TAIKO: AN ETHEREUM-EQUIVALENT BASED ROLLUP

2.0.0 (September 12, 2025)

TAIKO LABS (INFO@TAIKO.XYZ)

Abstract. Taiko Alethia is an Ethereum-equivalent based rollup that achieves scaling through Ethereum's native validator set rather than a separate sequencer. As a based rollup, Taiko Alethia leverages Ethereum L1 validators for transaction sequencing, ensuring maximum decentralization and inheriting Ethereum's liveness and censorship resistance properties. This architecture eliminates the need for centralized or semi-centralized sequencers while maintaining full EVM-equivalence. Supporting all existing Ethereum applications, tooling, and infrastructure remains the primary goal. Taiko Alethia operates with asynchronous batch-based processing, where multiple blocks are proposed together and proved with aggregated validity proofs. Block proposals and proofs are asynchronous, allowing flexibility in proof submission timing. Taiko Alethia implements a multiproving system where blocks are verified through a combination of validity proofs including ZK-SNARKs and SGX attestations, providing both security and efficiency. A two-phase preconfirmation system provides fast transaction confirmation, transitioning from whitelisted operators to permissionless L1 validator participation with cryptoeconomic security. The protocol consists of three main components: Ethereum L1 validators who sequence transactions, provers who generate validity proofs for batch transitions, and a set of smart contracts deployed on Ethereum L1 that manage the rollup's state and verify proofs. By building directly on Ethereum's consensus layer, Taiko Alethia achieves true decentralization from day one while maintaining the security guarantees and network effects of Ethereum. This based rollup design represents a paradigm shift in L2 architecture, prioritizing decentralization and Ethereum-alignment over maximum theoretical throughput.

1. Introduction

Ethereum is well on its way into executing its rollup-centric roadmap to achieve scalability[1]. This progress has been shared by the independent rollup projects, as well as Ethereum itself which has coordinated to accommodate rollup-friendly upgrades.

At its base layer, facing the blockchain trilemma, Ethereum has always been unwilling to sacrifice decentralization or security in favour of scalability. These principles have made it the most compelling network to secure value. Its popularity, however, has often congested the network, leading to expensive transaction fees and crowding out certain users and use cases. To serve as the world's settlement layer for an internet of value, the activity that Ethereum settles will increasingly be executed on rollups: layer-2 scaling environments tightly coupled to and secured by Ethereum.

Rollups have shifted the tradeoff space: scaling to serve all users who seek to transact on Ethereum and enabling lesser-value, non-financial applications without subordinating Ethereum's strong claim of credible neutrality. There now exists a new tradeoff space among different rollup constructions, and there exists a hope to again shift the solution curve, rather than move along it. Taiko Alethia attempts to do exactly that, by implementing a based rollup that stays as true to the EVM and Ethereum specifications as possible, while leveraging advanced cryptographic proofs including ZK-SNARKs and SGX attestations for efficient batch verification.

Taiko Alethia aims for full Ethereum-equivalence, allowing our rollup to support all existing Ethereum smart contracts and dapps, developer tooling, and infrastructure. Complete compatibility benefits developers who can deploy their existing solidity contracts as is, and continue using the tools they are familiar with. This compatibility also extends to network participants and builders of Taiko's L2 blockchain, who can, for example, run Taiko Alethia nodes which are minimally modified Ethereum execution clients like Geth, and reuse other battle-hardened infrastructure. Finally, it extends to end-users, who can experience the same usage patterns and continue using their preferred Ethereum products. We have seen the strong demand for cheaper EVM environments empirically, with dapp and protocol developers as well as users often migrating to sidechains or alternative L1s which run the EVM, even if it meant much weaker security guarantees.

To be Ethereum-equivalent means to emulate Ethereum along further dimensions, too. Prioritizing decentralization and security within the layer-2 architecture ensures there is no dissonance between the environments, and that the Ethereum community's core principles are upheld. With calldata cost reductions such as EIP-2028[2] and blob-based cost reduction in EIP-4844[3], as well as other mechanisms coming in the future, Ethereum's commitment to rollups is strong and credible; rollups' commitment to Ethereum ought to be the same.

2. Previous Work

The Ethereum ecosystem began exploring layer-2 solutions for scaling in 2017 with Plasma[5]. Layer-2s move computation off-chain while keeping data either on Ethereum or also off-chain.

Rollups, which post compressed transaction data to Ethereum, emerged as the leading scalability path over the past several years, offering stronger security guarantees than earlier L2 solutions (Plasma and State Channels) while supporting a broader range of applications. Initially proposed by Vitalik Buterin[6] and Barry Whitehat[7] in 2018, rollups have evolved through multiple paradigms.

Early ZK-Rollups, starting with Loopring in 2019, were application-specific due to ZKP limitations, precluding general EVM use cases and composability. Optimistic rollups like Optimism and Arbitrum achieved EVM-compatibility in 2021 but rely on fraud proofs, introducing challenges

Date: September 12, 2025.

including dependency on watchers for security and lengthy withdrawal periods that hinder cross-rollup composability.

Projects such as zkSync, Starkware, Polygon, and Scroll, supported by the Ethereum Foundation's Privacy and Scaling Explorations unit[8], have developed ZK-powered rollups that combine EVM compatibility with cryptographic security. However, most existing rollups, both optimistic and ZK-based, rely on centralized or semi-centralized sequencers, creating single points of failure, censorship risks, and MEV extraction concerns.

Based rollups, first conceptualized by Justin Drake[16], represent a paradigm shift in rollup design. Rather than operating independent sequencers, based rollups leverage Ethereum L1 validators for transaction sequencing. This approach inherits Ethereum's liveness, censorship resistance, and decentralization properties while eliminating sequencer-related risks. The concept builds upon earlier work on forced inclusion mechanisms[17] and shared sequencing[18].

Recent research has explored various aspects of based rollup design, including preconfirmations for faster transaction confirmations[19], multiproving systems for enhanced security[20], and mechanisms for cross-rollup composability[21]. The trade-off space has shifted from EVM-compatibility versus proving efficiency to decentralization versus maximum theoretical throughput.

Taiko Alethia's approach prioritizes Ethereum-equivalence and decentralization through its based rollup architecture with asynchronous batch-based processing. The protocol maintains security through a multiproving system combining ZK-SNARKs and SGX attestations. This design philosophy aligns with Ethereum's core values while providing practical scalability, as we describe throughout this paper.

3. Design Principles

Taiko Alethia's based rollup architecture is guided by fundamental principles that prioritize decentralization, security, and Ethereum alignment:

(1) **Based Sequencing.** Transaction sequencing is performed by Ethereum L1 validators rather than a separate sequencer set. This eliminates centralized sequencers and inherits Ethereum's liveness guarantees, ensuring the rollup cannot halt independently of L1.
(2) **Permissionless Participation.** Anyone can propose blocks and generate proofs without special permissions. The protocol is designed such that anyone is able to opt-in and propose or prove blocks. Phase 1 of preconfirmations Taiko Alethia is permissioned, but with phase 2 rollout and whitelist removal, will return to permisionless sequencing and proposing.
(3) **Ethereum-Equivalence.** The rollup maintains full compatibility with Ethereum by directly interpreting EVM bytecode, using identical hash functions, state trees, transaction trees, and precompiled contracts. This ensures seamless deployment of existing Ethereum applications and tooling.
(4) **Security.** State transitions are secured through multiple independent proof systems, including ZK-SNARKs and SGX attestations. This approach ensures security even if one proving system is compromised.
(5) **Censorship Resistance.** By inheriting Ethereum's censorship resistance through based sequencing and supporting forced transaction inclusion, users cannot be censored by any single party. Transactions submitted to L1 are guaranteed eventual inclusion.
(6) **Economic Alignment.** MEV and transaction fees flow to Ethereum validators and the Ethereum ecosystem, strengthening L1's economic security. The rollup's economic success directly reinforces Ethereum's sustainability.
(7) **Minimal Trust Assumptions.** Security relies solely on Ethereum's consensus and cryptographic proofs, with no additional trust assumptions. The protocol avoids, as much as possible, introducing new honest-majority assumptions or trusted parties.

These principles guide Taiko toward building a truly decentralized scaling solution that extends Ethereum without compromising its core values. By leveraging based sequencing, Taiko Alethia achieves decentralization while maintaining practical performance through preconfirmations and efficient proving systems.
The protocol may initially disable certain EIPs[9] for stability, with plans to enable them as the system matures (see Section B).

4. Overview

Taiko Alethia builds a secure, decentralized, based rollup on Ethereum with native preconfirmation support. These requirements dictate the following properties:

(1) **Data Availability on L1:** All block data required to reconstruct the post-block state is published on Ethereum, ensuring public availability. This satisfies the rollup property and maintains decentralization. For permissionless proof generation, all data needed to re-execute blocks step-by-step must be publicly accessible, enabling anyone to generate proofs using only public data.
(2) **Based Sequencing with Preconfirmations:** Block creation leverages Ethereum validators for sequencing while supporting fast preconfirmations. Through Taiko Alethia's based preconfirmation mechanism, users receive up to sub-second transaction confirmations from preconfirmers who provide economic guarantees before L1 inclusion. This combines the decentralization of based sequencing with the UX benefits of fast confirmations.

Taiko Alethia implements a sophisticated block lifecycle:

Preconfirmation Phase: Users submit transactions to preconfirmers who provide a receipt via inclusion in an "unsafe" block, which is created and gossiped via P2P. These preconfirmations offer economic guarantees backed by collateral, giving users confidence that their transactions are executed within seconds rather than waiting for L1 block times. When the batch is proposed to the L1, these blocks achieve finality (given that the L1 does not experience a reorg).

Block Proposal: Preconfirmed transactions are batched and proposed to the TaikoInbox contract on Ethereum. Once registered, all block properties become immutable, making execution deterministic. The block is immediately considered *executable* as anyone can calculate the post-block state. The Pacaya fork ensures preconfirmers honor their commitments through slashing conditions.

Block Verification: Provers generate validity proofs (ZK-SNARKs or SGX attestations) for proposed blocks. Since blocks are deterministic post-proposal, they can be proven in parallel. The Pacaya fork's multiproving system requires different proof types for enhanced security. Once proven, blocks achieve *on-chain verification*.

The Pacaya fork optimizes costs through intelligent batching:

Batch-based Protocol: Blocks are proposed in batches to amortize L1 costs. Each batch can contain multiple L2 blocks that share metadata and transaction sources (calldata or blobs). This enables frequent small blocks in a single batch, crucial for supporting responsive preconfirmations while maintaining economic viability.

Preconfirmation Integration: The batching system coordinates with preconfirmers to ensure committed transactions are included within their promised timeframes. This maintains the validity of preconfirmation guarantees while optimizing L1 resource usage.

This architecture enables up to sub-second preconfirmations with strong economic guarantees while preserving the security and decentralization properties of based sequencing (see Section 6).

5. The Taiko Alethia Blockchain

Taiko Alethia operates as an Ethereum-equivalent based rollup with asynchronous batch-based processing. The blockchain consists of batches containing multiple blocks of user transactions, where each batch is proposed to the TaikoInbox contract on Ethereum L1. Block proposals and proofs operate asynchronously, allowing flexibility in proof submission timing while maintaining security through economic bonds.

The protocol implements a multiproving system supporting multiple verifier types including SGX attestations, SP1 proofs, and Risc0 proofs. Each batch undergoes verification through designated verifier contracts, with proof systems presumed error-free validated.

The protocol architecture includes a two-phase preconfirmation system aimed at improving user experience through fast transaction confirmation. Phase 1 utilizes whitelisted operators managed through PreconfWhitelist and PreconfRouter contracts, while Phase 2 transitions to permissionless L1 validator participation with cryptoeconomic security. Authorized preconfirmers can propose batches through the PreconfRouter contract, enhancing user experience with up to sub-second confirmations while maintaining the decentralization properties of based sequencing.

5.1. **Core Contracts.**

Taiko Alethia implements its based rollup protocol through a suite of smart contracts deployed on both L1 and L2, with additional contracts supporting the preconfirmation and forced inclusion systems.

5.1.1. *Layer 1 Contracts.*

TaikoInbox. The primary L1 contract for batch proposals in Taiko Alethia. The contract implements asynchronous batch-based processing where multiple blocks are proposed together and proved with aggregated validity proofs. The contract maintains:

state: A State struct containing batch ring buffer, transition states, and bond balances.

inboxWrapper: Optional address that when set, restricts batch proposals.

verifier: Address for contract that handles proof verification.

signalService: Interface for cross-layer communication and state synchronization.

PreconfWhitelist. Manages the authorized preconfirmation operators with epoch-based selection:

operators: Registry of preconfirmation operators with activation/inactivation epochs.

epochSelection: Selects operators for specific epochs using randomness from beacon block roots.

operatorManagement: Supports adding, removing, and consolidating operators with configurable delays.

PreconfRouter. Provides a controlled entrypoint for sequencers to propose batches:

batchProposal: Routes batch proposals from authorized preconfirmers to the TaikoInbox.

authorization: Ensures only whitelisted preconfirmers or fallback addresses can propose.

blockValidation: Optionally validates expected last block ID before accepting proposals.

ForcedInclusionStore. Allows users to pay a fee to ensure their transactions are included in a block:

queue: FIFO queue of forced inclusion requests with head and tail tracking.

inclusionDelay: Configurable delay before forced inclusions become processable.

`getOldestForcedInclusionDeadline()`: Returns when the oldest forced inclusion becomes due.

TaikoWrapper. A delayed inbox implementation to enforce transaction inclusion:

`proposeBatch()`: Processes both forced inclusion and regular batches.

validation: Ensures minimum transaction requirements for forced inclusions.

integration: Works with TaikoInbox and ForcedInclusionStore contracts.

5.1.2. *Layer 2 Contracts.*

TaikoAnchor. The primary L2 contract that handles cross-layer message verification and manages EIP-1559 gas pricing:

(1) *Anchoring*: Anchors latest L1 block details to L2 and verifies cross-layer message integrity.
(2) *State Root Synchronization*: Synchronizes L1 state roots via signal service for multi-hop bridging.

(3) *Gas Pricing*: Calculates base fee using EIP-1559 parameters with gas excess tracking.
(4) *Public Input Hash*: Maintains a rolling public input hash using a 255-block ring buffer.

The TaikoAnchor contract uses a "golden touch" address for authorized anchoring transactions.

5.1.3. *Transaction Lists.* The txList contains the RLP-encoded list of all transactions in blocks within a batch. Transaction data can be provided either through calldata or EIP-4844 blobs for more efficient data availability. The protocol calculates a txsHash from the transaction data, which is used for verification purposes.

5.2. **Proposing Blocks.** Block proposing in Taiko Alethia occurs through the TaikoInbox contract's `proposeBatch()` function. The contract supports both permissionless proposing and restricted proposing through an optional inboxWrapper. Batches can contain multiple blocks and support both calldata and blob-based transaction data. Block proposals and proofs operate asynchronously, allowing flexibility in proof submission timing. The TaikoInbox contract performs validation including checking anchor blocks, timestamp progression, and ensuring proposers have sufficient bonds.

5.2.1. *Batch Metadata.* Batches are proposed to the TaikoInbox contract with BatchMetadata containing:

infoHash: A bytes32 hash containing batch information.
proposer: The Ethereum address of the batch proposer.
batchId: A 64-bit unsigned integer batch identifier.
proposedAt: A 64-bit timestamp for when the batch was proposed.

5.2.2. *Block Parameters.* Individual blocks within a batch are described by BlockParams:

numTransactions: The number of transactions in the block (uint16).
timeShift: Time adjustment parameter (uint8).
signalSlots: Array of signal slot hashes for cross-chain communication.

5.2.3. *Batch Validation.* Batch proposals undergo several validation checks:

- Verification that the batch is within allowed fork heights
- Validation of anchor block recency and validity
- Checking transaction data or blob availability
- Ensuring timestamp progression between blocks
- Verifying proposer has sufficient bond
- Limiting the number of blocks per batch

Transaction validation and execution details are handled by the L2 execution layer following standard Ethereum transaction processing rules.

(1) The txList is RLP decodable into a list of transactions, and;
(2) The number of transactions is within the block's specified limit, and;
(3) The sum of all transactions' gasLimit is no larger than the protocol constant $K_{\text{BlockMaxGasLimit}}$, and;
(4) Each and every transaction's signature is valid, i.e. it does not recover to the zero address.

Formally, $V^l(L)$ is defined as:

$$\begin{aligned}
(1)\ V^l(L) \quad \equiv \quad & \texttt{NOERR}(T \equiv \texttt{RLP}'(L)) \quad \wedge \\
& \|T\| \leq \texttt{numTransactions} \quad \wedge \\
& \Big(\sum_{j=0}^{\|T\|-1} T[j]_g\Big) \leq K_{\text{BlockMaxGasLimit}} \quad \wedge \\
& \prod_{j=0}^{\|T\|-1} (T[j]_g \geq K_{\text{TxMinGasLimit}}) \quad \wedge \\
& \prod_{j=0}^{\|T\|-1} (\texttt{NOERR}(\texttt{ECRECOVER}(T[j]) \neq 0))
\end{aligned}$$

Where $\texttt{NOERR}(S)$ is a catch-error function that returns `False` if statement S throws an error; $\texttt{RLP}'$ is the RLP decoding function; T_g is a transaction's gasLimit;

5.2.4. *Mapping.* A proposed block where both $V^b(\dot{B})$ and $V^l(\dot{B}_L)$ hold true will map to an actual Taiko block.

Taiko blocks are identical to Ethereum blocks, as defined by the Ethereum Yellow Paper[11]:

$$\begin{aligned}
(2) \qquad B_H \quad & \equiv \quad (H_p, H_o, H_c, H_r, H_t, H_e, H_b, H_d, \\
& \qquad H_i, H_l, H_g, H_s, H_x, H_m, H_n) \\
(3) \qquad B_U \quad & \equiv \quad [] \\
(4) \qquad B \quad & \equiv \quad (B_H, B_T, B_U)
\end{aligned}$$

Where H_p is the block's parentHash, H_o is the ommersHash, H_c is the beneficiary, H_r is the stateRoot, H_t is the transactionsRoot, H_e is the receiptsRoot, H_b is the logsBloom, H_d is the difficulty, H_i is the block number, H_l is the gasLimit, H_g is the gasUsed, H_s is the timestamp, H_x is the extraData, H_m is the mixHash, H_n is the nonce; B_T a series of the transactions; and B_U is a list of ommer block headers but this list will always be empty for Taiko because there is no Proof-of-Work.

A proposed block can only be mapped to a Taiko block in a *Mapping Metadata* which is the world state $\boldsymbol{\sigma}$:

$$\boldsymbol{\sigma} \equiv (\boldsymbol{\delta}, h[1..256], d, i, \theta)$$

Where $\boldsymbol{\delta}$ is the state trie, $h[1..256]$ are the most recent 256 ancestor block hashes, d is Taiko's chain ID, θ is the anchor transaction, and i is the block number.

Now we can define the block mapping function M as:

$$\begin{aligned}
(5) \qquad M(B) \quad & \equiv \quad M(H, T, U), \\
& \equiv \quad M(\boldsymbol{\delta}, h[1..256], d, i, \theta, \dot{B},) \\
& \equiv \quad M(\boldsymbol{\delta}, h[1..256], d, i, \theta, C, L)
\end{aligned}$$

such that:

$$
\begin{aligned}
&\texttt{CHAINID} = \quad \land \\
&\texttt{NUMBER} = i \quad \land \\
&U = [] \quad \land \\
&T = \theta :: V^t(\texttt{RLP}'(L)) \quad \land \\
&H_p = h(1) \quad \land \\
&H_o = K_{\text{EmptyOmmersHash}} \quad \land \\
&H_c = C_c \quad \land \\
&H_d = 0 \quad \land \\
&H_i = i \quad \land \\
&H_l = C_l + K_{\text{AnchorTxGasLimit}} \quad \land \\
&H_s = C_s \quad \land \\
&H_x = C_x \quad \land \\
&H_m = C_m \quad \land \\
&(H_r, H_t, H_e, H_l, H_g) = \Pi(\boldsymbol{\sigma}, (T_0, T_1, \ldots))
\end{aligned} \tag{6}
$$

Where Π is the block transition function; :: is the list concatenation operator; V^t is the *"Initial Tests of Intrinsic Validity"* function defined in the Transaction Execution section of the Ethereum Yellow Paper. To avoid confusion, in this document, we call V^t the *Metadata Validity* function.

$V^t(\texttt{RLP}'(L))$ yields a list of transactions that pass the tests; transactions that don't pass the tests are ignored and will not be part of the actual L2 block. Note that it is perfectly valid for $V^t(\texttt{RLP}'(L))$ to return an empty list.

5.3. **Anchor Transaction.** The anchor transaction is executed by the TaikoAnchor contract and serves as a bridge between L1 and L2 state. Only the "golden touch address" is authorized to execute anchor transactions.

The anchor transaction is required to be the first transaction in a Taiko block (which is important to make the block deterministic). The anchor transaction is currently used as follows:

(1) *L1 Block Synchronization*: Synchronizes L1 block data to L2, storing the L1 state root as a signal and updating the last synced L1 block height.
(2) *Public Input Hash Management*: Maintains and updates a public input hash that tracks the integrity of block-related inputs and prevents duplicate anchor transactions.
(3) *Gas Management*: Calculates and updates EIP-1559 base fee parameters, managing gas excess and target values.
(4) *Block Hash Storage*: Stores block hashes in a mapping for cross-layer verification.

The anchor transaction should entail:

(1) Persisting l1Height C_a and l1Hash C_h, data inherited from L1, to the storage trie. These values can be used by bridges to validate cross-chain messages (see Section 8).
(2) Comparing ρ_{i-1}, the *public input hash* stored by the previous block, with $\texttt{KEC}(i-1, d, h[2..256])$. The anchor transaction will throw an exception if such comparison fails. The protocol requires the anchor transaction to execute successfully and will not accept a proof for a block that fails to do so. Note that the genesis block has $\rho_0 \equiv \texttt{KEC}(0, d, [0, \ldots, 0])$.
(3) Persisting a new public input hash

With anchoring, the block mapping function M can be simplified to:

$$
\begin{aligned}
B &\equiv (H, T, U), \\
&\equiv M(\boldsymbol{\delta}, \theta, \dot{B},) \\
&\equiv M(\boldsymbol{\delta}, \theta, C, L)
\end{aligned} \tag{7}
$$

5.3.1. *Construction of Anchor Transactions.* All anchor transactions are signed by a *Golden Touch* address with a revealed private key.

Anchor transactions are constructed by Taiko Alethia L2 nodes as follows:

$$
\begin{aligned}
&\theta_x = 0 \quad \land \\
&\theta_n = \boldsymbol{\delta}[K_{\text{GoldenTouchAddress}}]_n + 1 \quad \land \\
&\theta_p = 0 \quad \land \\
&\theta_g = K_{\text{AnchorTxGasLimit}} \quad \land \\
&\theta_t = K_{\text{GoldenTouchAddress}} \quad \land \\
&\theta_v = 0 \quad \land \\
&(\theta_r, \boldsymbol{\delta}_s) = \texttt{K1ECDSA}(\boldsymbol{\delta}, K_{\text{GoldenTouchPrivateKey}})
\end{aligned} \tag{8}
$$

Where K1ECDSA is the ECDSA[12] signing function with the internal variable k set to 1, which guarantees the transaction's signature to only depend on the transaction data itself and is therefore deterministic.

According to the ECDSA's spec, when k is 1, θ_r must equal $\mathrm{G_x}$, the value of the x-coordinate of the base point on the SECP-256k1 curve. The TaikoInbox contract verifies this assertion.

5.4. **Proving Blocks.** Proofs need to be submitted to Ethereum so that batches of blocks can be verified on-chain. We stress again that all proposed batches of blocks are verified immediately because proposed batches are deterministic and cannot be reverted. The prover has *no* impact on the post-batch state. The proof is only required to prove to the TaikoInbox smart contract that the L2 state transitions and the rollup protocol rules are fully constrained. These on-chain verified L2 states are made accessible to other smart contracts (and indirectly to other L2s) so they can have access to the full L2 state, which is critical for bridges (see Section 8).

Batches of blocks can be proven in parallel and so proofs may be submitted out-of-order. As a result, when proofs are submitted for batches where the parent batch is not yet verified, we cannot know if the proof is for the correct state transition. A proof on its own can only verify that the state transition from one state to another state is done correctly, not that the initial state is the correct one. As such, proving a batch of blocks can create a State Transition which is an attestation that the batch in question transits from a prover-selected parent batch to a correctly calculated new world state. It is important to note that there is only a single valid state transition per batch: the state transition that transitions from the last on-chain verified batch to the next *valid* proposed batch. All other state transitions use an incorrect pre-batch state.

A State Transition is a simple structure with 3 elements:

$$
E \equiv (H_p, H_h, H_s) \tag{9}
$$

where H_p is the parent hash, H_h is the block hash, and H_s is the state root. External verifier contracts handle multiproving validation for batches of blocks through the TaikoInbox contract's verification system.

5.4.1. *Invalid Blocks.* Invalid blocks are those that fail to pass the Intrinsic Validity Function V^l (see Section 5.2.3) or contain malformed transaction data. Batches containing invalid blocks will fail the proving process through the multiproving system, where verifier contracts validate batch execution and reject batches that do not conform to protocol rules.

The proving system operates with asynchronous proof submission. The TaikoInbox contract's `proveBatches()` function allows proving multiple batches with a single aggregated proof.

Key characteristics of the proving system:

(1) *Asynchronous Proving*: Block proposals and proofs are asynchronous - proofs can be submitted after batch proposal.
(2) *Batch Proving*: Multiple batches can be proved together using aggregated proofs.
(3) *Verification*: Uses an external verifier contract to validate proofs, with proofs presumed error-free and thoroughly validated.
(4) *Proving Window*: Proofs must be submitted within a proving window timeframe.

The TaikoInbox contract manages state transitions and tracks batch verification status. Conflicting proofs can trigger contract pausing for resolution. The proving system delegates subproof and multiproving management to verifier contracts.

5.5. On-chain Verification of Blocks.

The verification process operates through the TaikoInbox contract's `verifyBatches()` function, which validates multiple batch transitions using aggregated proofs. The process involves batch validation, proof verification, and state transition management.

Verification workflow:

(1) *Batch Validation*: Each batch metadata is verified against stored batch data, including metaHash matching and batch existence checks.
(2) *Transition Management*: New transitions are created or existing ones overwritten based on parent hash matching, with transition IDs tracked in mappings.
(3) *Proof Verification*: The designated verifier contract validates the aggregated proof through the IVerifier interface's `verifyProof()` function.
(4) *State Updates*: Upon successful verification, transition states are updated with block hashes, state roots, and prover information.
(5) *Proving Window*: Transitions track whether they were created within the proving window, affecting prover attribution and verification status.

Conflict resolution occurs when multiple proofs target the same parent hash but produce different block hashes or state roots. In such cases, the conflicting transition's block hash is set to zero, a ConflictingProof event is emitted, and the contract pauses operations. After processing all transitions, successful verification triggers BatchesProved events and potential batch finalization through the verification process.

The verifying process takes place when a batch is proposed or proven - the `verifyBatches()` function will be called, and verify up to N batches, which is configurable.

A batch is only verifiable if the previous batch is also verified. Thus, batches can be proven out of order, but only verified in order.

6. Taiko Alethia Based Preconfirmations

The aforementioned Taiko Alethia based preconfirmation system evolves through two phases, progressively decentralizing from a whitelisted operator model to permissionless L1 validator participation.

6.1. Phase 1: Whitelisted Operators.

Phase 1 implements preconfirmations through whitelisted operators:

6.1.1. *PreconfWhitelist Contract.* The PreconfWhitelist contract manages authorized preconfirmation operators:

- **Operator Management:** Tracks operators with activation and inactivation epochs
- **Epoch-Based Selection:** Selects operators for specific epochs using beacon block root randomness
- **Access Control:** Allows adding/removing operators by owner or designated "ejecters"
- **Consolidation:** Implements a mechanism to consolidate the operator list by removing inactive operators

6.1.2. *PreconfRouter Contract.* The PreconfRouter provides controlled batch proposal routing:

- **Batch Proposal:** Routes batch proposals from authorized preconfirmers to the TaikoInbox
- **Authorization:** Ensures only whitelisted preconfirmers or fallback addresses can propose
- **Validation:** Optionally validates expected last block ID before accepting proposals
- **Handover Slots:** Configured with 8 handover slots for operator transitions

6.1.3. *Integration with TaikoInbox.* The TaikoInbox contract includes support for preconfirmation through:

- An optional inboxWrapper address that can restrict batch proposals
- Support for both permissionless and restricted proposing modes

In Phase 1, authorized operators can propose batches through the router, with the whitelist managing operator lifecycle and selection based on epochs.

6.2. Phase 2: L1 Validator Opt-in with Slashing.

Phase 2 transitions to a permissionless model where Ethereum L1 validators can opt into providing preconfirmations:

6.2.1. *Validator Registration.* L1 validators voluntarily register as preconfirmers by:

- **Staking ETH:** Validators deposit ETH collateral into a slashing contract to align economic incentives
- **Opt-in Mechanism:** Validators signal their intent to participate in preconfirmations

- **Infrastructure Requirements:** Running additional preconfirmation infrastructure alongside their validator client

6.2.2. *Economic Security Model.* The Phase 2 model enforces commitments through economic penalties:

- **Slashing Conditions:** Validators face slashing for:
  - Failing to include preconfirmed transactions in proposed blocks
  - Providing invalid or conflicting preconfirmations
- **Reward Distribution:** Validators earn fees from users for providing preconfirmations
- **Stake Requirements:** Minimum ETH stake ensures meaningful economic security

6.2.3. *Decentralization Benefits.* Phase 2 removes the whitelist requirement, enabling:

- **Permissionless Participation:** Any L1 validator can become a preconfirmer
- **Geographic Distribution:** Global validator set provides worldwide coverage
- **Censorship Resistance:** No central authority controls preconfirmer selection
- **Aligned Incentives:** Validators naturally integrate preconfirmations with block production

6.2.4. *Technical Implementation.* Phase 2 requires new infrastructure:

- **Slashing Contract:** Manages validator stakes and enforces penalties
- **Registration Contract:** Handles validator opt-in and stake management
- **Coordination Layer:** Enables validators to discover and relay preconfirmations
- **Monitoring System:** Tracks validator behavior and triggers slashing when violations occur
- **EIP-7917 on L1**[4]: Fusaka will enable EIP-7917 on L1 to expose a stable proposer lookahead to the execution layer

6.3. **Migration from Phase 1 to Phase 2.** The transition strategy includes:

- **Parallel Operation:** Both systems run simultaneously during transition period
- **Gradual Migration:** Validators progressively opt in as infrastructure matures
- **Whitelist Sunset:** Phase 1 whitelist deprecated once sufficient validators participate

This phased approach enables immediate preconfirmation benefits while building toward full decentralization with L1 validator participation and cryptoeconomic security.

6.4. **Sidecars.** The preconfirming is done by sidecars running in parallel to a Taiko execution and consensus layer.

The sidecars interact with the suite of contracts described in Section 5.1.

The job of the sidecar is to:

- Wait for their epoch to be active
- Accept transactions from users via the mempool
- Create preconfirmation blocks with those transactions, which will be gossiped via the Taiko node connected to the sidecar
- When a batch of blocks is deemed profitable or the window is over, submit them onchain for L1 finality
- Handover the preconfirmation role to the next preconfirmer in the list when their epoch ends

The current sidecars can be found at:

- `https://github.com/gattaca-com/taiko-gateway` - Sidecar implementation by Gattaca
- `https://github.com/NethermindEth/Catalyst` - Sidecar implementation by Nethermind
- `https://github.com/chainbound/taiko-mk1` - Sidecar implementation by Chainbound

When phase 2 rolls out, any L1 validator will be able to run a sidecar and become a preconfirmer, as well as create their own sidecar to fit their needs, as long as it follows the protocol rules.

Block building and preconfirming can be separated into a block building market and a sequencing market, thus L1 validaotors can opt-in to provide preconfirmations, but delegate their rights to the block building market.

7. Taiko L2 Hard Forks

Taiko Alethia's protocol evolution occurs through planned hard forks that introduce significant architectural improvements while maintaining backward compatibility and state continuity.

7.1. **Ontake Fork.** The Ontake fork represents Taiko Alethia's initial mainnet protocol implementation:

7.1.1. *Architecture.*

- **Based Contestable Rollup (BCR):** Original design with tier-based proof system
- **Block-by-Block Processing:** Individual block proposals and verification
- **Contestation Mechanism:** Proofs could be contested within validity windows
- **Core Contract:** TaikoL1.sol managed L1 protocol logic

7.1.2. *Fork Heights.*

- **Mainnet:** Block 538,304
- **Hekla Testnet:** Block 840,512

7.2. **Pacaya Fork.** The Pacaya fork introduces a major architectural shift from block-based to batch-based processing:

7.2.1. *Key Innovations.*

- **Batch-Based Protocol:** Blocks proposed in batches (zero, one, or multiple blocks per batch)
- **Simplified Proving:** Removed proof contestation in favor of multiproving
- **Multiproving System:** Every batch uses multiproving with support for SGX (sgxGeth + sgxReth), SP1, and Risc0 proof types
- **Preconfirmation Support:** Architecture enables based preconfirmations

7.2.2. *Contract Changes.*

- **TaikoInbox:** Replaces TaikoL1.sol as primary L1 contract
- **TaikoAnchor:** Replaces TaikoL2.sol for L2 anchoring
- **ComposeVerifier:** New multiproof verification contract
- **ForcedInclusionStore:** Ensures censorship resistance

7.2.3. *Fork Heights.*

- **Mainnet:** Batch 1,166,000
- **Hekla Testnet:** Batch 1,299,888

7.3. **Migration Architecture.** Taiko Alethia implements sophisticated fork migration through a router pattern:

7.3.1. *Fork Router System.*

- **PacayaForkRouter:** Routes function calls between Ontake and Pacaya implementations
- **Delegatecall Pattern:** Maintains state while switching logic contracts
- **Function Mapping:** Legacy functions route to old contracts, new functions to updated ones

7.3.2. *Storage Compatibility.*

- **State Preservation:** New contracts maintain storage layout compatibility
- **Seamless Transition:** No state migration required during fork activation
- **Backward Compatibility:** Historical data remains accessible

7.4. **Fork Measurement.** Different forks use different measurement units:

- **Ontake:** Measured by block numbers
- **Pacaya and beyond:** Measured by batch IDs

This change reflects the architectural shift from block-based to batch-based processing, enabling more efficient L1 operations and supporting advanced features like preconfirmations.

8. Cross-Chain Communication

Taiko Alethia enables third parties to develop cross-chain bridges. To facilitate this, the protocol ensures that a subset of L1 block hashes are accessible from L2 smart contracts and a subset of L2 block hashes are also accessible from the TaikoInbox smart contract. These block hashes can be used to verify the validity of cross-chain messages in standard smart contracts. Taiko Alethia does not have to provide any bridging solutions itself, as the supporting core functionality are ready for others to build upon. An exception to this is the Ether bridge which requires special handling (see Section 8.1).

On Ethereum, the TaikoInbox contract persists the height and hash of the L2 blocks. On Taiko Alethia, the anchor function in the TaikoAnchor contract is used to persist the height and block hash of the previous Ethereum block (from when the L2 block was proposed), as well as the previous L2 block hash (which allows L2 smart contracts to easily fetch the full history of L2 block hashes).

8.1. **Ether on L2.** The Taiko Alethia Ether bridge allows users to bridge Ether from and to Taiko Alethia. 2^{128} Ether is minted to a special vault contract called the TokenVault in the genesis block. When a user deposits Ether to L2, the same amount of Ether will be transferred from the TokenVault to the user on L2. When a user withdraws some Ether from L2, Ether on L2 will be transferred back to TokenVault (no L2 Ether will ever be burnt).

A small amount of Ether will also be minted to a few EOAs to bootstrap the L2 network, otherwise nobody would be able to transact. To make sure the Ether bridge is solvent, a corresponding amount of Ether will be deposited to the Ether bridge on L1.

9. Future Improvements

9.1. **Block Data Compression.** A big part of the cost of a rollup block is the data that is required to be stored on L1. It has been shown that standard general compression schemes like DEFLATE[13] work well on transaction data. It is possible to implement these schemes efficiently in a circuit and so the data published on L1 can be compressed while the circuits can decompress the data again. This will make it possible to reduce the amount of data that needs to be published on L1, significantly reducing costs.

9.2. **Shasta Fork Improvements.** The Shasta fork represents the next major protocol upgrade, currently in development:

9.2.1. *Planned Features.* Planned features include:

- **Optimized Batch Processing:** Optimization of batch proposal, proving and verification that will reduce the cost of the protocol by 10x or more
- **Protocol Simplifications:** Continued removal of complex logic, and pushing complexity off-chain
- **Bond Processing on L2:** Processing bonds on L2 reduces costs on L1
- **Phase 2 preconfirmation preparation:** Shasta will make the protocol ready for the future upgrade to permisionless preconfirmations(Phase 2)

9.2.2. *Phase 2 preconfirmations.* Phase 2 of preconfirmations will enable any L1 validator to participate, improving liveness and censorship resistance as explained in Section 6. It will also enable slashing for safety faults, providing economic guarantees to users.

References

[1] https://ethereum-magicians.org/t/a-rollup-centric-ethereum-roadmap/4698
[2] https://eips.ethereum.org/EIPS/eip-2028
[3] https://eips.ethereum.org/EIPS/eip-4844
[4] https://eips.ethereum.org/EIPS/eip-7917
[5] J. Poon, V. Buterin; https://plasma.io/plasma-deprecated.pdf
[6] Vitalik Buterin; https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477
[7] Barry Whitehat; https://ethresear.ch/t/roll-up-roll-back-snark-side-chain-17000-tps/3675
[8] https://github.com/privacy-scaling-explorations
[9] https://github.com/taikoxyz/taiko-mono/tree/main/packages/protocol
[10] https://ethereum.org/en/developers/docs/mev
[11] https://ethereum.github.io/yellowpaper/paper.pdf

[12] https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
[13] https://en.wikipedia.org/wiki/Deflate
[14] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md
[15] https://dl.acm.org/doi/10.1145/3479722.3480987
[16] Justin Drake; https://ethresear.ch/t/based-rollups-superpowers-from-l1-sequencing/15016
[17] https://docs.arbitrum.io/sequencer#unhappyuncommon-case-sequencer-isnt-doing-its-job
[18] https://docs.espressosys.com/sequencer/espresso-sequencer-architecture/readme
[19] https://ethresear.ch/t/based-preconfirmations/17353
[20] https://ethresear.ch/t/multi-proofs-for-rollup-security/16248
[21] https://ethresear.ch/t/cross-rollup-composability/15288

Appendix A. Terminology

Anchor Transaction: The first transaction in every Taiko L2 block to perform data validation and L1-to-L2 communication.

State Transition: A data structure to capture a block's proving result based on a prover-chosen parent block.

Golden Touch Address: An address with a revealed private key to transact all anchor transactions.

Appendix B. Ethereum Upgrades on Taiko

Name	Status
EIP-606 – Hardfork Meta: Homestead	Enabled
EIP-779 – Hardfork Meta: DAO Fork	Disabled
EIP-150 – Gas cost changes for IO-heavy operations	Enabled
EIP-155 – Simple replay attack protection	Enabled
EIP-158 – State clearing	Enabled
EIP-609 – Hardfork Meta: Byzantium	Enabled
EIP-1013 – Hardfork Meta: Constantinople	Enabled
EIP-1716 – Hardfork Meta: Petersburg	Enabled
EIP-1679 – Hardfork Meta: Istanbul	Enabled
EIP-2387 – Hardfork Meta: Muir Glacier	Disabled
Berlin Network Upgrade	Enabled
London Network Upgrade	Disabled
Arrow Glacier Network Upgrade	Disabled
EIP-3675 – Upgrade consensus to Proof-of-Stake	Enabled
Shanghai Network Upgrade	Disabled (future)
Cancun Network Upgrade	Disabled (future)

Appendix C. Protocol Constants

Name	Description	Value
K_{ChainID}	Taiko Alethia's chain ID	167000 (mainnet)
$K_{\text{BatchRingBufferSize}}$	Ring buffer size for batch storage	360,000
$K_{\text{MaxBlocksPerBatch}}$	Maximum blocks per batch	768
$K_{\text{LivenessBondBase}}$	Base liveness bond per batch	25 TAIKO
$K_{\text{ProvingWindow}}$	Time window for proof submission	2 hours
$K_{\text{CooldownWindow}}$	Cooldown period after proving	2 hours
$K_{\text{BlockMaxGasLimit}}$	Maximum gas limit per block	240,000,000
$K_{\text{MaxUnverifiedBatches}}$	Maximum unverified batches	324,000
$K_{\text{MaxBatchesToVerify}}$	Maximum batches verified per operation	8
$K_{\text{StateRootSyncInterval}}$	Block interval for state root sync	4
$K_{\text{MaxAnchorHeightOffset}}$	Maximum L1 block height offset	96
$K_{\text{MaxSignalsToReceive}}$	Maximum cross-chain signals	16
$K_{\text{TxMinGasLimit}}$	Minimum gas limit per transaction	21,000

Name	Value
$K_{\text{GoldenTouchAddress}}$	0x0000777735367b36bC9B61C50022d9D0700dB4Ec
$K_{\text{GoldenTouchPrivateKey}}$	0x92954368afd3caa1f3ce3ead0069c1af414054aefe1ef9aeacc1bf426222ce38
$K_{\text{AnchorTxGasLimit}}$	1,000,000
$K_{\text{EmptyOmmersHash}}$	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347