. Multiplicity is a parameter that can be tuned between two extremes to optimize performance.