Enhancing Effectiveness of Ethereum Rollups with a Byzantine Fault Tolerant Finality Layer

Emanuele Valzano
Tor Vergata University of Rome
Italy
valzano@ing.uniroma2.it

Matteo Nardelli*
Bank of Italy
Italy
matteo.nardelli@bancaditalia.it

Valeria Cardellini
Tor Vergata University of Rome
Italy
cardellini@ing.uniroma2.it

Abstract

Ethereum Layer 2 (L2) rollups improve scalability but expose a trade-off between fast sequencer soft finality and slow Layer 1 (L1) settlement finality, limiting latency-sensitive applications that require timely and durable guarantees on transaction ordering. We introduce a Byzantine Fault Tolerant (BFT) finality layer that extends existing rollup architectures without requiring L1 or rollup protocol changes. This layer provides deterministic transaction-level finality ahead of L1 settlement by committing to the sequencer's transaction order, bridging the gap between soft and hard finality. At its core, the layer uses a 1-chain variant of the Jolteon consensus protocol, adapted to the rollup setting where a single sequencer determines transaction ordering. Experiments show sub-second committee finalization latency and stable performance under Byzantine faults.

CCS Concepts

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; *Distributed architectures*; • **Security and privacy** → *Security services.*

Keywords

Transaction ordering finality, BFT consensus, Jolteon, Ethereum

ACM Reference Format:
Emanuele Valzano, Matteo Nardelli, and Valeria Cardellini. 2026. Enhancing Effectiveness of Ethereum Rollups with a Byzantine Fault Tolerant Finality Layer . In *The 20th ACM International Conference on Distributed and Event-based Systems (DEBS '26), June 23–26, 2026, Lisbon, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3809481.3812611

1 Introduction

Ethereum is a general-purpose blockchain, where Layer 2 (L2) rollups constitute the principal solution to scaling. Rollups are off-chain solutions that batch many transactions and submit a single state commitment to the main chain. They deliver throughput and latency improvements, but their architectures typically rely on a centralized component, the *Sequencer*. The Sequencer collects transactions, assigns a total order, and exposes an ordered transaction feed. Although this approach is effective for performance, centralization limits both the strength and timeliness of transaction finality guarantees. Current designs provide two distinct levels of finality: *soft finality*, a fast but provisional Sequencer attestation of transaction inclusion that remains reversible; and *hard finality*, a slower, but Layer 1 (L1) secured confirmation that occurs once transactions are published to and finalized by Ethereum. This trade-off is problematic for applications that require timely and irreversible agreement on transaction ordering: it forces latency-sensitive applications to operate under weaker guarantees, whereas security-critical applications must tolerate higher latency.

In this paper, we propose a solution that complements the single-sequencer rollup architecture by introducing an intermediate *finality layer* between L2 and the Ethereum L1. It adds a third tier of guarantees: besides soft and hard finality semantics, applications can also access a new tier of finality that is stronger than Sequencer promises and significantly faster than full L1 settlement. Therefore, applications may continue to rely on Sequencer soft finality with no additional application-visible latency. Before Sequencer-issued transactions reach L1, clients remain vulnerable to transient forks and reordering. To address this, we introduce a Byzantine Fault Tolerant (BFT) consensus protocol that provides agreement on *when* a prefix of the Sequencer's transaction feed becomes irrevocable. While the Sequencer determines the transaction order, the finality layer determines when this order can no longer be changed. The design builds on Jolteon [11], which is adapted to the rollup context and optimized for minimal latency via a 1-chain commit rule—safety is preserved due to the absence of competing transaction branches in current rollup deployments. We validate the approach with a Rust prototype that incorporates real cryptographic primitives and a production-grade P2P networking stack. The paper is organized as follows. Section 2 provides the necessary background and motivation. Section 4 introduces the proposed solution, Section 5 presents the architecture and consensus protocol of the finality layer, and Section 6 evaluates its performance. Section 7 reviews related work, and Section 8 concludes.

2 Background and Motivation

Ethereum and Scalability Issues. Ethereum is a decentralized blockchain with smart contract functionality, acting as a distributed state machine in which the Ethereum Virtual Machine (EVM) manages state updates. Due to its machinery, it exhibits limited throughput, which can be improved with L2 solutions (mostly rollups). While L2 enables handling execution and storage off-chain, L1 still accepts transactions and state commitments, guaranteeing finality of state updates. Rollups fall into two categories (e.g., [12]): (i) *optimistic rollups*, which assume correctness and rely on a fraud-proof challenge window, and (ii) *Zero-Knowledge (ZK) rollups*, which

*The views expressed in this paper are those of the authors and do not necessarily reflect the views of the Bank of Italy.

DEBS '26, Lisbon, Portugal

ACM ISBN 979-8-4007-2693-4/2026/06
https://doi.org/10.1145/3809481.3812611

publish validity proofs, reducing finality latency but requiring significant computational overhead for proof generation.

Optimistic Rollups with Arbitrum Nitro. Among optimistic rollup protocols, Arbitrum Nitro stands out for its low-latency block production [4]. Nitro executes smart contracts off-chain and settles transaction results and fraud proofs on Ethereum. It separates transaction sequencing from deterministic execution: a dedicated *Sequencer* orders transactions and commits to their order on Ethereum by publishing transaction batches, while the *State Transition Function* applies these transactions deterministically—so that all honest nodes can independently derive the same state. Correctness is enforced via an interactive fraud-proof protocol that resolves execution disputes on-chain by isolating and verifying a single instruction, so only contested computations incur L1 costs.

Motivating Use Cases. The lack of immediate, irrevocable ordering guarantees limits the applicability of rollups for latency-sensitive workloads, as many applications require timely, deterministic agreement on the order of events (even if state settlement can be finalized later). For example, a physical merchant accepting an L2 payment must either trust sequencer soft finality or wait for L1 publication and finalization before releasing goods. Similarly, an interactive gaming application may need to promptly reward a user based on an on-chain outcome, for which relying only on centralized sequencing provides weak guarantees, while waiting for Ethereum settlement is too slow for the user workflow.

Why a Finality Layer? In Arbitrum Nitro and other rollups, a malicious or faulty Sequencer could publish divergent views of its off-chain transaction ordering, leading to conflicting states that can be resolved once the corresponding transaction batch is finalized on L1. Therefore, transaction inclusion remains provisional for extended periods, until the corresponding L1 block reaches Ethereum finality, which occurs after roughly two epochs, i.e., about 13–15 minutes [7]. From a systems perspective, the core issue is the absence of timely and irrevocable agreement on the ordering of transactions: while the Sequencer determines the transaction order, no mechanism exists to finalize a *prefix* of this order before L1 settlement. This motivates an intermediate finality layer that provides fast agreement on when a sequenced batch becomes irrevocable.

Byzantine Consensus Protocols. Byzantine consensus enables agreement on the irrevocable ordering of transaction batches in the presence of faulty participants. HotStuff [14] is a BFT consensus protocol that organizes decisions into a chain of *quorum-certified* blocks. Each block carries a quorum certificate (QC), and leaders propose new blocks referencing the highest QC known. HotStuff uses a *3-chain commit rule*: a block becomes final once it is the first of three consecutive QC-certified blocks. Jolteon [11] is a latency-optimized variant of HotStuff that adopts a 2-chain rule at the expense of a quadratic fallback (i.e., view-change) mechanism inspired by PBFT [6]: in each view, the leader collects timeout certificates containing replicas' highest QCs, enabling future proposals to safely extend the latest non-conflicting block. Both protocols guarantee *safety*, guaranteeing that honest replicas never commit conflicting states, and *liveness*, guaranteeing the protocol eventually makes progress under favorable network conditions.

3 System Model and Assumptions

We consider a rollup architecture where transactions are ordered by a single sequencer and eventually settled on an L1 blockchain. Our goal is to provide deterministic transaction-level finality for prefixes of this feed by establishing agreement on when a sequenced prefix becomes irrevocable (rather than determining or validating the transaction order itself). We separate responsibilities: the sequencer determines the transaction order, the finality layer determines when that order becomes irrevocable, and the L1 enforces state finality.

Fault Model. The proposed finality layer consists of a committee of $n = 3f + 1$ replicas connected by authenticated channels and operating under eventual synchrony (partial synchrony). Up to f replicas can exhibit Byzantine behavior. Digital signatures and hash functions are assumed to be secure.

Sequencer. We assume the existence of a single rollup sequencer, reflecting current rollup deployments. The sequencer is the unique source of transaction ordering. We assume that, if the sequencer produces conflicting versions of the transaction feed, such equivocation is cryptographically detectable by mechanisms external to the finality layer. Consequently, replicas operate on a logically single authenticated feed and do not attempt to resolve conflicting prefixes through consensus: proposals inconsistent with the authenticated feed are not voted upon, preventing quorum formation if divergence occurs.[1]

Finality Guarantees. Once a batch is committed by the BFT committee, the position of its transactions within the sequencer's feed is irrevocable and will not be reverted, even in the presence of Byzantine replicas. The protocol guarantees agreement on transaction ordering but does not provide state-level finality or execution validity, which remain subject to the rollup's existing fraud or validity mechanisms and eventual L1 settlement.

Scope and Limitations. The protocol does not decentralize transaction sequencing and assumes a live sequencer that eventually produces transactions. If the sequencer halts or censors transactions, progress relies on the rollup's fallback mechanism (e.g., delayed inbox on L1). If the sequencer produces invalid transactions, they are discarded by the L2 *State Transition Function*. Sequencer replacement, permissionless committee formation, and economic incentives are orthogonal concerns and left for future work.

4 Solution Overview

We propose an intermediate *finality layer* that provides stronger security guarantees without waiting for L1 settlement, while preserving the low-latency soft-finality path of existing rollups. Our design goals include: (i) modularity and compatibility with existing rollup architectures, (ii) resilience under minimal trust assumptions, and (iii) scalability to large networks.

Our finality layer consists of a BFT consensus network that consumes the Sequencer's real-time *transaction feed* and commits a *batch* that finalizes the transaction ordering for each *L2 block* produced by the Sequencer. The finality layer adopts a 1-chain variant of Jolteon [11], which we adapt to operate in the rollup

[1] We do not attempt to decentralize the sequencer, validate transaction correctness, or prevent censorship; these limitations are inherited from the underlying rollup design. In particular, a Byzantine sequencer may emit incorrect or undesirable transactions, but cannot cause conflicting prefixes of the transaction feed to be finalized.

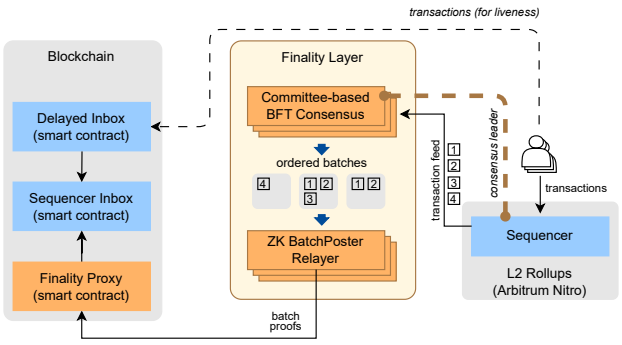

Figure 1: Finality layer architecture. Nitro components are shown in blue and the finality layer in orange.

setting. To ensure safety, responsibility for *authorizing* batch publication is delegated to the finality layer: transaction batches are accepted by L1 only if attested by cryptographic proofs produced by our protocol, ensuring that once a batch is finalized, its ordering cannot be revoked even by the Sequencer. Our protocol provides transaction-level finality rather than state-level finality[2], as finalizing state could introduce consistency issues with the canonical rollup finality path. Transaction-level finality enhances efficiency by eliminating the need for nodes to re-execute transactions during consensus. To ensure *liveness*, Nitro includes a Delayed Inbox smart contract on L1, and any message placed in it can be forcibly included into the L2 chain after a (24-hour) force-inclusion window, ensuring that the rollup continues to make progress even if the Sequencer stops publishing batches. Inclusion occurs once the message reaches the Sequencer Inbox contract, which maintains the complete sequence of messages. Note that, the proposed solution can be easily integrated into other rollup solutions with centralized sequencers.

5 Byzantine Fault Tolerant Finality Layer

5.1 Architecture

Figure 1 shows the architecture of the resulting system, with the newly introduced components in orange. The finality layer extends L2 rollups by leveraging only an existing API that exposes the transaction feed, i.e., ordered transactions to appear in L2 blocks (without the full EVM-compatible block structure). We update the Sequencer configuration to avoid publishing transaction batches on chain, which are instead published by the ZK BatchPoster Relayers. The existing rollup components remain unchanged; therefore, we present only the components of our finality layer.

Committee-based BFT Consensus. This is a set of nodes participating in the consensus protocol to agree on the irrevocability of prefixes of an ordered transaction sequence produced by the Sequencer. We employ a committee-based protocol to provide deterministic finality. The committee subscribes to the Sequencer's feed and commits batches that may embed multiple L2 blocks. Since the consensus network operates asynchronously with respect to

[2]Transaction-level finality fixes the inclusion and order of a transaction, while state-level finality guarantees that the resulting system state is irrevocable.

Let L denote the leader replica (bound to the Sequencer). Each replica maintains: highest voted view v_{vote}, highest QC qc_{high}, and last consumed feed index s_{last}. Replicas initialize $v_{vote} = (0, 0)$, qc_{high} as the QC of the genesis block, and $s_{last} = 0$. Protocol parameters: $(maxSize, maxRounds)$, i.e., the maximum size of a batch (in bytes) and maximum number of rounds for a checkpoint.

Consensus Protocol for Replica i

- **Propose.** From $qc_{high}.v$, the leader L advances to the next view $v = (r, c)$ according to the received feed and the $(maxSize, maxRounds)$ parameters; then, L gossips: $B = [\ id, v, qc, lastFeedSeq, feedMerkleRoot\ ]$.
- **Vote.** Upon proposal B, execute *Lock* and then *Commit*. Accept B if the following hold:
  (1) $B.v > v_{vote}$,
  (2) $B.v = B.qc.v + 1$,
  (3) $B.lastFeedSeq > s_{last}$,
  (4) From view $qc_{high}.v$, the feed up to $B.lastFeedSeq$ produces $(r', c') = (B.v.r, B.v.c)$ and $feedMerkleRoot' = B.feedMerkleRoot$.
  If valid, update s_{last}, v_{vote}, and send vote $\{B.id, B.v\}_i$ to L.
- **QC Formation.** The leader aggregates $2f+1$ votes to form $QC(B)$, which updates qc_{high}.
- **Lock.** On any valid qc, update $qc_{high} \leftarrow \max(qc_{high}, qc)$.
- **Commit.** On $QC(B)$, commit B.

Figure 2: Consensus protocol with a 1-chain commit rule.

the L2, and the Sequencer may produce transactions at a higher rate, each consensus node buffers incoming L2 blocks; the consensus protocol finalizes as many of them as possible. In particular, the protocol determines the batch boundaries by finalizing feed prefixes and posting to L1 the corresponding group of L2 blocks for each transaction batch. L2 blocks are finalized cumulatively until the batch boundary is reached. For example, in Figure 1, the first consensus batch embeds the first two L2 blocks, while the second batch contains the same two plus the next L2 block.

ZK BatchPoster Relayers. This is a permissionless pool of nodes that relay finalized batches to L1. Each relayer generates two separate ZK proofs witnessing correctness of the batch[3]. We use ZK to let L1 verify batch validity despite the EVM's inability to execute our cryptographic algorithms.

Finality Proxy. To avoid modifying the existing Sequencer Inbox contract, we introduce a *Finality Proxy* contract that verifies the ZK proofs attached to each batch. If the proofs are valid, the proxy publishes the batch to the Inbox contract. We update the configuration of Sequencer Inbox only to authorize calls from the Finality Proxy contract.

5.2 A 1-Chain Commit Rule Variant of Jolteon

In the rollup setting, the Sequencer is the unique source of transaction ordering. This fundamentally changes the role of the consensus layer. Unlike "classical" BFT state machine replication, where replicas must totally order arbitrary client requests, our replicas merely attest to the consistency and irrevocability of a single transaction log emitted by the Sequencer. Transactions are not finalized individually, but aggregated into *batches*, corresponding to a publication units on L1. This centralized ordering authority has two consequences. First, leader rotation and view-change mechanisms become unnecessary; the protocol therefore fixes the leader and binds it cryptographically to the Sequencer. Second, the absence of competing branches allows the protocol to safely adopt a *1-chain*

[3]The ZK proofs we employ do not refer to the L2 state commitments; therefore, rollups continue to operate optimistically to achieve state finality.

commit rule, under which a block becomes final as soon as it collects a quorum certificate. Safety follows from two properties. First, blocks proposed by the leader extend a single, totally ordered transaction feed produced by the Sequencer; hence, conflicting proposals for the same log position do not arise unless the Sequencer equivocates. Second, a block is finalized only after collecting a QC signed by at least $2f + 1$ replicas; by quorum intersection, two conflicting blocks cannot both obtain a QC when at most f replicas are Byzantine. Replicas only vote for proposals extending the highest certified block they have observed, ensuring monotonic growth of the committed chain. Liveness holds as long as the Sequencer continues to publish transactions and network conditions permit replicas to exchange messages, guaranteeing that valid blocks eventually accumulate a QC. When combined with the rollup's fallback mechanism (see Section 4), the system preserves full BFT guarantees.

Terminology. Consensus proceeds in *checkpoints*, which correspond to the construction of a batch of L2 transactions (to be posted on L1). Within a checkpoint c, progress advances in *rounds* r, grouped in lexicographically ordered views $v = (r, c)$. Each block B has the form $B = [\, id, v, qc, lastFeedSeq, feedMerkleRoot \,]$, where qc is the QC of the parent block, $lastFeedSeq$ identifies the highest Sequencer feed index included, and $feedMerkleRoot$ commits to the ordered transaction prefix forming the batch content. The block id is $H(v, qc, lastFeedSeq, feedMerkleRoot)$. A block B is *certified* when $2f + 1$ replicas sign $(B.id, B.v)$, forming a QC, expressed as *QC(B)*. The view of *QC(B)* is denoted by $QC.v$, which equals $B.v$.

Protocol. Our protocol follows the structure of Jolteon, adapted to the rollup setting (pseudocode in Figure 2). Since the leader is fixed (aligned with the Sequencer), the protocol can adopt a simplified commit rule. Unlike Jolteon 2-chain commit rule, we introduce a *1-chain commit* rule, under which a block is committed upon obtaining a QC. This enforces a strictly linear chain, eliminates forks by construction, and reduces commit latency (to three rounds instead of five). Replicas simply vote for the leader's proposals and commit blocks as soon as they are certified.

On-Chain Safeguards. A subtle issue arises because batches are constructed cumulatively: a malicious relayer could prematurely post a partial batch that still satisfies the on-chain proof logic, causing later extensions of the checkpoint to be rejected. To avoid this, the Finality Proxy contract accepts a batch only if it refers to the checkpoint's closing block, which is committed by the *first block of the next checkpoint*. To be sure that the previous checkpoint is complete, that first block must be justified by a QC provided by the relayers. To preserve liveness, if the next checkpoint does not appear within the *maxRounds* parameter, the contract enters a fallback window in which it accepts any batch justified by a committed block, selecting the batch associated to the highest round number. This requires only that one honest relayer remains synchronized with the consensus network. The relayer fetches from the consensus network the sealed batch, the checkpoint's closing block, and the next block that opens the new checkpoint along with its QC. It then generates two succinct (zk-SNARK) proofs witnessing that (i) the submitted QC is a valid threshold signature on the first block of the next checkpoint, and (ii) the batch matches the $feedMerkleRoot$ field of the checkpoint's closing block.

Finality Exposure. A transaction may obtain finality either (i) once its batch is posted and finalized on L1, or (ii) earlier, when included in a committed batch within an active checkpoint—finality is guaranteed by the consensus among replicas. The latter provides significantly stronger guarantees than the soft finality currently exposed by rollups and it is much faster than hard finality, although it becomes L1-hard only after eventual publication on-chain.

6 Evaluation

We implement the finality layer in Rust, using the `tokio` runtime for asynchronous task management, `ed25519` for digital signatures, RocksDB as storage engine, `libp2p` for networking, and the Keccak algorithm for cryptographic hashing to ensure Ethereum compatibility. We focus on the performance of the consensus component of the finality layer. The primary metric is *committee finalization latency*: it measures the delay from when an L2 block is proposed to the consensus layer until it is included in a batch certified by a valid QC.[4] We consider an increasing number of replicas as the fault tolerance parameter f grows from 1 to 6, therefore up to $n = 19$ nodes (recall, $n = 3f + 1$). Nodes are deployed across two cloud providers, DigitalOcean and Hetzner; each node has 4 dedicated vCPUs and 8 GB of RAM. As workload, the experiments in Sections 6.1, 6.2, and 6.3 use a fixed value of 100 Transactions Per Second (TPS) injected by the L2 system. Note that such load does not influence the communication overhead of our protocol as nodes send only fixed-size consensus messages. The experiment in Section 6.4 evaluates system performance with varying TPS. Each experiment runs the consensus nodes for 60 seconds and logs the finalization latency of every block certified during that time window. The evaluation does not include proof generation, relay-to-L1 delay, or on-chain verification, which are part of the full end-to-end publication path and are left to future work.

6.1 Baseline Setting

We evaluate the system scalability when deployed in a single datacenter (reduced networking delays), so to assess the consensus protocol performance as the number of nodes changes. The results reveal that the median finalization latency consistently remains below 6 ms, with only a slight increase as nodes are added (Figure 3). Interestingly, the 19-node configuration shows a reduction of the median latency with respect to 16 nodes: this can be attributed to the very low baseline times where small statistical variations, potentially due to operating system scheduling, cause fluctuations, as reflected in the upper quartile of the 16-node configuration. Overall, the system scales well with the number of nodes.

6.2 Geographically Distributed Setting

We consider replicas deployed across geographically distributed cloud regions. We use a round-robin policy that distributes replicas uniformly across three regions: US, Europe, and a final pool containing Sydney and Singapore. The goal is to assess the system's scalability in realistic deployments, and quantify the impact of network delays on finalization latency. Figure 4 shows the results: network latency represents the primary bottleneck, while the consensus protocol overhead remains limited. The finalization

[4] We estimate finalization latency as the time between a leader's proposal and the formation of its QC, isolating consensus latency. Pre-proposal buffering and batch construction are orthogonal and can be overlapped with consensus execution.

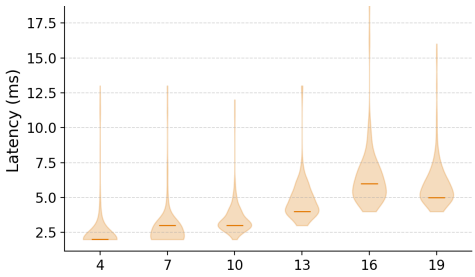

Figure 3: Finalization latencies under different numbers of replicas, when all nodes are deployed in the same region.

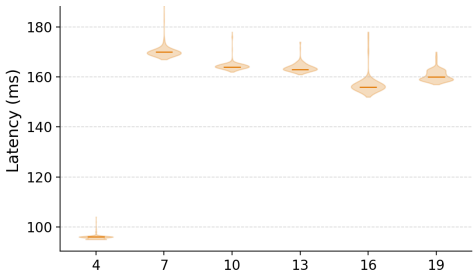

Figure 4: Finalization latencies under different numbers of replicas, when replicas are deployed globally.

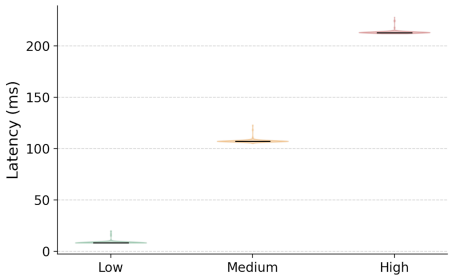

Figure 5: Finalization latency distribution under different scenarios of geographical distance of replicas from the leader. Each scenario corresponds to a deployment of 4 replicas.

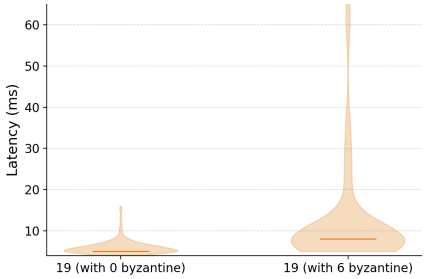

Figure 6: Finalization latency distributions. All nodes are deployed in the same region to analyze the consensus overhead.

latency shows a median value consistently below 180 ms, even with 19 nodes. Although there is an initial steep increase in latency going from 4 to 7 nodes, it almost stabilizes from 7 nodes onward. While an increasing number of nodes raises the computational overhead of the consensus, mostly due to linearly increasing QC verification times (the QC consists of multiple `ed25519` signatures), network delays represent the main system bottleneck[5].

To analyze the finalization latency increment observed going from 4 to 7 nodes, we extended the experiment and found that the latency increment was due to network topology rather than the number of nodes. In the 4-node setup, replicas were relatively close to the leader; conversely, in larger configurations, the quorum required votes from distant nodes. Figure 5 summarizes these findings, showing how the finalization latency increases under different network topologies with replicas progressively more distant from the leader. Our implementation uses a gossiping protocol for message dissemination, which constructs an application-layer overlay network that is decoupled from the underlying physical topology. Thus, messages traverse overlay paths, and end-to-end delay is governed by overlay latency. In our deployment, the overlay exhibited low stretch, causing the physically farthest replica to also induce the highest delivery delay. Therefore, the finalization latency is bounded by the slowest vote arrival along the overlay from any replica required to form the quorum. With respect to deployment, the low-distance scenario involves EU cloud regions, the medium-distance scenario spans EU and US-East regions, and the high-distance scenario spans US-West and Singapore regions.

6.3 Byzantine Fault Tolerance

We evaluate the impact of Byzantine replicas on consensus progress and finalization latency. To this end, we modify the node software to simulate a denial-of-service amplification/reflection attack. We do not exhaustively explore all forms of sequencer misbehavior, which are bounded by the system assumptions (Section 3). In this scenario, a maximal set of f malicious replicas collaborate to flood the network with malicious votes that correspond to incorrect block hashes and gossip these votes every 10 ms. Figure 6 shows that the attack causes higher variance of values but only a minimal increase (of about 3 ms) in the median finalization latency, indicating strong system resilience and efficient progress despite Byzantine faults.

6.4 Performance under Varying Loads

In this experiment, we evaluate the system's ability to handle the incoming load while, at the same time, observe how the finalization latency changes under increasing loads. We deploy only 4 nodes as the incoming load affects only the local processing at each replica and does not impact communication overhead. All nodes were deployed in the same cloud region.

Figure 7 shows that the system scales well as the incoming load (in TPS) increases, reaching approximately 100 ms finalization latency at 40, 000 TPS, the theoretical upper bound reported for Arbitrum[6].

[5]The `verify_batch` function provided by the `ed25519-dalek` library amortizes cryptographic costs across multiple signatures. Therefore, we expect the system to scale to significantly larger node counts before digital signature verification or network bandwidth pose real bottlenecks.

[6]Current realistic L2 TPS values peak at around 1, 400 TPS [8].

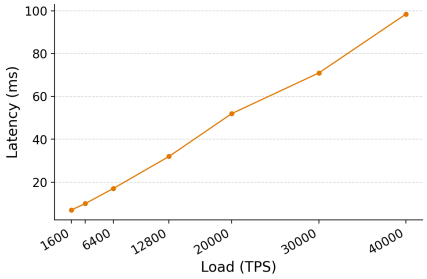

Figure 7: Finalization latency under varying loads.

7 Related Work

Rollup and Finality Models. Ethereum rollups emerged as the primary solution for scaling blockchain execution while preserving decentralization. Optimistic rollups such as Arbitrum [4] and Optimism[7] execute transactions off-chain and rely on fraud proofs to guarantee correctness. ZK rollups, such as zkSync and StarkNet, publish succinct validity proofs to ensure verifiable state transitions. Either way, finality depends on the publication and confirmation of batches on L1, which results in confirmation delays of several minutes due to Ethereum's finality times. Our work complements these approaches by introducing a third tier of deterministic, transaction-level finality achieved before L1 publishing and with minimal impact on the rollup architecture.

Finality Layers. Espresso provides transaction-level finality for multiple rollups through its HotShot BFT consensus protocol [10]. While Espresso targets cross-rollup coordination, our design is specialized for single rollups. This allows us to avoid leader rotation and achieve finality in hundreds of milliseconds. The batch publication design also differs: Espresso relies on TEE attestations, whereas our system uses ZK proofs, avoiding trusted hardware assumptions. A complementary approach is AltLayer MACH [2], which provides state-level finality through a combination of pessimistic re-execution, optimistic attestation, and ZK validity proofs. MACH introduces nontrivial execution overhead and can diverge from the rollup's canonical state due to its independent attestation pipeline. Related designs, such as the EigenLayer AVS framework [9], support external validation networks but do not natively implement consensus for transaction ordering. Conversely, our solution provides transaction-level finality, avoiding expensive state re-execution and remaining fully aligned with the rollup state progression.

Consensus Protocols. Blockchains fostered the design of efficient consensus protocols [13]. Among them, HotStuff [14] is a leader-driven BFT protocol with linear communication complexity, which stands out for simplicity and performance. Jolteon and Ditto [11] refine the protocol: while the former reduces commit latency via a 2-chain rule, the latter adds an asynchronous fallback inspired by PBFT [6]. We build on Jolteon by adapting it to the rollup context, which allows to eliminate leader rotation and to adopt a 1-chain commit rule.

L1-L2 Bridges. The security of rollups relies on on-chain contracts, which guarantee controlled batch publication and fraud-proof verification. Several other works focus on data availability (e.g., [5]) and fraud proof pipelines (e.g., [1]). Nonetheless, these mechanisms focus on ensuring correctness rather than providing faster finality. Our Finality Proxy complements these contracts by enforcing safe publication, avoiding premature posting, and preserving consistency between L2 and L1.

[7] https://docs.optimism.io/op-stack/fault-proofs/explainer

8 Conclusion

We introduced a BFT finality layer for Ethereum rollups that provides sub-second, deterministic transaction-level finality while preserving compatibility with existing architectures. Our design combines a 1-chain variant of Jolteon with a verifiable publication mechanism, enabling strong guarantees for latency-sensitive and security-critical applications. While the Sequencer remains responsible for transaction ordering, the finality layer determines when this ordering becomes irrevocable. Our prototype-based evaluation shows that the system scales efficiently, tolerates Byzantine faults, and sustains high workloads. Results demonstrate that L2 design can incorporate robust finality without sacrificing performance.

Future work will extend the prototype beyond the consensus path to evaluate proof generation (using Groth16 [3]), relay-to-L1 publication delay, and on-chain verification costs. We also aim to explore dynamic committee management and cryptoeconomic staking mechanisms to enable a permissionless setting. Another promising direction is the protocol extension to support state-level finality, while preserving compatibility with existing rollup pipelines.

References

[1] Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. 2021. Fraud and Data Availability Proofs: Detecting Invalid Blocks in Light Clients. In *Financial Cryptography and Data Security*, Nikita Borisov and Claudia Diaz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 279–298.
[2] AltLayer. 2025. Documentation. https://docs.altlayer.io/altlayer-documentation.
[3] Arkworks Contributors. 2025. Groth16 Implementation in arkworks. https://github.com/arkworks-rs/groth16
[4] Lee Bousfield, Rachel Bousfield, Chris Buckland, et al. 2022. Arbitrum Nitro: A Second-Generation Optimistic Rollup. https://resources.cryptocompare.com/asset-management/808/1688638950286.pdf
[5] Margarita Capretto, Martin Ceresa, Antonio Fernandez Anta, Pedro Moreno-Sanchez, and Cesar Sanchez. 2025. A Secure Sequencer and Data Availability Committee for Rollups. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 2129–2143. doi:10.1145/3719027.3765187
[6] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proc. of USENIX OSDI'99*. 173–186.
[7] Chainlink. 2025. CCIP Execution Latency: Transaction Finality and Timing. https://docs.chain.link/ccip/ccip-execution-latency#finality
[8] Chainspect. 2025. Arbitrum One Network Metrics. https://chainspect.app/chain/arbitrum.
[9] EigenLayer Team. 2024. Eigenlayer: The Restaking Collective. https://caladex.io/api/files/papers/EigenLayer_WhitePape.pdf.
[10] Espresso Systems. 2025. Rollup Architecture. https://docs.espressosys.com/network/concepts/rollup-architecture
[11] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, et al. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography and Data Security*. Springer, 296–315. doi:10.1007/978-3-031-18283-9_14
[12] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. 2022. Blockchain Scaling Using Rollups: A Comprehensive Survey. *IEEE Access* 10 (2022), 93039–93054. doi:10.1109/ACCESS.2022.3200051
[13] Jie Xu, Cong Wang, and Xiaohua Jia. 2023. A Survey of Blockchain Consensus Protocols. *ACM Comput. Surv.* 55, 13s, Article 278 (July 2023), 35 pages. doi:10.1145/3579845
[14] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, et al. 2019. HotStuff: BFT Consensus in the Lens of Blockchain. arXiv 1803.05069.