# Taler Blockchain

**Technical Whitepaper**

## Table of Contents

---

## 1. Introduction

Public blockchains have demonstrated the viability of decentralized computation, enabling applications ranging from digital currencies and decentralized finance to on-chain governance and identity systems. However, many existing networks suffer from unpredictable inflationary models, complex reward mechanisms, limited smart contract capabilities, and governance systems that either centralize power or see negligible participation.

Taler addresses these challenges by building on the Parity Substrate framework (version 1.3.0) — a modular, extensible toolkit for constructing application-specific blockchains. Substrate provides production-ready implementations of networking (libp2p), consensus (BABE/GRANDPA), and a flexible runtime environment compiled to WebAssembly. This foundation allows Taler to focus its innovation on the areas that matter most to its stakeholders: a transparent and predictable staking economy, robust on-chain governance, and a capable smart contract environment.

Key design principles of the Taler blockchain include:

- **Deterministic staking rewards**: A fixed annual reward model replaces unpredictable inflation curves, giving stakers clear expectations about their returns.
- **Voluntary time-lock incentives (Vivid Staking)**: Participants who commit to longer lock periods receive proportionally higher yields, aligning long-term incentives without mandating participation.
- **Hard issuance cap**: Total token supply is bounded, ensuring that the staking reward model converges toward equilibrium as the cap is approached.
- **Forkless upgrades**: The network can evolve its logic through on-chain governance and runtime upgrades without requiring coordinated hard forks.

- **WebAssembly smart contracts**: Developers can deploy general-purpose smart contracts using the ink! framework and any Wasm-compatible toolchain.
- **Multi-layered governance**: Democracy, council, technical committee, and treasury modules work in concert to enable decentralized decision-making.

---

## 2. Technical Architecture

### 2.1 Runtime and Block Structure

The Taler blockchain compiles its state transition function — the *runtime* — into a WebAssembly (Wasm) blob. This runtime can be executed either natively (when the on-disk binary matches the on-chain version) or within an embedded Wasm virtual machine. This dual-execution model enables forkless upgrades: when a new runtime is enacted through governance, nodes automatically download and begin executing the updated Wasm code without requiring a software release.

The block structure follows the standard Substrate format:

- **Header** := ParentHash + ExtrinsicsRoot + StorageRoot + Digest + BlockNumber
- **Block** := Header + Extrinsics + Justifications

State roots within the header are stored in a Base-16 Modified Merkle Patricia Trie, providing efficient cryptographic proofs of inclusion and state integrity.

**Extrinsics** represent all data external to the blockchain that is included in a block. This encompasses signed transactions, unsigned transactions, and inherent data (such as timestamps). Every state change on Taler is encoded as an extrinsic, providing a uniform abstraction over individual transfers, batch operations, governance actions, and other state modifications.

Key block parameters include:

| Parameter | Value |
|---|---|
| Block time (target) | 6 seconds |
| Epoch duration | 4 hours (2,400 blocks) |
| Maximum block weight | 2 seconds of computation |
| Normal dispatch ratio | 75% of block weight |
| Maximum block size | 5 MB |

The BlakeTwo256 hashing algorithm is used throughout the runtime for all Merkle tree computations, storage key derivation, and extrinsic hashing.

### 2.2 Pallets (Modules)

Taler's runtime is composed of a set of *pallets* — modular Rust libraries compiled to Wasm that define the blockchain's core logic. Each pallet manages a specific domain of

functionality and has access to the full computational capabilities of the runtime, subject to weight limits that ensure blocks can always be produced within the target block time.

The following table enumerates all pallets included in the Taler runtime:

| Pallet | Description |
|---|---|
| frame_system | Core system pallet; manages accounts, block metadata, events, and runtime versioning. |
| pallet_babe | Implements the BABE (Blind Assignment for Blockchain Extension) block authoring consensus. |
| pallet_grandpa | Implements the GRANDPA finality gadget for deterministic block finality. |
| pallet_timestamp | Provides on-chain time tracking, driven by validator-submitted inherents. |
| pallet_authorship | Tracks the author of each block for reward distribution purposes. |
| pallet_indices | Assigns short numeric indices to accounts for human-readable addressing. |
| pallet_balances | Manages native token (TAL) balances, transfers, locks, holds, and reserves. |
| pallet_transaction_payment | Computes and charges transaction fees based on weight and byte length. |
| pallet_staking | **Modified.** Handles validator election, nomination, bonding, and a custom deterministic reward model with Vivid Staking. |
| pallet_session | Manages session keys and validator set rotation across sessions. |
| pallet_election_provider_multi_phase | Provides the NPoS election algorithm using Sequential Phragmén with off-chain and on-chain phases. |
| pallet_bags_list | Maintains an efficiently ordered list of voters for the staking election. |
| pallet_democracy | Enables stakeholder referenda with coin-weighted and time-lock-weighted voting. |
| pallet_collective (Instance 1) | Implements the **Council** — an elected body with privileged governance actions. |
| pallet_collective (Instance 2) | Implements the **Technical Committee** — experts who can fast-track or cancel proposals. |

| Pallet | Description |
|---|---|
| pallet_elections_phragmen | Manages council member elections via the Phragmén algorithm. |
| pallet_membership | Manages membership of the Technical Committee, controlled by Council vote. |
| pallet_treasury | Collects fees and slashing penalties; disburses funds through approved proposals. |
| pallet_bounties | Enables curated bounty programs funded by the treasury. |
| pallet_tips | Provides a tipping mechanism for ad-hoc treasury disbursements. |
| pallet_contracts | Enables deployment and execution of Wasm smart contracts (ink! compatible). |
| pallet_assets | Provides a framework for creating and managing fungible tokens (analogous to ERC-20). |
| pallet_nfts | Provides non-fungible token (NFT) creation and management. |
| pallet_nft_fractionalization | Enables fractional ownership of NFTs by splitting them into fungible asset shares. |
| pallet_identity | On-chain identity registration and verification through third-party registrars. |
| pallet_recovery | Social recovery mechanism using trusted friends to regain access to accounts. |
| pallet_vesting | Manages time-locked token release schedules; used for premine lock-up. |
| pallet_proxy | Allows one account to act on behalf of another with configurable permission filters. |
| pallet_multisig | Enables multi-signature accounts requiring M-of-N approvals. |
| pallet_scheduler | Schedules future execution of dispatchable calls; used by governance for enactment delays. |
| pallet_preimage | Stores large call data (preimages) referenced by hash in governance proposals. |
| pallet_sudo | Provides a superuser account for bootstrap-phase administration. **Removed after network stabilization.** |
| pallet_im_online | Monitors validator liveness via periodic heartbeat messages. |

| Pallet | Description |
|---|---|
| pallet_offences | Records and processes validator misbehavior (equivocation, unresponsiveness). |
| pallet_authority_discovery | Enables validators to discover each other on the network. |
| pallet_utility | Provides batch calls, dispatch_as, and other utility extrinsics. |
| pallet_transaction_storage | Enables on-chain storage of arbitrary data indexed by block. |
| pallet_insecure_randomness_collective_flip | Provides a pseudo-random number source based on historical block hashes. **Insecure; not suitable for adversarial contexts.** |

## 2.3 Consensus

Taler employs a **hybrid consensus architecture** that cleanly separates two concerns — block *production* and block *finality* — into independent but cooperating protocols. This separation is a deliberate design choice: it allows the chain to continue producing blocks optimistically while a more heavyweight finality process runs in parallel, and it enables either protocol to be upgraded independently through governance without disrupting the other.

### *2.3.1 Block Production: BABE*

**BABE (Blind Assignment for Blockchain Extension)** is the block production engine responsible for determining which validator may author a block in each time slot. BABE is inspired by the Ouroboros Praos protocol but differs in its chain selection rule and timing model.

**Time Structure.** BABE divides time into two nested units:

- **Slots**: Fixed 6-second intervals, each representing an opportunity for a block to be produced.
- **Epochs**: Groups of 2,400 consecutive slots (~4 hours). Epoch boundaries trigger rotation of the randomness seed and, via the session module, potential rotation of the active validator set.

**Primary Slot Assignment via VRF.** At the beginning of each epoch, every validator independently evaluates a **Verifiable Random Function (VRF)** for each slot in the epoch. The VRF takes three inputs:

1. The **epoch randomness seed** $r_e$ (agreed upon by all nodes two epochs in advance),
2. The **slot number** $\text{slot}_i$,
3. The validator's **private key** $\text{sk}_v$.

The VRF produces a pseudo-random output value and an accompanying cryptographic proof:

$$(\text{out}, \pi) = \text{VRF}_{\text{sk}_v}(r_e \parallel \text{slot}_i)$$

If out $< \tau_v$, where $\tau_v$ is a threshold proportional to the validator's stake weight and the protocol parameter $c$ (where $1 - c$ is the probability of a slot being empty), then the validator is a **primary slot leader** for that slot and is authorized to produce a block. The proof $\pi$ is included in the block header, allowing any other node to verify the claim using only the validator's public key — without learning slot assignments in advance.

Because the VRF depends on each validator's unique private key, slot assignments are **private until the block is published**. No validator can predict another's assignments, which protects against targeted denial-of-service attacks on upcoming block producers.

Taler uses the **Schnorrkel/sr25519** signature scheme (based on the Ristretto construction over the Ed25519 curve) for its VRF implementation. This scheme provides the necessary properties: deterministic output, proof generation, efficient verification, and resistance to output malleability.

**Secondary Slot Assignment.** Since the VRF-based primary assignment is probabilistic, some slots may have no primary leader (empty slots) and others may have multiple leaders (temporary forks). To guarantee that every slot produces at least one block and maintain a consistent block time, BABE uses a **deterministic secondary assignment** as a fallback:

$$\text{secondary\_leader} = \text{blake2}_{256}(r_e \parallel \text{slot}_i) \bmod |\text{authorities}|$$

This round-robin-like mechanism ensures a block is always produced. However, secondary blocks are treated as lower priority: the fork choice rule favors chains with more primary (VRF-selected) blocks. Taler uses the SecondaryPlain variant for secondary slots.

**Epoch Randomness.** The randomness seed for epoch $e_m$ (where $m > 2$) is derived from all VRF outputs produced in epoch $e_{m-2}$:

$$r_m = H(m \parallel \text{VRF}_1 \parallel \text{VRF}_2 \parallel \cdots \parallel \text{VRF}_k)$$

where $H$ is a cryptographic hash function and $\text{VRF}_1 \ldots \text{VRF}_k$ are the VRF outputs from all BABE primary blocks in the reference epoch. The two-epoch look-ahead ensures that validators must register their keys well before they can influence the randomness, mitigating grinding attacks.

**Fork Choice Rule.** When forks occur (due to multiple primary leaders in the same slot or network partitions), BABE resolves them as follows:

1. **Always build on the chain that GRANDPA has finalized.** Any chain that does not include the last finalized block is discarded.
2. Among chains extending the last finalized block, **select the chain with the most primary blocks** (i.e., VRF-produced blocks, not secondary fallback blocks).

3. In the event of a tie, select the **longest chain**.

This weight-based rule provides probabilistic finality on its own — even if GRANDPA were temporarily stalled — because an adversary controlling less than half of the stake cannot produce more primary blocks than honest validators over time.

**Relative Time Synchronization.** BABE does not rely on any centralized time service (NTP). Instead, validators synchronize their local clocks using a **relative time protocol**: each validator records the arrival times and slot numbers of received blocks, projects future slot boundaries using the known slot duration, and takes the **median** of these projections to calibrate its local clock. This approach tolerates moderate clock drift between validators while remaining resistant to manipulation by any single party.

- **BABE Configuration Parameters (Taler)**

| Parameter | Value |
|---|---|
| Slot duration | 6,000 ms |
| Epoch duration | 2,400 slots (~4 hours) |
| Primary probability $(c)$ | 1/4 (1 in 4 slots expected to have a primary on average) |
| Secondary slot mode | SecondaryPlain |
| VRF key type | Schnorrkel/sr25519 |
| Epoch change trigger | External (managed by session pallet) |

### *2.3.2 Block Finality: GRANDPA*

**GRANDPA (GHOST-based Recursive ANcestor Deriving Prefix Agreement)** is the finality gadget that provides **deterministic, provable finality** for the Taler blockchain. While BABE provides probabilistic finality through the fork choice rule, GRANDPA delivers a much stronger guarantee: once a block is finalized, it can never be reverted, and this fact can be proven cryptographically to any party (including light clients and bridges) using the finality justifications attached to blocks.

**Design Philosophy.** GRANDPA's key innovation, distinguishing it from other BFT finality mechanisms such as Tendermint or Casper FFG, is that **validators vote on chains, not individual blocks**. Votes are applied transitively: a vote for block $B$ at height $h$ is implicitly a vote for all ancestors of $B$. This allows GRANDPA to finalize an entire chain of blocks — potentially dozens or hundreds — in a single voting round, rather than requiring a separate consensus round for each block. This property is critical for performance: the $O(n^2)$ message complexity of BFT agreement is amortized across many blocks rather than repeated per block.

**The GHOST Voting Rule.** GRANDPA uses a $\frac{2}{3}$**-GHOST function** $g(V)$ to process a set of votes $V$. Starting from the genesis block, the function inductively follows child blocks as long as a single child has votes from more than $\frac{2}{3}$ of the total validator weight (counting both direct votes and votes for descendants). The function returns the highest block in the chain for which this supermajority condition holds.

Formally, for a set of votes $V$:

$g(V) =$ the block $B$ with highest block number such that $V$ has a supermajority for $B$

where a supermajority for block $B$ means that validators whose votes are for blocks $\geq B$ (i.e., $B$ or any descendant of $B$), plus any equivocating validators, constitute at least $\frac{n+f+1}{2}$ of the $n$ total voters (where $f$ is the maximum tolerated Byzantine voters, with $f < \frac{n}{3}$).

A crucial property: if all honest validators see different subsets of the same overall vote set, the $g$ function applied to these subsets will always return blocks on the **same chain**. This ensures that even under partial network visibility, honest participants converge.

**Round Structure.** GRANDPA operates in sequential rounds. Each round $r$ proceeds through the following phases:

1. **Primary Broadcast.** A designated primary validator (rotating each round) broadcasts its estimate of the last block that could have been finalized in the previous round, $E_{r-1,v}$. This estimate serves as a coordination hint — if the primary is honest and the network is functioning, all honest validators will converge on the same prevote target.
2. **Prevote Phase.** Each validator waits until either time $t_{r,v} + 2T$ has elapsed (where $T$ is the estimated maximum one-way network delay) or the previous round is completable. It then broadcasts a **prevote** for the head of the best chain containing the primary's estimate (or its own estimate if the primary's block doesn't satisfy certain conditions). The prevote represents the validator's view of the longest chain that should eventually be finalized.
3. **Precommit Phase.** Each validator waits until $g(V_{r,v}) \geq E_{r-1,v}$ (ensuring the prevote GHOST result extends the previous round's estimate) and one of several timing or completability conditions is met. It then broadcasts a **precommit** for $g(V_{r,v})$ — the highest block supported by a $\frac{2}{3}$ supermajority of prevotes.
4. **Finalization.** When a validator observes that $g(C_{r,v})$ (the $\frac{2}{3}$GHOST of precommits) is later than its last finalized block and the prevotes have a supermajority, it **finalizes** that block. A commit message containing the block hash and a set of precommit signatures is broadcast to the network as a compact, independently verifiable proof of finality.

**Safety (Accountable).** GRANDPA provides **accountable safety**: if two conflicting blocks are ever finalized (a safety violation), it is always possible to identify at least $f + 1$ Byzantine validators who equivocated (cast conflicting votes). This is achieved through a constructive challenge procedure: by collecting and comparing vote sets across rounds, the protocol can extract cryptographic evidence of which validators signed contradictory messages. These validators can then be slashed.

The safety property holds **asynchronously** — it does not depend on any network timing assumptions. Even if the network is arbitrarily delayed, honest validators will never finalize conflicting blocks as long as fewer than $\frac{1}{3}$ of validators are Byzantine.

**Liveness.** Under the **partially synchronous network model** (messages between honest validators are delivered within time $T$ after some unknown Global Stabilization Time), GRANDPA guarantees progress. Specifically:

- If the primary of round $r$ is honest and the network is synchronous, all honest validators will finalize at least one new block within $6T$ of the round start.
- If the primary is Byzantine, the round still completes (is "completable") within bounded time, and the next round begins with a new primary.
- No round can deadlock: even in fully asynchronous conditions, as long as $\frac{2}{3}$ of validators are honest, every round eventually becomes completable.

The key to liveness is the **completability** condition: a round is completable when all possible blocks that could be finalized in that round have been determined — meaning the estimate $E_{r,v}$ is well-defined and stable. This allows validators to safely advance to the next round without waiting indefinitely.

**Multi-Block Finalization.** Because validators vote on chains rather than individual blocks, a single GRANDPA round can finalize many blocks simultaneously. If validators have been producing blocks faster than GRANDPA rounds complete (which is the normal case — BABE produces a block every 6 seconds while GRANDPA rounds take somewhat longer), a single round may finalize 2, 5, or even dozens of blocks at once. This is a significant throughput advantage over protocols like Tendermint that must reach agreement on each block individually.

**Interaction with BABE.** The only constraint GRANDPA imposes on block production is that BABE must build on the chain that GRANDPA has finalized. If GRANDPA has finalized block $B$ at height $h$, BABE will only extend chains that include $B$. This coupling is minimal and one-directional: GRANDPA does not dictate which blocks BABE produces, only which finalized prefix they must extend.

This separation enables **optimistic execution**: the network can continue producing and executing blocks even while GRANDPA is still finalizing earlier blocks. If a finalized block is later found to be invalid (e.g., through availability checks in a multi-chain context), GRANDPA can delay finality while block production continues uninterrupted.

- **GRANDPA Configuration Parameters (Taler)**

| Parameter | Value |
|---|---|
| Byzantine fault tolerance | $f < \frac{n}{3}$ |
| Maximum authorities | 100 |
| Maximum nominators rewarded per validator | 256 |

| Parameter | Value |
|---|---|
| Max set ID session entries | BondingDuration × SessionsPerEra |
| Equivocation report longevity | BondingDuration × SessionsPerEra × EpochDuration slots |
| Equivocation handling | On-chain reporting via pallet_offences; slashing via pallet_staking |

### *2.3.3 Equivocation Detection and Reporting*

Both BABE and GRANDPA include on-chain **equivocation detection** mechanisms. An equivocation occurs when a validator produces conflicting outputs for the same consensus context:

- **BABE equivocation**: A validator produces two or more blocks in the same slot.
- **GRANDPA equivocation**: A validator casts two or more conflicting prevotes or precommits in the same round.

Any node that observes an equivocation can submit a **key ownership proof** and the conflicting evidence as an unsigned extrinsic. The pallet_offences module processes the report and triggers the slashing mechanism defined in pallet_staking. Reports are valid for a window defined by ReportLongevity, ensuring that even delayed detection (e.g., after a network partition heals) results in accountability.

The combination of GRANDPA's accountable safety and on-chain equivocation reporting means that **any safety violation is both detectable and punishable**, creating a strong economic disincentive against Byzantine behavior.

### *2.3.4 Liveness Monitoring: I'm Online*

The pallet_im_online module complements the consensus protocols by monitoring validator **liveness**. Each session, validators are expected to send periodic heartbeat messages ("I'm online" signals). Validators who fail to produce blocks and fail to send heartbeats during a session are reported as unresponsive. The severity of the resulting slashing penalty scales with the number of simultaneously unresponsive validators, as described in Section 3.6.1.

### *2.3.5 Session Key Management*

Taler's consensus requires validators to maintain a set of **session keys**, each serving a specific protocol role:

| Key Type | Protocol | Purpose |
|---|---|---|
| BABE key (sr25519) | BABE | VRF evaluation for slot assignment; block signing |
| GRANDPA key (ed25519) | GRANDPA | Prevote/precommit signing for finality votes |
| ImOnline key (sr25519) | ImOnline | Heartbeat message signing |

| Key Type | Protocol | Purpose |
|---|---|---|
| Authority Discovery key | Networking | Validator peer discovery on the p2p network |

Session keys are rotated at session boundaries (every epoch) and are registered on-chain by the validator's **stash account** via a controller account. This separation ensures that the long-lived stash key (which controls bonded funds) never needs to be exposed on a hot validator machine. If a session key is compromised, it can be rotated without moving funds, limiting the window of vulnerability.

### 2.4 Accounts and Balances

Taler uses an account-based model. Accounts are identified by their public key (32-byte AccountId derived from sr25519 or ed25519 key pairs) and may optionally be referenced by a numeric index.

The custom SS58 address prefix for Taler is **10960**, giving all Taler addresses a distinctive visual format.

The balance system enforces the following constraints:

| Parameter | Value | Description |
|---|---|---|
| Existential Deposit | $10 \times$ TMILLICENTS | Minimum balance to keep an account alive. Accounts falling below this threshold are reaped. |
| Max Locks | 50 | Maximum number of concurrent balance locks per account (used by staking, democracy, vesting). |
| Max Reserves | 50 | Maximum number of named balance reserves per account. |
| Max Holds | 5 | Maximum number of balance holds per account. |
| Max Freezes | 1 | Maximum number of balance freezes per account. |

Balances are partitioned into *free*, *reserved*, and *held* portions. Free balance participates in transfers and fee payment. Reserved balance is earmarked by system operations (e.g., identity deposits, governance deposits) and can be slashed. Held balance supports the newer hold-based mechanism used by preimages and other pallets.

### 2.5 Transaction Fees

Transaction fees on Taler are computed as the sum of three components:

$$\text{Fee} = \text{BaseFee} + \text{WeightFee} + \text{LengthFee} + \text{Tip}$$

where:

- **WeightFee** = $f_w(\text{weight})$ — computed using a custom ConstantPercent function that maps computational weight to fee at 2% of the reference-time weight value, significantly lower than the Substrate default.
- **LengthFee** = TransactionByteFee × length — where TransactionByteFee = 1 MILLICENT per byte.
- **BaseFee** = ExtrinsicBaseWeight converted to fee — a fixed per-extrinsic charge.
- **Tip** — an optional priority fee set by the transaction sender.

Operational extrinsics (e.g., consensus-critical transactions) pay a 5× multiplier on fees to reflect their higher priority.

A dynamic fee multiplier adjusts fees based on network congestion. The target block fullness is **25%**, and the multiplier moves according to a polynomial adjustment variable ($\frac{75}{1{,}000{,}000}$), with bounds:

$$\text{MinMultiplier} = \frac{1}{1{,}000{,}000{,}000} \quad \text{MaxMultiplier} = \text{MAX\_VALUE}$$

**Fee distribution**: Transaction fees are split 80/20 between the on-chain treasury and the block author, respectively. Tips follow the same 80/20 distribution.

### 2.6 Smart Contracts

Taler supports WebAssembly (Wasm) smart contracts via pallet_contracts, enabling developers to deploy general-purpose, Turing-complete programs that execute on-chain in a sandboxed, deterministic environment. The primary development toolchain is **ink!**, a Rust-based embedded domain-specific language (eDSL) maintained by Parity Technologies. However, any language that compiles to Wasm and adheres to the pallet_contracts ABI may be used, including Solidity via the Solang compiler, AssemblyScript, and others.

#### *2.6.1 Contract Lifecycle*

The lifecycle of a smart contract on Taler consists of three distinct phases:

1. **Code Upload.** The contract's Wasm bytecode is uploaded to the chain via the upload_code extrinsic. The code is stored on-chain, indexed by its CodeHash (a Blake2-256 hash of the bytecode). Uploading code does not instantiate a contract — it merely makes the code available for future instantiation. A **storage deposit** proportional to the code size is reserved from the uploader's account, plus a 30% lockup deposit on the code hash to discourage trivial uploads.

2. **Instantiation.** A contract instance is created by calling the instantiate extrinsic with a reference to a previously uploaded CodeHash, constructor parameters, an initial endowment (TAL transferred to the new contract), and a unique salt value. The salt ensures that multiple instances of the same code produce different contract addresses. The contract address is derived deterministically:

$$\text{Address} = H(\text{deployer} \parallel \text{code\_hash} \parallel \text{input\_data} \parallel \text{salt})$$

where $H$ is Blake2-256. This means contract addresses are predictable before deployment, enabling counterfactual interactions.

3. **Execution.** Once instantiated, the contract can be called via the call extrinsic. Callers specify the destination contract, the function selector and arguments (encoded as raw bytes), an optional TAL value transfer, and a weight (gas) limit. The contract executes within the Wasm virtual machine, reads and writes to its private storage trie, and may call other contracts or transfer funds.

### *2.6.2 Execution Model and Metering*

Contract execution is **metered by weight**, which serves the same purpose as gas in Ethereum: it bounds computation to ensure that block production is never stalled by a long-running contract.

- **Weight limit**: Every contract call specifies a maximum weight. If execution exceeds this limit, all state changes are reverted and the weight is consumed (not refunded).
- **Weight refund**: If execution completes within the limit, the unused portion is refunded to the caller.
- **Weight pricing**: Weight is converted to TAL fees using the same WeightToFee function that prices regular extrinsics (2% of reference-time weight value), making contract execution significantly cheaper than on networks that use the Substrate default.

Storage operations (reads, writes, deletions) are also metered. Each storage access has a defined weight cost based on benchmark measurements against the RocksDB backend.

### *2.6.3 Storage Model*

Each contract instance maintains its own **isolated storage trie**, a key-value store where keys are up to 128 bytes and values are arbitrary byte sequences. The storage trie is rooted in the global chain state, so its integrity is covered by the block's state root hash.

Contracts pay for storage through a **deposit-based model** rather than a per-operation fee:

| Parameter | Value |
|---|---|
| Deposit per storage item | deposit(1, 0) ≈ 15 CENTS |
| Deposit per storage byte | deposit(0, 1) ≈ 6 CENTS |
| Default deposit limit | deposit(1024, 1024 * 1024) |

When a contract writes a new storage entry, the deposit is reserved from the caller's (or contract's) account. When the entry is deleted, the deposit is returned. This incentivizes contracts to clean up unused storage and prevents unbounded state growth.

### 2.6.4 Code Separation and Reuse

A critical architectural feature is the **separation of code and instances**. Multiple contract instances can share the same underlying Wasm bytecode, referenced by CodeHash. This has several benefits:

- **Reduced storage costs**: Deploying 100 instances of the same ERC-20 token contract stores the bytecode only once.
- **Atomic code upgrades**: A contract can be designed to delegate calls to a code hash that can be updated (via a proxy pattern), enabling upgradeable contracts.
- **Shared auditing**: Once a code hash is audited, all instances using it inherit the same security guarantees.

The code hash lockup deposit (30% of the base deposit) is reserved when code is uploaded and returned when the code is removed (only possible if no instances reference it).

### 2.6.5 Cross-Contract Calls

Contracts may call other contracts, creating a **call stack** with a maximum depth of **5 frames**. Each nested call inherits the remaining weight budget from its parent. If a nested call fails (runs out of weight or explicitly reverts), only the nested call's state changes are reverted — the parent call may choose to handle the failure gracefully.

Cross-contract calls support value transfers: the calling contract can attach TAL to the message, which is transferred atomically with the call. If the call reverts, the value transfer is also reverted.

### 2.6.6 Contract Determinism

All contract execution on Taler is **strictly deterministic**. The Wasm virtual machine does not expose floating-point operations (beyond what the Wasm spec requires for determinism), external I/O, or any non-deterministic host functions. This ensures that every node executing the same contract with the same inputs produces identical results, which is essential for consensus.

Contracts that need randomness must query the on-chain randomness source (see Section 2.15), with the caveat that this source is not cryptographically secure against validator manipulation.

### 2.6.7 Runtime Integration and Call Filtering

Smart contracts interact with the broader Taler runtime through **chain extensions** — whitelisted host functions that the runtime exposes to the contract sandbox. At present, the CallFilter is set to Nothing, meaning contracts **cannot dispatch calls to other runtime pallets**. This is a deliberate security measure: allowing contracts to call arbitrary pallets would expose the runtime to reentrancy risks, unexpected weight consumption, and potential breaking changes if pallet interfaces evolve.

The roadmap for relaxing this restriction involves:

1. Auditing specific pallet interfaces for stability and safety (e.g., pallet_balances::transfer, pallet_assets::transfer, pallet_identity::identity_of).
2. Implementing a whitelist of permitted dispatchable calls via a custom CallFilter.
3. Enacting the change through a governance referendum and runtime upgrade.

Chain extensions may also be used to expose custom read-only queries (e.g., “check if account X has a verified identity”) without granting contracts the ability to modify state outside their sandbox.

#### *2.6.8 Development Toolchain*

The primary toolchain for contract development on Taler is:

| Tool | Purpose |
|---|---|
| **ink!** | Rust eDSL for writing contracts with macros that generate the ABI and Wasm bytecode. |
| **cargo-contract** | CLI tool for building, testing, and deploying ink! contracts. |
| **Contracts UI** | Web interface for deploying and interacting with contracts on a running chain. |
| **Solang** | Compiles Solidity to Wasm, enabling Ethereum developers to port contracts. |
| **drink!** | Testing framework for running contracts in a simulated blockchain environment. |

Key contract configuration parameters:

| Parameter | Value |
|---|---|
| Maximum code size | 123 KB |
| Maximum storage key length | 128 bytes |
| Call stack depth | 5 frames |
| Code hash lockup deposit | 30% of deposit |
| Maximum delegate dependencies | 32 |
| Maximum debug buffer length | 2 MB |
| Unsafe/unstable interface | Disabled |

---

### 2.7 Assets and NFTs

Taler provides native runtime support for both fungible and non-fungible tokens without requiring smart contract deployment:

**Fungible Assets** (pallet_assets): Any account can create a new asset class by depositing 100 DOLLARS. Assets support transfers, approvals (delegated spending), metadata, and freezing. This is functionally comparable to ERC-20 tokens but implemented at the runtime level with lower fees and higher throughput.

**Non-Fungible Tokens** (pallet_nfts): Full-featured NFT support including collections, individual items, attributes, metadata, tips, deadlines, and off-chain signatures for gasless approvals. Collection creation requires a deposit of 100 DOLLARS, and each item requires 1 DOLLAR.

**NFT Fractionalization** (pallet_nft_fractionalization): Allows an NFT to be "split" into fungible token shares, enabling fractional ownership. The fractionalized tokens are managed as standard assets, inheriting all the functionality of pallet_assets.

### *2.7.1 Fungible Assets*

The pallet_assets module provides a framework for creating and managing custom fungible tokens on Taler. Each asset is identified by a numeric AssetId (u32) and maintains its own set of accounts, metadata, and administrative roles.

**Asset Creation and Administration.** Any account may create a new asset class by paying a creation deposit. Upon creation, the creator becomes the asset's **owner** and may designate additional administrative roles:

| Role | Capabilities |
|---|---|
| **Owner** | Full control: can set team roles, destroy the asset, and force-transfer. |
| **Admin** | Can mint, burn, freeze/thaw accounts, and manage metadata. |
| **Issuer** | Can mint new tokens to any account. |
| **Freezer** | Can freeze and thaw individual accounts or the entire asset. |

**Core Operations.** The following operations are supported for each asset:

- **Mint**: The issuer creates new tokens and credits them to a target account.
- **Burn**: The admin destroys tokens from a target account.
- **Transfer**: Any token holder can transfer their balance to another account, subject to the asset not being frozen.
- **Approve & Transfer From**: A token holder can approve another account to spend up to a specified amount on their behalf (analogous to ERC-20 approve/transferFrom). Each approval requires an on-chain deposit of 1 DOLLAR to prevent storage spam.
- **Freeze / Thaw**: The freezer can freeze individual accounts (preventing all transfers in/out) or freeze the entire asset (halting all activity globally). Thawing reverses the freeze.
- **Set Metadata**: The admin can attach a name (up to 50 characters), symbol, and decimal precision to the asset for display purposes.
- **Destroy**: The owner can destroy an asset class, returning all deposits. This operation is rate-limited: up to 1,000 accounts are cleaned up per call to avoid excessive block weight.

**Account Model.** Each account holding a non-zero balance of a given asset has an **asset account** stored on-chain. Creating this account requires a deposit of 1 DOLLAR (the AssetAccountDeposit), which is returned when the balance reaches zero and the

account is reaped. This deposit prevents dust attacks where an attacker creates millions of tiny balances to bloat chain state.

| Parameter | Value |
|---|---|
| Asset creation deposit | 100 DOLLARS |
| Asset account deposit | 1 DOLLAR |
| Approval deposit | 1 DOLLAR |
| Metadata deposit (base) | 10 DOLLARS |
| Metadata deposit (per byte) | 1 CENT |
| Maximum name/symbol length | 50 characters |
| Maximum items removed per destroy call | 1,000 |

**Differences from ERC-20.** While pallet_assets provides functionality analogous to ERC-20, several differences are worth noting:

- **Administrative roles**: ERC-20 contracts typically have a single owner or no owner. Taler assets have a granular role system.
- **Freeze capability**: There is no standard freeze mechanism in ERC-20; it must be implemented per-contract. Taler assets support freezing natively.
- **On-chain metadata**: Asset name, symbol, and decimals are stored on-chain and queryable by any pallet or contract. ERC-20 metadata is optional and contract-dependent.
- **No transfer hooks**: Unlike some ERC-20 extensions (e.g., ERC-777), pallet_assets does not support pre/post-transfer hooks. This eliminates reentrancy risks but also limits composability within the asset transfer itself.

### *2.7.2 Non-Fungible Tokens (NFTs)*

The pallet_nfts module provides a comprehensive NFT system supporting collections, individual items, rich metadata, attributes, and advanced features such as off-chain signatures and deadlines.

**Collections and Items.** NFTs are organized into **collections**, each identified by a CollectionId (u32). Within a collection, individual NFTs are identified by an ItemId (u32). A collection is owned by its creator and can contain an arbitrary number of items.

**Collection Management:**

| Operation | Description |
|---|---|
| **Create** | Creates a new collection. Requires a deposit of 100 DOLLARS. |
| **Destroy** | Destroys an empty collection, returning the deposit. |
| **Set Team** | Assigns admin, issuer, and freezer roles (similar to pallet_assets). |
| **Lock Collection** | Permanently locks certain collection settings (max supply, metadata mutability, attribute mutability). |

| Operation | Description |
|---|---|
| **Set Accept Ownership** | Allows an account to signal willingness to accept collection ownership transfer. |

**Item Operations:**

| Operation | Description |
|---|---|
| **Mint** | Creates a new item within a collection. Requires a deposit of 1 DOLLAR. |
| **Burn** | Destroys an item, returning the deposit. |
| **Transfer** | Transfers ownership of an item to another account. |
| **Lock Transfer** | The collection admin can lock an item, preventing transfers until unlocked. |
| **Approve Transfer** | The item owner can approve a specific account to transfer the item on their behalf, optionally with a deadline (maximum 360 days). |

**Attributes and Metadata.** Each item and collection can carry two types of ancillary data:

- **Metadata**: A URI or binary blob (up to 256 bytes) typically pointing to off-chain content (images, JSON descriptors). Metadata has a base deposit of 10 DOLLARS plus 1 CENT per byte.
- **Attributes**: Key-value pairs (keys up to 64 bytes, values up to 256 bytes) stored directly on-chain. Attributes can be set by the collection owner, the item owner (with approval limits), or the system. Up to 20 item-level attribute approvals and 10 attributes per batch call are supported.

Attributes come in three namespaces:

1. **Collection attributes**: Apply to the collection as a whole (e.g., “royalty_percent”, “license”).
2. **Item attributes**: Apply to individual items (e.g., “rarity”, “color”).
3. **System attributes**: Set by the runtime for internal bookkeeping.

**Advanced Features.** Taler’s NFT module includes several features beyond basic minting and transfer:

- **Tips**: Any account can tip an NFT item, transferring TAL directly to the item owner. Up to 10 tips can be batched in a single extrinsic.
- **Off-chain Signatures**: Item owners can sign transfer approvals off-chain using their sr25519 key. The signed approval can be submitted by anyone, enabling gasless (meta-transaction) NFT transfers where a third party pays the fees.
- **Swap Offers**: Item owners can create atomic swap offers — “I’ll trade my Item A for your Item B” — with an optional price differential and deadline. The counterparty can accept the swap in a single extrinsic, and both items transfer atomically.

- **Buy / Set Price**: Items can be listed for sale at a fixed price. Any account can purchase a listed item by paying the asking price, which transfers atomically alongside the item.

**All Enabled Features.** Taler enables the full feature set of pallet_nfts via the PalletFeatures::all_enabled() configuration, which includes trading, attributes, approvals, and atomic swaps.

| Parameter | Value |
|---|---|
| Collection deposit | 100 DOLLARS |
| Item deposit | 1 DOLLAR |
| Metadata deposit (base) | 10 DOLLARS |
| Attribute deposit (base) | 10 DOLLARS |
| Deposit per byte | 1 CENT |
| Maximum attribute key length | 64 bytes |
| Maximum attribute value length | 256 bytes |
| Maximum approvals per item | 20 |
| Maximum item attribute approvals | 20 |
| Maximum tips per call | 10 |
| Maximum deadline duration | 360 days |
| Maximum attributes per call | 10 |
| String limit | 256 characters |

### *2.7.3 NFT Fractionalization*

The pallet_nft_fractionalization module bridges the NFT and fungible asset systems, enabling **fractional ownership** of non-fungible tokens. This is particularly useful for high-value NFTs where multiple parties wish to share ownership, or for creating liquid markets around otherwise illiquid unique assets.

**Fractionalization Process:**

1. An NFT owner calls fractionalize, specifying the collection ID, item ID, and the number of fungible shares to create.
2. The NFT is locked in the fractionalization pallet (it cannot be transferred or burned while fractionalized).
3. A new fungible asset is created via pallet_assets with the specified share count, using the symbol “FRAC” and name “Frac” by default. All shares are minted to the fractionalizer’s account.
4. The shares can then be transferred, traded, or used like any other fungible asset.

**Unification Process:**

1. Any account that holds **all** shares of a fractionalized NFT can call unify, burning all shares and unlocking the original NFT.

2. The NFT is transferred to the unifier's account, and the fungible asset is destroyed.

This creates a clean round-trip: an NFT can be split into shares, the shares can circulate and change hands, and whoever reassembles all shares can reclaim the original NFT.

| Parameter | Value |
|---|---|
| Fractionalization deposit | 100 DOLLARS (same as asset creation) |
| Default share symbol | "FRAC" |
| Default share name | "Frac" |
| Pallet ID | fraction |

**Use Cases:**

- **Collective art ownership**: A community can co-own a valuable digital artwork, with each member holding a proportional share.
- **NFT-backed lending**: Fractional shares can serve as collateral in DeFi protocols built on Taler's contract layer.
- **Liquidity provision**: By converting an illiquid NFT into fungible shares, holders can access liquidity without selling the entire asset.

### 2.8 Proxy Accounts

The Taler runtime includes a proxy mechanism (pallet_proxy) that allows one account to grant another account permission to act on its behalf — analogous to a power of attorney in traditional legal systems. Proxies are useful for operational security setups where a "cold" account holding significant funds delegates day-to-day operations to a "hot" account with limited permissions.

Each proxy relationship is typed, restricting the delegate to a specific category of actions:

| Proxy Type | Permitted Actions |
|---|---|
| Any | All extrinsics without restriction. |
| NonTransfer | All extrinsics **except** balance transfers, asset transfers, NFT transfers, vested transfers, and index transfers. |
| Governance | Only governance-related extrinsics: democracy, council, technical committee, elections, and treasury calls. |
| Staking | Only staking-related extrinsics: bonding, nominating, setting controller, Vivid Staking, etc. |

Proxy types form a hierarchy: Any is a superset of NonTransfer, which is a superset of Governance and Staking. This means an Any proxy can perform everything a NonTransfer proxy can, and so on.

**Announcements and time-delays**: Proxies may be configured with an optional time delay. When a delay is set, the proxy must first *announce* its intended call, then wait for the specified number of blocks before executing it. During the delay window, the proxied account (or any other authorized party) may cancel the pending call. This provides a safety mechanism against compromised proxy keys.

Key proxy parameters:

| Parameter | Value |
|---|---|
| Maximum proxies per account | 32 |
| Maximum pending announcements | 32 |
| Proxy deposit (base) | deposit(1, 8) |
| Proxy deposit (per additional proxy) | deposit(0, 33) |
| Announcement deposit (base) | deposit(1, 8) |
| Announcement deposit (per announcement) | deposit(0, 66) |

Proxy deposits are reserved from the delegating account's balance and returned when the proxy relationship is removed.

### 2.9 Multisig Accounts

The multisig module (pallet_multisig) enables the creation of **M-of-N multi-signature accounts** — accounts that require approval from a threshold number of signatories before any extrinsic can be dispatched. Multisig accounts are derived deterministically from the sorted set of signatory accounts and the threshold value, meaning no explicit "creation" transaction is needed; the address is computable off-chain.

A typical multisig workflow proceeds as follows:

1. **Initiation**: One signatory submits the call (or its hash) along with their approval. The call data and approval state are stored on-chain, and a deposit is reserved.
2. **Approval**: Additional signatories submit their approvals referencing the same call hash. Each approval is recorded on-chain.
3. **Execution**: When the $M$th approval is received (meeting the threshold), the call is automatically dispatched from the multisig account's origin.
4. **Cancellation**: The initiating signatory may cancel a pending multisig operation, reclaiming the deposit.

| Parameter | Value |
|---|---|
| Maximum signatories | 100 |
| Deposit (base) | deposit(1, 88) |
| Deposit (per additional signatory) | deposit(0, 32) |

Deposits scale linearly with the number of signatories to account for on-chain storage costs. The base deposit covers the storage of the call data and approval state, while the per-signatory deposit covers each additional AccountId (32 bytes) stored.

Multisig accounts are particularly useful for treasury management, team-controlled validator operations, and any scenario requiring shared custody or organizational approval workflows.

### 2.10 Identity

The identity module (pallet_identity) provides an on-chain framework for accounts to register human-readable identity information and have it verified by trusted third parties called **registrars**.

An identity record may include fields such as display name, legal name, email, website, Twitter handle, and up to 100 additional custom fields. All identity data is stored on-chain and requires a deposit proportional to the data size, discouraging spam.

**Registration and verification workflow**:

1. **Set identity**: An account submits its identity information along with a deposit.
2. **Request judgement**: The account requests verification from one or more registrars, paying the registrar's fee.
3. **Provide judgement**: A registrar reviews the evidence (off-chain) and submits a judgement on-chain. Possible judgements include Reasonable, KnownGood, OutOfDate, LowQuality, Erroneous, and others.
4. **Clear identity**: An account may remove its identity at any time, reclaiming the deposit.

**Sub-accounts**: An identity holder may register up to 100 sub-accounts, each with its own name, linked to the parent identity. This is useful for organizations that operate multiple validator nodes or service accounts under a single verified identity.

**Registrars**: Registrars are appointed by root or by a majority council vote (EnsureRootOrHalfCouncil). Each registrar sets its own fee and acceptable fields. Up to 20 registrars may operate concurrently.

| Parameter | Value |
|---|---|
| Basic identity deposit | 10 DOLLARS |
| Per-field deposit | 250 CENTS |
| Sub-account deposit | 2 DOLLARS |
| Maximum sub-accounts | 100 |
| Maximum additional fields | 100 |
| Maximum registrars | 20 |
| Slashed deposits destination | Treasury |

Identity verification is essential for building social consensus in governance, establishing validator reputation, and enabling future use cases such as under-collateralized lending or identity-gated access control in smart contracts.

### 2.11 Account Recovery

The recovery module (pallet_recovery) implements a **social recovery** mechanism, allowing an account owner to designate a set of trusted "friends" who can collectively vouch for the recovery of access to the account. This addresses the critical problem of lost or compromised private keys without relying on centralized custodians.

**Setup**:

1. The account owner creates a recovery configuration specifying:
   - A set of **friends** (trusted accounts),
   - A **threshold** (minimum number of friends required to authorize recovery),
   - A **delay period** (time the recovery process must remain open before it can be completed).
2. A deposit is reserved to cover on-chain storage.

**Recovery process**:

1. A recovering account (the new key) initiates a recovery attempt against the lost account.
2. The designated friends each submit a **vouch** transaction confirming the recovery request.
3. Once the threshold is met and the delay period has elapsed, the recovering account gains proxy-like access to the lost account, enabling it to transfer funds and reconfigure the account.

| Parameter | Value |
|---|---|
| Configuration deposit (base) | 5 DOLLARS |
| Friend deposit (per friend) | 50 CENTS |
| Maximum friends | 9 |
| Recovery deposit | 5 DOLLARS |

The relatively small maximum friend count (9) and the required deposit ensure that recovery configurations remain manageable and resistant to abuse, while still providing robust protection against key loss.

### 2.12 Vesting

The vesting module (pallet_vesting) enforces time-locked token release schedules. Vested tokens are included in an account's total balance and may be used for staking and governance voting, but they **cannot be transferred** until they have vested according to the defined schedule.

Each vesting schedule is defined by three parameters:

- **Locked**: The total number of tokens subject to the schedule.
- **Per-block unlock**: The number of tokens that become transferable with each new block.

- **Starting block**: The block number at which the unlock begins.

The amount of unlocked (transferable) tokens at block $b$ for a schedule starting at block $s$ with per-block rate $r$ and total lock $L$ is:

$$\text{Unlocked}(b) = \min(r \times (b - s),\ L) \quad \text{for } b \geq s$$

An account may have up to **28 concurrent vesting schedules**, accommodating complex distribution scenarios (e.g., multiple funding rounds, grants, or premine tranches). The minimum vested transfer amount is **100 DOLLARS**.

Vested funds may be used for all purposes except transferring and reserving, ensuring that premine holders and grant recipients can participate fully in staking and governance from day one.

### 2.13 Scheduler

The scheduler module (pallet_scheduler) enables the **time-based execution** of dispatchable calls. A call can be scheduled to execute at a specific future block number, optionally with periodic repetition.

Only **root origin** (i.e., governance or sudo during the bootstrap phase) may schedule calls, preventing abuse by arbitrary users. The scheduler is a critical component of the governance pipeline: when a referendum passes, the approved call is scheduled for execution after the enactment delay period.

| Parameter | Value |
|---|---|
| Maximum scheduled calls per block | 50 |
| Maximum weight per scheduled block | 80% of max block weight |
| Schedule origin | Root only |

The scheduler integrates with the **preimage** module: large call data is stored as a preimage (referenced by hash), and the scheduler retrieves and dispatches it at the appointed block. This separation ensures that scheduling a call does not bloat block storage with redundant call data.

### 2.14 Utility Functions

The utility module (pallet_utility) provides several convenience extrinsics that improve the developer and user experience:

- **batch**: Dispatches multiple calls in a single extrinsic. If any call fails, subsequent calls still execute, and the batch completes with a partial success event. Useful for combining multiple operations (e.g., claim rewards + transfer + nominate) into a single transaction.
- **batch_all**: Similar to batch, but **atomic** — if any call in the batch fails, the entire batch is reverted. Suitable for operations that must succeed together or not at all.

- **dispatch_as**: Dispatches a call with a modified origin. This is used internally by governance and proxy mechanisms to execute calls on behalf of other accounts or with elevated privileges.
- **force_batch**: Dispatches multiple calls, ignoring any individual failures. Always completes, reporting which calls succeeded and which failed.

These primitives reduce the number of transactions users must submit, lower aggregate fees, and enable atomic multi-step workflows that would otherwise require complex off-chain coordination.

### 2.15 On-Chain Randomness

The pallet_insecure_randomness_collective_flip module provides a source of pseudo-random values derived from the hashes of recent blocks. Other pallets and smart contracts may query this source when randomness is needed (e.g., for fair selection algorithms or game mechanics).

> **Security notice**: This randomness source is **not cryptographically secure** against adversarial manipulation. Validators who author blocks can influence the block hash and therefore bias the random output. For applications requiring strong randomness guarantees (e.g., lotteries, cryptographic protocols), an external verifiable random function (VRF) or a dedicated randomness beacon should be used. Future upgrades may integrate a BABE-epoch-based VRF randomness source to address this limitation.

### 2.16 Transaction Storage

The transaction storage module (pallet_transaction_storage) enables users to store arbitrary data on-chain, indexed by the block in which it was included. This provides a simple, censorship-resistant data availability layer.

| Parameter | Value |
|---|---|
| Max transactions per block | Default (configurable) |
| Max transaction size | Default (configurable) |
| Fee destination | None (burned) |

Transaction storage is useful for timestamping documents, anchoring off-chain data hashes, and providing permanent data availability for applications that require on-chain proof of existence without the complexity of smart contract deployment.

---

## 3. Staking and Reward Model

### 3.1 Nominated Proof of Stake (NPoS)

Taler uses a Nominated Proof-of-Stake (NPoS) system for Sybil resistance and validator selection. The system consists of two roles:

- **Validators**: Nodes that produce blocks and participate in consensus. Validators must bond (lock) a stake as collateral.
- **Nominators**: Token holders who back validators with their stake. Nominators share in both the rewards and the slashing risks of their chosen validators.

Validator election is performed using the **Sequential Phragmén** algorithm, which optimally distributes nominator stake across validators to maximize the minimum backing stake of any elected validator. This algorithm runs in a multi-phase election process:

1. **Signed phase**: Off-chain workers or external parties submit election solutions on-chain with a deposit.
2. **Unsigned phase**: Validators generate and submit solutions directly.
3. **On-chain fallback**: If no valid solution is received, an on-chain computation produces the result.

The maximum number of active validators is **1,000**, and each nominator can nominate up to **16** validators. Up to **256** nominators per validator receive rewards in each era.

Unbonding requires a **28-day** waiting period before staked tokens become transferable again.

### 3.2 Deterministic Reward Calculation

Taler's most significant departure from the standard Substrate staking model is its **deterministic, deposit-like reward calculation**. Rather than deriving rewards from an inflation curve that targets a particular staking ratio, Taler computes rewards based on a fixed annual interest rate applied to each participant's bonded stake.

The **standard annual interest rate** is:

$$r_{\text{standard}} = 12\%$$

For a nominator $n$ who has nominated validators $v_1, v_2, \ldots, v_N$, with stake $\text{Value}_n^v$ allocated to each validator $v$ by the Phragmén algorithm, the era reward is:

$$P_n = \sum_{v=1}^{N} \frac{\text{EraDuration}_{\text{ms}}}{31{,}557{,}600{,}000} \cdot r_n \cdot \text{Value}_n^v \cdot (1 - c_v)$$

where: - $r_n$ is the effective annual interest rate for nominator $n$ (including any Vivid Staking bonus), - $c_v$ is the commission rate of validator $v$, - $\text{EraDuration}_{\text{ms}}$ is the actual duration of the era in milliseconds, - $31{,}557{,}600{,}000$ is the number of milliseconds in a Julian year.

For a validator $v$ with its own bonded stake $\text{Value}_v$ and $K$ nominators:

$$P_v = \frac{\text{EraDuration}_{\text{ms}}}{31{,}557{,}600{,}000} \cdot r_v \cdot \text{Value}_v + \sum_{n=1}^{K} \frac{P_n \cdot c_v}{1 - c_v}$$

The first term represents the validator's own staking reward (calculated identically to a nominator's, without commission deduction on the validator's own stake). The second term represents the commission income collected from all nominators backing this validator.

- **Worked Example**

  Suppose nominator X stakes 1,000 TAL and nominates validators A and B. The Phragmén algorithm allocates 600 TAL to A and 400 TAL to B. Validator A has a 1% commission; validator B has a 15% commission. The standard interest rate is 12%. Era duration is 86,400,000 ms (1 day).

  **Nominator X's reward:**

$$P_X = \left(\frac{86{,}400{,}000}{31{,}557{,}600{,}000} \times 0.12 \times 600 \times 0.99\right) + \left(\frac{86{,}400{,}000}{31{,}557{,}600{,}000} \times 0.12 \times 400 \times 0.85\right)$$

$$P_X = 0.195 + 0.113 = 0.308 \text{ TAL}$$

  **Validator A's commission income** (assuming zero self-stake for simplicity):

$$P_A = 0 + \frac{0.195 \times 0.01}{1 - 0.01} = 0.002 \text{ TAL}$$

  **Validator B's commission income:**

$$P_B = 0 + \frac{0.113 \times 0.15}{1 - 0.15} = 0.020 \text{ TAL}$$

### 3.3 Vivid Staking

Vivid Staking is an optional mechanism that allows participants to lock their staked funds for a predefined period (measured in months) in exchange for a higher interest rate. The additional yield per month of lock-up is:

$$r_{\text{vivid}} = 0.5\% \text{ per month}$$

The maximum lock duration is 12 months, yielding a maximum bonus of $12 \times 0.5\% = 6\%$ additional annual interest. Combined with the standard rate:

$$r_{\max} = 12\% + 6\% = 18\% \text{ per annum}$$

| Lock Duration | Additional APR | Total APR |
|---|---|---|
| 0 months (standard) | 0% | 12.0% |
| 1 month | 0.5% | 12.5% |
| 3 months | 1.5% | 13.5% |
| 6 months | 3.0% | 15.0% |
| 12 months | 6.0% | 18.0% |

Vivid Staking is activated by calling staking::vivid(months_count). The lock takes effect at the beginning of the following era. During the lock period:

- **Prohibited**: Bonding additional funds, unbonding, and withdrawing.
- **Permitted**: Changing nomination targets, changing the reward payout destination account.

Vivid Staking can be cancelled before it takes effect (i.e., before the current era ends) by calling staking::unvivid().

### 3.4 Automatic Reward Payout

Taler introduces **automatic reward payout** using the on_idle hook mechanism. At the end of each block, if there is remaining unused block weight, the runtime automatically processes pending reward payouts for the previous era.

This ensures that in normal network conditions, stakers receive their rewards without any manual intervention. If the network is under sustained heavy load and automatic payouts cannot complete within a single era, the staking::payout_stakers extrinsic remains available for manual invocation.

A Runtime API method StakingApi::estimated_staking_payout(account) is also provided, returning the estimated reward accrued for a given account from the start of the current era to the present moment.

### 3.5 Issuance Cap

The total supply of TAL is hard-capped at:

$$\text{IssuanceLimit} = 108{,}888{,}888{,}888 \text{ TAL}$$

TAL tokens are divisible to **12 decimal places** (1 TAL = $10^{12}$ smallest units, where the smallest unit is defined as 1 TMILLICENT = $10{,}000{,}000$ base units, 1 TCENT = $1{,}000 \times$ TMILLICENTS, and 1 TALER = $100 \times$ TCENTS).

As the total issuance approaches this cap, staking rewards will naturally decrease, eventually reaching zero when the cap is met. This creates a long-term deflationary pressure and incentivizes early participation.

### 3.6 Slashing

Validators and their nominators may be subject to slashing penalties for misbehavior. Slashing is designed to be proportional to the severity and breadth of the offense:

#### *3.6.1 Unavailability*

Validators must send periodic “I’m online” heartbeat messages each session. Failure to produce blocks or send heartbeats results in an **unresponsiveness** offense. The slash amount is:

$$F_{\text{unavail}} = \min\left(\frac{3\left(x - \left\lfloor\frac{n}{10}\right\rfloor - 1\right)}{n},\ 1\right) \times 7\%$$

where $x$ is the number of unresponsive validators and $n$ is the total validator set size. Notably, if fewer than $\sim 10\%$ of validators are simultaneously unresponsive, no slash is applied.

| Unresponsive Validators (out of 100) | Slash |
|---|---|
| ≤ 10 | 0% |
| 14 | ~0.6% |
| 33 | ~5% |

### *3.6.2 Equivocation*

**BABE equivocation** occurs when a validator produces multiple blocks in the same slot. **GRANDPA equivocation** occurs when a validator casts conflicting finality votes in the same round. The slash for equivocation is:

$$F_{\text{equiv}} = \min\left(\left(\frac{3x}{n}\right)^2,\ 1\right)$$

where $x$ is the number of offenders discovered in the same time window and $n$ is the total validator count.

| Equivocating Validators (out of 100) | Slash |
|---|---|
| 1 | 0.09% |
| 5 | 2.25% |
| 20 | 36% |

All slashing penalties are deferred for **27 eras** (~27 days), during which the Council may vote to cancel a slash if it was caused by non-malicious circumstances. Uncancelled slashes are deposited into the treasury.

An offending validator is immediately removed from the active set regardless of whether a monetary slash is applied.

---

## 4. Governance

Taler implements a multi-layered on-chain governance system. All governance actions ultimately execute as privileged runtime calls, enabling stakeholders to modify any aspect of the chain's behavior — including runtime upgrades, parameter changes, and treasury spending — without hard forks.

### 4.1 Sudo (Bootstrap Phase)

At network launch, the pallet_sudo module designates a single superuser account capable of executing any dispatchable call with root privileges. This is a temporary measure to facilitate initial network configuration, parameter tuning, and emergency intervention during the bootstrap phase.

Once the network reaches a stable state with active governance participation, pallet_sudo is **permanently removed** via a runtime upgrade enacted through the governance process itself. After removal, all privileged operations must be authorized through democracy, the council, or the technical committee.

### 4.2 Democracy

The democracy module enables any TAL holder to participate in network governance through **referenda** — binding on-chain votes that, if approved, execute a specified dispatchable call.

**Proposing a Referendum**: Any stakeholder may propose a referendum by depositing a minimum of **100 DOLLARS** (in TAL). Other stakeholders may second the proposal by depositing an equal amount. The proposal with the highest total deposit is selected for the next voting period.

**Voting**: Referenda use coin-weighted voting with **conviction multipliers** (lock-time weighting). Voters choose to lock their tokens for a period after the vote, with longer lock periods amplifying their voting power:

| Lock Period | Conviction Multiplier |
|---|---|
| None | 0.1× |
| 1× Enactment Period | 1× |
| 2× Enactment Period | 2× |
| 4× Enactment Period | 3× |
| 8× Enactment Period | 4× |
| 16× Enactment Period | 5× |
| 32× Enactment Period | 6× |

Voters lock their entire account balance for the chosen period — partial-balance voting is not supported.

**Delegation**: Token holders may delegate their voting power to a trusted account. Delegation is recursive up to a depth limit, and a delegator may override their delegate's vote on any specific referendum at any time before it concludes. Cyclic delegation is prevented.

Key democracy parameters:

| Parameter | Value |
|---|---|
| Launch period (time between referenda) | 28 days |

| Parameter | Value |
|---|---|
| Voting period | 28 days |
| Enactment delay | 30 days |
| Fast-track voting period | 3 days |
| Cooloff period | 28 days |
| Minimum deposit | 100 DOLLARS |
| Maximum concurrent proposals | 100 |

**Tallying**: The default tally type is *supermajority-to-pass* with adaptive quorum biasing. If the council unanimously endorses a proposal, it moves to a *supermajority-to-fail* threshold (making passage easier). Simple majority applies when the council supports the proposal by a 3/4 supermajority.

### 4.3 Council

The Council is an elected body of accounts that hold special governance privileges. Council members are elected via the **Phragmén approval voting** algorithm, with elections occurring every **7 days**. The council vote is independent of the staking process — staking nominations do not influence council elections.

| Parameter | Value |
|---|---|
| Desired council members | 13 |
| Desired runners-up | 7 |
| Term duration | 7 days |
| Candidacy bond | 10 DOLLARS |
| Motion duration | 5 days |
| Maximum council members | 100 |

Council powers include:

- **Proposing referenda** with modified quorum biasing (supermajority-to-fail for unanimous proposals, simple majority for 3/4+ support).
- **Cancelling dangerous referenda** with a 2/3 council vote.
- **Approving treasury proposals** with a 3/5 council vote.
- **Rejecting treasury proposals** with a simple majority.
- **Cancelling slashes** with a 3/4 supermajority.
- **Vetoing** a single proposal per member (one-time use per proposal, subject to cooloff period).
- **Electing the Technical Committee**.

### 4.4 Technical Committee

The Technical Committee (TC) is a group of technically proficient individuals appointed by the Council. Its primary responsibilities include:

- **Fast-tracking referendum proposals**: With a 2/3 TC vote, a proposal can be moved to a rapid voting period (3 days instead of 28).
- **Instant enactment**: With a unanimous TC vote, a proposal can be enacted immediately.
- **Cancelling pre-vote proposals**: A unanimous TC vote (or root) can cancel a proposal before it reaches a referendum.
- **Vetoing council proposals**: Any single TC member may veto a council proposal once.

| Parameter | Value |
|---|---|
| Maximum TC members | 100 |
| Motion duration | 5 days |

The TC membership is managed by root or by a majority council vote, ensuring accountability.

### 4.5 Treasury

The Treasury is an on-chain fund designed to finance network development, maintenance, and community initiatives. It is automatically funded through two channels:

1. **80% of all transaction fees** (the remaining 20% goes to block authors).
2. **100% of slashing penalties** that are not cancelled by the Council.

**Spending mechanism**: Treasury funds are disbursed through three mechanisms:

- **Proposals**: Any stakeholder may submit a spending proposal with a bond of 5% of the requested amount (minimum 1 DOLLAR). The Council votes to approve or reject. Approved proposals are paid out at the end of each **spend period** (12 months).
- **Bounties**: The Council can create bounties for specific tasks and assign curators who manage the work and release funds. Curator deposits range from 1 to 100 DOLLARS.
- **Tips**: Elected council members can tip contributors. The final tip amount is the median of all council-submitted tip values.

| Parameter | Value |
|---|---|
| Proposal bond | 5% of requested amount |
| Minimum proposal bond | 1 DOLLAR |
| Spend period | 12 months |
| Burn rate (unspent funds) | 10% per spend period |
| Tip countdown | 1 day |
| Tip finders fee | 20% |
| Maximum tip amount | 500 DOLLARS |
| Payout period | 30 days |

The 10% burn rate on unspent treasury funds at the end of each spend period creates a deflationary mechanism that incentivizes the community to actively allocate treasury resources.

---

## 5. Network Launch and Genesis

### 5.1 Genesis Configuration

The genesis block defines the initial state of the Taler blockchain. The genesis configuration (chain specification) is generated once and remains immutable thereafter. Key genesis parameters include:

| Configuration Area | Content |
|---|---|
| balances | Initial account balances, including the premine allocation. |
| session | Session keys for the initial validator set. |
| staking | Initial validators and their nominators with bonded stakes. |
| sudo | The initial superuser account key. |
| vesting | Premine lock-up schedules. |
| elections | Initial council members. |

### 5.2 Premine and Vesting

The total premine allocation is:

$$\text{Premine} = 888{,}888{,}888 \text{ TAL}$$

The premine is distributed among $N$ initial validators, each receiving $\frac{\text{Premine}}{N}$ tokens. These tokens are subject to a **10-year vesting schedule** enforced by pallet_vesting, during which they are gradually unlocked. A small additional allocation per account covers initial transaction fees.

The minimum vested transfer amount is **100 DOLLARS**, and up to 28 concurrent vesting schedules per account are supported.

### 5.3 Runtime Upgrades

Taler supports forkless runtime upgrades via the system::set_code extrinsic. The upgrade process is as follows:

1. During the sudo phase: The superuser directly invokes system::set_code with the new compiled Wasm runtime binary (taler_runtime.compact.compressed.wasm).
2. After sudo removal: A runtime upgrade is proposed as a governance referendum. If approved, the scheduler automatically enacts the code change after the enactment delay.

Upon upgrade, all nodes automatically download and begin executing the new runtime Wasm from the chain state, without requiring manual software updates or coordinated restarts.

> **Important**: The production and fast-runtime (debug) builds have different time constants. Upgrading between these build types on a live chain will halt block production and is strictly prohibited.

---

## 6. Interoperability and Future Work

Future development directions include:

- **Parachain integration**: Taler may connect to the Polkadot relay chain as a parachain or parathread, enabling trustless cross-chain communication and asset transfers via XCMP (Cross-Chain Message Passing).
- **Bridge contracts**: Native bridge pallets or smart-contract-based bridges to Ethereum and other EVM-compatible chains.
- **Contract call whitelisting**: Gradually expanding the set of runtime calls accessible from smart contracts, enabling richer dApp interactions with identity, governance, assets, and staking.
- **Governance evolution**: Migration from Governance V1 to OpenGov (Governance V2), introducing tracks, origins, and more granular delegation.
- **State rent and contract improvements**: Implementing storage deposits and automatic contract eviction to manage chain state growth.

---

## 7. Conclusion

Taler presents a blockchain architecture that combines the robustness of modular framework with innovative modifications to the staking and reward model. The Vivid Staking extension rewards long-term commitment without mandating it, creating a voluntary alignment of incentives.

The multi-layered governance system — spanning direct democracy, an elected council, a technical committee, and a self-funded treasury — provides the tools for the network to evolve autonomously. Forkless runtime upgrades ensure that governance decisions translate directly into protocol changes without coordination overhead or community fragmentation.

With native support for Wasm smart contracts, fungible and non-fungible tokens, on-chain identity, social recovery, proxy and multisig accounts, and a hard issuance cap of 108,888,888,888 TAL, Taler is positioned as a versatile platform for decentralized applications, digital asset management, and community-driven governance. As the ecosystem matures, the removal of the sudo module and the expansion of interoperability features will complete Taler's transition to a fully decentralized, self-sustaining network.

## Appendix A: Glossary

| Term | Definition |
|---|---|
| **Account** | A unique on-chain entity identified by a cryptographic public key (AccountId, 32 bytes). Accounts hold balances, submit extrinsics, participate in staking and governance, and interact with smart contracts. |
| **Authority** | A node that is authorized to participate in the consensus process — specifically, to author blocks (BABE authority) or cast finality votes (GRANDPA authority). In Taler, the set of authorities is derived from the elected validator set. |
| **BABE** | Blind Assignment for Blockchain Extension. A slot-based block production mechanism that uses a verifiable random function (VRF) to assign block authoring slots to validators. BABE ensures liveness by guaranteeing that every slot produces at least one block via secondary slot assignment. |
| **Block** | The fundamental unit of the blockchain. A block contains a header (parent hash, state root, extrinsics root, digest, block number) and a body (an ordered list of extrinsics). Each block represents a discrete state transition. |
| **Bonding** | The act of locking tokens as collateral to participate in staking, either as a validator or a nominator. Bonded tokens cannot be transferred until they are unbonded and the unbonding period has elapsed. |
| **Bounty** | A treasury-funded task delegated to a curator. Bounties are proposed by the Council to incentivize specific work (e.g., bug fixes, feature development, audits) and are paid out upon successful completion as verified by the curator. |
| **Chain Specification** | A JSON configuration file that defines the genesis state of the blockchain, including initial balances, validator keys, governance parameters, and other module configurations. Generated once and immutable thereafter. |
| **Commission** | A percentage of nominator rewards retained by a validator as compensation for operating infrastructure. Commission is deducted before rewards are distributed to nominators. |
| **Consensus** | The process by which nodes in a distributed network agree on the canonical chain of blocks. Taler uses a hybrid consensus model: BABE for block production and GRANDPA for finality. |
| **Conviction** | A multiplier applied to governance votes based on the duration for which the voter agrees to lock their tokens after a referendum. Longer lock periods yield higher conviction (up to 6×), amplifying voting power. |

| Term | Definition |
|---|---|
| **Council** | An elected governance body of up to 13 members (expandable to 100) with special privileges, including proposing referenda with modified quorum biasing, cancelling dangerous referenda, approving treasury spending, and appointing the Technical Committee. |
| **Delegation** | The act of assigning one's voting power to another account in governance referenda. Delegation is voluntary, revocable at any time, and recursive up to a depth of five. |
| **Democracy** | The governance module that enables token holders to propose and vote on binding referenda. Referenda execute arbitrary dispatchable calls upon approval, enabling any on-chain state change including runtime upgrades. |
| **Deposit** | A quantity of tokens reserved (locked) from an account's balance as collateral for on-chain actions such as creating a proxy, registering an identity, or submitting a governance proposal. Deposits are returned when the associated action is reversed or completed. |
| **Epoch** | A fixed-length period (2,400 blocks, ~4 hours) within which the BABE authority set and VRF randomness remain constant. At epoch boundaries, new session keys may take effect and the randomness seed is rotated. |
| **Equivocation** | A slashable offense in which a validator produces conflicting outputs for the same consensus round — either two blocks in the same BABE slot or two conflicting finality votes in the same GRANDPA round. |
| **Era** | A period of 6 sessions (~24 hours) at the end of which staking rewards are calculated, the validator set is re-elected, and slashing penalties are assessed. |
| **Existential Deposit** | The minimum balance an account must maintain to remain active on-chain. Accounts whose free balance falls below this threshold are automatically removed ("reaped"), and their remaining dust balance is destroyed. On Taler, this is set to $10 \times$ TMILLICENTS. |
| **Extrinsic** | Any data originating outside the runtime that is included in a block. Extrinsics encompass signed transactions (e.g., balance transfers), unsigned transactions (e.g., equivocation reports), and inherents (e.g., timestamps). Every state change on Taler is encoded as an extrinsic. |
| **Finality** | The guarantee that a block and all its ancestors will never be reverted. GRANDPA provides deterministic (as opposed to probabilistic) finality, meaning finalized blocks are irreversible under the assumption that fewer than one-third of validators are Byzantine. |

| Term | Definition |
|---|---|
| **FRAME** | Framework for Runtime Aggregation of Modularized Entities. The Substrate development framework used to compose blockchain runtimes from individual pallets. |
| **Genesis Block** | Block number 0 — the first block in the chain, whose state is defined by the chain specification. The genesis block has no parent and establishes the initial configuration of all pallets. |
| **GRANDPA** | GHOST-based Recursive ANcestor Deriving Prefix Agreement. A Byzantine fault-tolerant finality gadget that operates alongside BABE. GRANDPA can finalize multiple blocks simultaneously and tolerates up to $\frac{1}{3}$ Byzantine authorities. |
| **Heartbeat** | A periodic "I'm online" message sent by validators during each session via the pallet_im_online module. Failure to send heartbeats or produce blocks results in an unresponsiveness report and potential slashing. |
| **Inherent** | A special type of extrinsic that is not signed by any external account but is instead inserted by the block author. Inherents provide data that the runtime needs but cannot determine on its own, such as the current timestamp. |
| **ink!** | A Rust-based embedded domain-specific language (eDSL) for writing WebAssembly smart contracts targeting Substrate's pallet_contracts. ink! provides a safe, ergonomic interface for contract development with familiar Rust tooling. |
| **Issuance Cap** | The hard upper limit on the total supply of TAL tokens: 108,888,888,888 TAL. No new tokens can be minted once this limit is reached, regardless of staking reward calculations. |
| **Multisig** | A multi-signature account that requires approval from a configurable threshold ($M$) of designated signatories ($N$) before any extrinsic can be dispatched from the account. Multisig addresses are derived deterministically from the signatory set and threshold. |
| **Nominator** | A token holder who participates in staking by delegating their bonded tokens to one or more validators. Nominators share in the rewards earned by their chosen validators and are also subject to slashing if those validators misbehave. |
| **NPoS** | Nominated Proof of Stake. The staking mechanism used by Taler, in which nominators back validators with their stake and the Phragmén election algorithm selects the active validator set to maximize decentralization. |
| **Pallet** | A modular component of a Substrate runtime, implemented in Rust and compiled to Wasm. Each pallet encapsulates a specific domain of blockchain logic (e.g., balances, staking, governance) and exposes storage items, dispatchable calls, events, and errors. |

| Term | Definition |
|---|---|
| **Phragmén** | A sequential election algorithm (originally devised by Lars Edvard Phragmén) adapted by the Web3 Foundation for NPoS validator elections. It distributes nominator stake across validators to maximize the minimum backing of any elected validator. |
| **Preimage** | The full call data associated with a governance proposal, stored on-chain by hash reference. Preimages allow proposals to be submitted compactly (by hash) and the actual call data to be uploaded separately before execution. |
| **Premine** | The initial allocation of 888,888,888 TAL distributed at genesis to founding validators, subject to a 10-year vesting schedule. |
| **Proxy** | An account authorized to submit extrinsics on behalf of another account, optionally restricted to a specific category of calls (e.g., governance-only, staking-only). Proxies can be configured with time delays for additional security. |
| **Quorum Biasing** | A mechanism by which the Council can adjust the effective supermajority threshold for a referendum. Unanimous council support lowers the threshold (supermajority-to-fail), while council opposition raises it. |
| **Referendum** | A binding on-chain vote on a specific proposal. Referenda have a fixed voting period, after which votes are tallied using coin-weighting and conviction multipliers. Approved referenda execute the associated dispatchable call after an enactment delay. |
| **Registrar** | A trusted entity authorized to provide identity judgements on-chain. Registrars are appointed by the Council (or root) and each sets its own fee and verification standards. Up to 20 registrars may operate concurrently. |
| **Runtime** | The state transition function of the blockchain, compiled to WebAssembly. The runtime defines all business logic — how blocks are validated, how state changes are applied, and how extrinsics are processed. It can be upgraded on-chain without a hard fork. |
| **Session** | A period within an era during which the validator set and session keys remain constant. Each era consists of 6 sessions. At session boundaries, new keys may rotate in and validator performance metrics are evaluated. |
| **Slash** | A punitive reduction of a validator's (and their nominators') bonded stake in response to misbehavior such as equivocation or prolonged unresponsiveness. Slashed funds are sent to the treasury unless cancelled by the Council. |
| **Slot** | A fixed 6-second time interval in the BABE consensus protocol. Each slot may produce zero or more candidate blocks, with exactly one being selected for the canonical chain. |

| Term | Definition |
|---|---|
| **Smart Contract** | A user-deployed, gas-metered WebAssembly program that runs within pallet_contracts. Contracts maintain their own storage, can transfer tokens, call other contracts, and (subject to whitelisting) interact with other runtime pallets. |
| **SS58** | The address encoding format used by Substrate-based blockchains. Taler uses prefix **10960**, producing visually distinctive addresses that are not easily confused with those of other networks. |
| **Staking** | The process of locking tokens as collateral to secure the network. Validators stake directly; nominators stake by backing validators. Stakers earn deterministic interest-based rewards and face slashing for validator misbehavior. |
| **Sudo** | A privileged module present during the network bootstrap phase that grants a single designated account unrestricted root access to all dispatchable calls. Removed permanently once governance is operational. |
| **TAL** | The native token of the Taler blockchain. TAL is used for transaction fee payment, staking, governance voting, smart contract execution, identity deposits, and all other on-chain economic activities. |
| **TALER** | The primary denomination of TAL. 1 TALER = 100 TCENTS = 100,000 TMILLICENTS = $10^{12}$ base units. |
| **Technical Committee** | A governance body appointed by the Council, composed of technically proficient members responsible for fast-tracking urgent proposals, cancelling dangerous pre-vote proposals, and protecting the network against malicious governance actions. |
| **Treasury** | An on-chain fund accumulated from 80% of transaction fees and 100% of slashing penalties. Treasury funds are disbursed through council-approved proposals, bounties, and tips to finance network development and ecosystem growth. |
| **Trie** | A Base-16 Modified Merkle Patricia Trie data structure used to store the blockchain's state. The trie root hash is included in each block header, providing a cryptographic commitment to the entire chain state at that block height. |
| **Unbonding** | The process of withdrawing tokens from staking. Unbonding initiates a 28-day waiting period during which the tokens remain locked and subject to slashing, after which they become freely transferable. |
| **Validator** | A node operator who participates in block production and finality consensus. Validators must bond stake as collateral and maintain reliable infrastructure. They earn staking rewards and commission from nominators, but face slashing for misbehavior. |
| **Vesting** | A time-lock mechanism that restricts the transferability of tokens according to a predefined release schedule. Vested tokens count |

| Term | Definition |
|---|---|
| | toward an account's total balance for staking and governance purposes but cannot be transferred until unlocked. |
| **Vivid Staking** | A Taler-specific mechanism that allows stakers to voluntarily lock their bonded funds for 1–12 months in exchange for an additional 0.5% annual interest per month of lock-up, up to a maximum bonus of 6% (for a total of 18% APR). |
| **VRF** | Verifiable Random Function. A cryptographic primitive used by BABE to randomly and verifiably assign block authoring slots to validators. VRF outputs are unpredictable before computation but verifiable after, ensuring fair slot assignment. |
| **Wasm** | WebAssembly. A portable, binary instruction format used as the compilation target for Taler's runtime and smart contracts. Wasm enables deterministic execution across heterogeneous hardware and supports forkless runtime upgrades. |
| **Weight** | A measure of the computational resources consumed by an extrinsic, expressed in picoseconds of execution time on reference hardware. Weights are used to meter block capacity, compute transaction fees, and ensure that blocks can always be produced within the target block time. |

## References


1. Parity Technologies. *Substrate Developer Hub*. https://docs.substrate.io/
2. Stewart, A.; Kokoris-Kogias, E. *GRANDPA: A Byzantine Finality Gadget*. arXiv:2007.01560, 2020.
3. Web3 Foundation. *BABE — Blind Assignment for Blockchain Extension*. https://research.web3.foundation/Polkadot/protocols/block-production/Babe
4. Web3 Foundation. *GRANDPA Finality*. https://research.web3.foundation/Polkadot/protocols/finality
5. Burdges, J. et al. *Overview of Polkadot and its Design Considerations*. arXiv:2005.13456, 2020.
6. David, B. et al. *Ouroboros Praos: An Adaptively-Secure, Semi-Synchronous Proof-of-Stake Blockchain*. EUROCRYPT 2018.
7. Burdges, J. *Schnorrkel: Schnorr VRFs and Signatures on the Ristretto Group*. https://github.com/w3f/schnorrkel
8. Polkadot Wiki. *Governance V1*. https://wiki.polkadot.network/docs/learn-governance
9. Polkadot Wiki. *Treasury*. https://wiki.polkadot.network/docs/learn-treasury
10. Polkadot Wiki. *Staking*. https://wiki.polkadot.network/docs/learn-staking
11. Phragmén, E. *Sequential Phragmén Method for Multi-Winner Elections*. Adapted by Web3 Foundation for NPoS validator election.
12. Parity Technologies. *ink! Smart Contract Language*. https://use.ink/