DIG Network

A Proof-of-Stake Layer 2 on Chia

Michael Taylor

Part 1, Version 1.0 · May 2026

Abstract

DIG is a Layer 2 PoS blockchain. Validator custody, finality, and state-rollup audit live on Chia L1. Validators register by locking XCH in per-validator `registration_coins`. Entry and exit happen only at checkpoint events on a singleton lineage. Each epoch ends in a checkpoint block whose header-merkle root, state root, and validator-set commitment post to Chia L1 in one SpendBundle. A CLVM puzzle verifies the SpendBundle. It runs a Groth16 SNARK plus a BLS aggregate check. Together they prove a majority of the current validator set signed the epoch. L1 cost is constant in validator count up to a circuit cap of 20,000.

DIG inherits Chia's coinset model, CLVM execution, and curve-shape emission (halvings, non-zero asymptote). DIG departs from Chia on premine: Chia had a prefarm, DIG has none. DIG adds Ethereum's validator lifecycle (activation queue, attestation flags, inactivity leak, slashing), Polygon's checkpoint-rollup pattern, and a Chia-native L1 anchoring scheme in five CLVM puzzles.

The reward token is DIG. Smallest unit is the dojo. 1 DIG = 1,000,000,000 dojos (10^9). DIG is paid as block subsidy. It is the gas asset on the DIG blockchain. In Part 2, it is the payment asset for data storage in the DFSP protocol. 50% of transaction fees go to validators. The other 50% is burned, EIP-1559 style.

Chia L1 is the security and custody layer. DIG L2 is the application layer. XCH becomes a yield-bearing staking asset. Zero changes to Chia consensus. DIG preserves the public interfaces of Chia's networking, peer, wallet, and serialization protocols; the only difference is the network itself, so existing Chia wallets, explorers, and applications interoperate with minimal migration.

This paper covers consensus, L1 anchoring, validator lifecycle, and binaries. Part 2 covers DFSP, the decentralized file-storage protocol that runs on top. Part 3 specifies Digstore, the default content-addressable archive format that DFSP serves as opaque encrypted payload.

Table of Contents

Part 1: Foundation

1. The Four Big Ideas
2. Positioning
3. Design Heritage

Part 2: L1 Anchoring and Consensus

4. L1 Anchoring: The Five Puzzles
5. Validator Lifecycle
6. Epoch Geometry and Finality
7. Consensus and Slashing

Part 3: Economics and Governance

8. Tokens and Economics
9. Governance

Part 4: Implementation

10. Networking and Binaries
11. Storage
12. The chia-l2-consensus Anchor Crate

13. Security Properties

Part 5: Status

14. Status and Roadmap
15. What Comes Next: Part 2 (DFSP)
16. Acknowledgements and References

Part 1: Foundation

1. The Four Big Ideas

1. **Validators register on Chia L1, and only on Chia L1.** No staking pool. No share token. No ERC-20 analogue. A validator's stake is a `registration_coin` on Chia L1, curried with their BLS pubkey and a withdrawal puzzle hash. The coin's existence is the proof. Spend the coin, lose the stake.
2. **The active validator set is a Merkle root on Chia L1.** A Chia L1 singleton, the `checkpoint_singleton`, carries the current `validator_merkle_root`. A pubkey can sign DIG L2 blocks only when it is a member of that root. A `registration_coin` is necessary but not sufficient.
3. **Finality is a Groth16 proof plus a BLS aggregate signature.** Every epoch, the active set produces a checkpoint binding the new state root, validator Merkle root, exit-ledger root, four DFSP registry roots (Part 2 §13), and epoch number. Chia L1 verifies it with a CLVM puzzle running a Groth16 SNARK proving k signers are tree members with 2k > n, plus a BLS aggregate check that those k signed the digest. L1 cost is constant in k and n up to `MAX_SIGNERS` = 20,000.
4. **L2 state is committed to Chia L1 every epoch.** The `state_root` in each checkpoint is the root of a sparse Merkle tree over every DIG L2 coin at epoch close. Any observer can verify any coin's existence at any past epoch with a ~1 KiB Merkle proof against the L1-confirmed checkpoint. No DIG node trust required.

Chia L1 is the security and custody layer. DIG L2 is the application layer.

2. Positioning

Chia L1 is good at what it was designed for: low-energy PoST consensus, coinset accounting, CLVM puzzles, offers, CATs, DataLayer. It was not built for high-throughput application workloads.

Capability	Chia L1	DIG L2
Block cadence	~18.75 s (timelord-paced)	Proposer-driven within an epoch
Soft finality	Chia confirmations	BLS quorum attestation
Hard finality	Chia confirmations	Chia L1 checkpoint confirmation
Validator-set economics	None (farmer rewards)	Stake-gated BFT (one vote per validator)
Stake asset	None (PoST)	XCH locked on Chia L1
Native asset	XCH	DIG
Native-asset role	Payment, storage	Reward, gas, DFSP payment
Reward to consensus role	XCH to farmers	DIG to validators

What DIG delivers to the Chia ecosystem:

- **XCH becomes a yield-bearing staking asset.** Validators lock XCH. At ~$5,000 USD-equivalent per validator at launch, the staking surface creates a demand sink proportional to validator-set size. Governance sets the mojo amount. The dollar peg is design intent, not on-chain.
- **Zero changes to Chia consensus.** DIG is five CLVM puzzles plus an off-chain validator network producing SpendBundles. Uses existing Chia primitives: singletons, announcements, `ASSERT_HEIGHT_RELATIVE`, BLS verification. No soft fork. No hard fork. No CHIP touching consensus. DIG nodes also speak Chia's networking, peer, wallet, and serialization protocols verbatim; the only difference is the network itself (§10.2).
- **Fair launch.** No premine. No governance token. Chia had a prefarm; DIG does not. DIG follows Chia's emission curve shape (halvings, non-zero asymptote) but starts from zero supply at genesis. Voting weight is validator-set membership, not a token.
- **DIG cannot harm Chia consensus.** A DIG outage halts L2 block production at worst. L1 puzzles continue to behave. `registration_coins` remain spendable. Collateral remains safe.

3. Design Heritage

DIG borrows. The borrows are documented. The combination is the contribution: Chia primitives, Ethereum staking shape, Polygon checkpoint pattern.

Component	Heritage	DIG adaptation
Coinset + CLVM execution	Chia	Verbatim. L2 transactions are CLVM SpendBundles. Same tooling.
Networking, peer, wallet, and serialization protocols	Chia	Public interfaces preserved exactly: same message envelope, Streamable encoding, handshake fields, certificate scheme, peer discovery, and JSON-RPC shapes. The only difference is the network itself: DIG advertises its own `network_id` (§10.2). The node behind the interface is an independent Rust implementation on `chia-sdk-client`, not a code copy.
Validator record	Ethereum beacon-chain	`DigValidator` mirrors Ethereum's Validator container. Adds `l1_registration_coin_id` and `withdrawal_puzzle_hash`.
Activation queue, churn limit	Ethereum	Verbatim formula. Activations land in the next checkpoint.
Participation flags	Ethereum Altair	`TIMELY_SOURCE`, `TIMELY_TARGET`, `TIMELY_HEAD`. Verbatim in `dig-slashing::participation`.
Inactivity score + leak	Ethereum Bellatrix	Verbatim formula and constants.

Slashing offenses	Ethereum + custom	Four correlation-penalty offenses: `ProposerEquivocation`, `InvalidBlock`, `AttesterDoubleVote`, `AttesterSurroundVote`. (Participation and inactivity penalties from Altair/Bellatrix reduce balance but are not classified as slashing offenses; see §7.)
Checkpoint-as-anchor	Polygon Heimdall	Chia singleton spend verified by Groth16 + BLS. Not a Tendermint multisig.
Emission curve shape	Chia	Halving schedule with non-zero asymptote. DIG starts from zero supply (no premine).

What DIG does not inherit:

- **Proof-of-space-and-time.** Stake-gated BFT, one vote per validator. No farmer, no timelord, no harvester.
- **EVM.** No smart-contract VM. Transactions are CLVM SpendBundles.
- **Governance token.** Voting weight is validator-set membership.
- **Beacon-chain / execution-layer split.** One block format.
- **Delegation.** v1 validators stake directly. Not in v1 scope.

Part 2: L1 Anchoring and Consensus

4. L1 Anchoring: The Five Puzzles

Five CLVM puzzles. All live in `chia-l2-consensus`, sourced in Rue. Every L1-touching operation routes through one. Nothing else on Chia L1 needs to know DIG exists.

4.1 network_coin (singleton)

Singleton lineage on Chia L1. Instantiated once per DIG network. Mints `registration_coins`. Carries `collateral_amount` as the only mutable parameter directly relevant to minting. All other governance-mutable parameters live in the `ParameterRegistry` singleton (§4.6).

Three spend paths:

- **Mint.** Emits a child `registration_coin` curried with the validator's BLS pubkey, withdrawal puzzle hash, current `collateral_amount`, and checkpoint singleton launcher ID. Burns the collateral into the registration coin.
- **Parameter update.** Consumes a Groth16 + BLS proof that the validator set agreed (2k > n) on a new `collateral_amount`. Re-creates the network coin with the new value.
- **Recreate.** Pass-through. Advances the singleton lineage with no state change.

Pseudocode:

```
puzzle network_coin {
 curried:
   SINGLETON_STRUCT              // Chia singleton wrapper
   GROTH16_VERIFICATION_KEY      // shared with checkpoint_singleton + ParameterRegistry
   CHECKPOINT_SINGLETON_ID
   PARAMETER_REGISTRY_ID
   NETWORK_ID
   REGISTRATION_COIN_MOD_HASH
   collateral_amount             // governance-mutable; XCH mojos
 solution:
   spend_path                    // Mint | ParameterUpdate | Recreate
   spend_args

 on Mint(bls_pubkey, withdrawal_puzzle_hash, payment_coin_id):
   require coin_amount(payment_coin_id) == collateral_amount
   new_reg_inner = curry(
       REGISTRATION_COIN_MOD_HASH,
       bls_pubkey,
       CHECKPOINT_SINGLETON_ID,
       withdrawal_puzzle_hash,
       collateral_amount,
       NETWORK_ID
   )
   return [
       ASSERT_COIN_SPENT(payment_coin_id),
       CREATE_COIN(puzzle_hash(new_reg_inner), collateral_amount),
       CREATE_COIN(self_puzzle_hash, 1)          // recreate singleton
   ]

 on ParameterUpdate(new_collateral_amount, groth16_proof,
                    aggregate_pubkey, bls_aggregate_sig,
                    validator_merkle_root, validator_count, epoch):
   digest = sha256(
       DOMAIN_TAG_COLLATERAL_UPDATE || PROTOCOL_VERSION || NETWORK_ID ||
```

```
        u64_be(epoch) || validator_merkle_root || u64_be(validator_count) ||
        u128_be(new_collateral_amount)
    )
    require groth16_verify(
        GROTH16_VERIFICATION_KEY,
        groth16_proof,
        public_inputs = [
            validator_merkle_root, validator_count,
            aggregate_pubkey.x_hi, aggregate_pubkey.x_lo,
            aggregate_pubkey.y_hi, aggregate_pubkey.y_lo,
            digest, NETWORK_ID, epoch
        ]
    )
    require bls_verify(aggregate_pubkey, digest, bls_aggregate_sig)
    new_self = curry_with(collateral_amount := new_collateral_amount)
    return [
        CREATE_COIN(puzzle_hash(new_self), 1)
    ]

  on Recreate():
    return [
        CREATE_COIN(self_puzzle_hash, 1)
    ]
}
```

4.2 registration_coin

One coin per active validator. Curried with BLS pubkey, checkpoint singleton ID, withdrawal puzzle hash, and a snapshot of `collateral_amount`. The coin's existence on Chia L1 is the only proof of stake.

Spent only at exit. Permissionless: anyone can construct the bundle. The bundle asserts a checkpoint exit-query announcement (a fresh checkpoint spend whose exit-ledger root contains an entry releasing this validator). The exit spend consumes the `registration_coin` and creates a `withdraw_delay_coin` carrying the unslashed amount (attested by Groth16 + BLS). The slashed portion, if any, is enforced by not creating a coin for it (see §4.5).

The `registration_coin` puzzle accepts no solution-provided conditions. All semantics are in the puzzle. The spender provides only proofs. This is the SEC-008 invariant in `chia-l2-consensus`.

Pseudocode:

```
puzzle registration_coin {
  curried:
    BLS_PUBKEY                     // validator's BLS signing key
    CHECKPOINT_SINGLETON_ID        // anchors which checkpoint releases this coin
    WITHDRAWAL_PUZZLE_HASH         // where unslashed XCH eventually pays out
    COLLATERAL_AMOUNT_SNAPSHOT     // snapshot at mint time
    NETWORK_ID
    WITHDRAW_DELAY_COIN_MOD_HASH
    WITHDRAW_DELAY_BLOCKS
  solution:
    exit_announcement_coin_id      // the checkpoint coin that emitted it
    unslashed_amount               // attested in the announcement
    epoch                          // attested in the announcement

  on Exit(...):
    // The exit-query announcement is keyed by registration_coin_id, not by
    // pubkey alone, so re-registration (§5.7) cannot replay an exit.
    expected_announcement = sha256(
        DOMAIN_TAG_EXIT_QUERY || PROTOCOL_VERSION || NETWORK_ID ||
        coin_id(self) ||                    // this registration_coin's id
        BLS_PUBKEY ||
        u128_be(unslashed_amount) ||
```

```
        u64_be(epoch)
    )
    require ASSERT_PUZZLE_ANNOUNCEMENT(
        exit_announcement_coin_id,
        expected_announcement
    )
    require unslashed_amount <= COLLATERAL_AMOUNT_SNAPSHOT  // cannot exceed stake
    delay_inner = curry(
        WITHDRAW_DELAY_COIN_MOD_HASH,
        WITHDRAWAL_PUZZLE_HASH,
        unslashed_amount,
        WITHDRAW_DELAY_BLOCKS
    )
    return [
        CREATE_COIN(puzzle_hash(delay_inner), unslashed_amount)
        // Slashed portion = COLLATERAL_AMOUNT_SNAPSHOT - unslashed_amount
        // is destroyed by NOT creating a coin for it.
    ]
}
```

4.3 checkpoint_singleton

Singleton lineage on Chia L1. State curried on each recreation:

Field	Type	Meaning
`state_root`	Bytes32	L2 state rollup root at this epoch
`epoch`	u64	Incremented by 1 per spend
`validator_merkle_root`	Bytes32	Sparse Merkle root over active validator set
`validator_count`	u64	Number of active validators
`exit_ledger_root`	Bytes32	Append-only merkle log of exit entries

Protocols built on DIG L2 (Part 2's DFSP among them) commit additional registry roots into the checkpoint digest (§6.5), not into the curried singleton state above. Those roots are deterministically recomputable from L2 state and need not be carried forward by the singleton.

Three spend paths, all verified on-chain:

1. **Checkpoint update.** The only state-changing path. Spender provides a Groth16 proof (k signers are tree members; 2k > `validator_count`; their G1 sum equals the advertised aggregate key) and a BLS aggregate signature over the new state digest. The puzzle recreates the singleton with new state, increments epoch, and emits announcements that downstream `registration_coin` exits can assert.
2. **Membership query.** Read-only. Emits "pubkey P is a member of the current validator set at epoch E". Used by L2 governance and external observers.
3. **Exit query.** Read-only. Emits "registration_coin_id C with pubkey P appears in the exit ledger with unslashed amount U at epoch E". Used by `registration_coin` exit spends; keyed by `registration_coin_id` rather than pubkey alone, because re-registration (§5.7) permits multiple coins per pubkey over a validator's lifetime.

All three re-create the singleton. Read-only paths preserve state.

Pseudocode:

```
puzzle checkpoint_singleton {
 curried:
   SINGLETON_STRUCT
   GROTH16_VERIFICATION_KEY
   NETWORK_ID
   NETWORK_COIN_LAUNCHER_ID
   state_root
   epoch
   validator_merkle_root
   validator_count
   exit_ledger_root
 solution:
   spend_path                  // Update | MembershipQuery | ExitQuery
   spend_args

 on Update(new_state_root, new_validator_merkle_root,
           new_validator_count, new_exit_ledger_root,
           header_merkle_root, groth16_proof, aggregate_pubkey,
           bls_aggregate_sig,
           // when DFSP active:
           collateral_registry_root, store_registry_root,
           node_registry_root, retrieval_proofs_root):
   digest = sha256(
       DOMAIN_TAG_CHECKPOINT || PROTOCOL_VERSION || NETWORK_ID ||
       new_state_root || new_validator_merkle_root ||
       u64_be(new_validator_count) || u64_be(epoch + 1) ||
       new_exit_ledger_root || header_merkle_root ||
       // DFSP extension (zero hashes if not active):
       collateral_registry_root || store_registry_root ||
       node_registry_root || retrieval_proofs_root
   )
   require groth16_verify(
       GROTH16_VERIFICATION_KEY,
       groth16_proof,
       public_inputs = [
           validator_merkle_root,          // PREVIOUS root signs new state
           validator_count,
           aggregate_pubkey.x_hi, aggregate_pubkey.x_lo,
           aggregate_pubkey.y_hi, aggregate_pubkey.y_lo,
           digest,
           NETWORK_COIN_LAUNCHER_ID,
           epoch,
           epoch + 1
       ]
   )
   require bls_verify(aggregate_pubkey, digest, bls_aggregate_sig)
   new_self = curry_with(
       state_root := new_state_root,
       epoch := epoch + 1,
       validator_merkle_root := new_validator_merkle_root,
       validator_count := new_validator_count,
       exit_ledger_root := new_exit_ledger_root
   )
   return [
       CREATE_COIN(puzzle_hash(new_self), 1),
       // Emit announcement consumable by exit spends in next epoch:
       CREATE_PUZZLE_ANNOUNCEMENT(new_exit_ledger_root)
   ]

 on MembershipQuery(bls_pubkey, merkle_path):
   require merkle_verify(validator_merkle_root, bls_pubkey, merkle_path)
   msg = sha256(
       DOMAIN_TAG_MEMBERSHIP_QUERY || NETWORK_ID ||
       bls_pubkey || u64_be(epoch)
   )
   return [
       CREATE_COIN(self_puzzle_hash, 1),
       CREATE_PUZZLE_ANNOUNCEMENT(msg)
   ]

 on ExitQuery(registration_coin_id, bls_pubkey, unslashed_amount,
              exit_ledger_merkle_path):
   leaf = sha256(
       registration_coin_id || bls_pubkey || u128_be(unslashed_amount)
   )
   require merkle_verify(exit_ledger_root, leaf, exit_ledger_merkle_path)
   msg = sha256(
       DOMAIN_TAG_EXIT_QUERY || PROTOCOL_VERSION || NETWORK_ID ||
       registration_coin_id || bls_pubkey ||
       u128_be(unslashed_amount) || u64_be(epoch)
   )
   return [
       CREATE_COIN(self_puzzle_hash, 1),
       CREATE_PUZZLE_ANNOUNCEMENT(msg)
   ]
}
```

4.4 withdraw_delay_coin

Created by a `registration_coin` exit spend. One per exiting validator. Curried with the validator's withdrawal puzzle hash and the unslashed amount. Holds that amount as its mojo value.

Enforces `ASSERT_HEIGHT_RELATIVE` for `withdraw_delay_blocks` (default ~24,000 L1 blocks, about five days). After the delay, anyone can spend the coin to the withdrawal puzzle hash. DIG's `withdrawable_epoch`: a time-locked guarantee that no further L2 evidence can slash this validator before funds release.

Pseudocode:

```
puzzle withdraw_delay_coin {
 curried:
   WITHDRAWAL_PUZZLE_HASH
   UNSLASHED_AMOUNT             // == this coin's mojo value at creation
   WITHDRAW_DELAY_BLOCKS
 solution:
   ()                          // no solution-provided conditions; SEC-008 invariant

 on Spend():
   return [
       ASSERT_HEIGHT_RELATIVE(WITHDRAW_DELAY_BLOCKS),
       // Permissionless spend: anyone can submit; funds always flow to
       // the curried withdrawal_puzzle_hash.
       CREATE_COIN(WITHDRAWAL_PUZZLE_HASH, UNSLASHED_AMOUNT)
   ]
}
```

4.5 Slashing on L1

Slashing is partial, L2-attested, and enforced as L1 subtraction.

L2 records offenses in per-epoch participation accounting. At exit, the L2 majority produces a Groth16 + BLS-attested unslashed amount: `unslashed = original_collateral - slashed_portion`. The `registration_coin` exit puzzle reads this amount from the checkpoint exit-ledger announcement and curries it into the `withdraw_delay_coin`.

The slashed portion is enforced by not creating a coin for it. Chia L1 does not encode slashing rules. It verifies that the L2 majority signed off on the recoverable number. Whether the slashed portion is redistributed or retained is an L2 economic question, mutable by governance.

Three properties:

- **Validators cannot be slashed more than their stake.** Exit puzzle bounds payout at `original_collateral`. The L2 proof produces only the unslashed portion.
- **Chia L1 is slashing-agnostic.** L1 puzzles verify "the majority agreed on this number" and apply it.
- **Slashing is socially recoverable.** If the L2 validator set later concludes a slashing was in error (appeals pathway in `dig-slashing`), a subsequent checkpoint can rewrite the exit ledger entry before the `withdraw_delay_coin` is created.

4.6 ParameterRegistry

Singleton lineage on Chia L1. Holds the network's governance-mutable parameter set in its curried state. Separate from `network_coin` so that the parameter surface can grow without disturbing the minting path. Mutated only by the governance circuit (§9.2).

The parameter snapshot includes consensus parameters (reward quotients, churn limits, fee burn ratio, slashing thresholds, inactivity-leak constants) and an extension surface for protocols built on top of DIG L2. Part 2's DFSP adds storage-protocol parameters (burn rate, bond split, retrieval quotients, `MIN_NODE_STAKE`, and others) into the same parameter snapshot.

Two spend paths:

- **Parameter update.** Consumes a Groth16 + BLS proof that the validator set agreed (2k > n) on a new parameter snapshot, signed via the governance circuit's digest (§9.2). Re-creates the singleton with the new snapshot. Same circuit shape as the checkpoint update.
- **Read.** Read-only. Emits the current parameter snapshot hash. Consulted by L2 puzzles and off-chain components that need current parameter values.

L2 fullnodes observe parameter updates by walking the `ParameterRegistry` lineage on Chia L1, exactly as they walk the `network_coin` and `checkpoint_singleton` lineages (§12). The current snapshot is cached in memory and consulted by epoch-boundary operations.

`ParameterRegistry` is the only governance state surface that survives across DIG protocols: extensions add parameters to the same snapshot rather than introducing their own registries.

Pseudocode:

```
puzzle parameter_registry {
 curried:
   SINGLETON_STRUCT
   GROTH16_VERIFICATION_KEY    // shared with checkpoint_singleton + network_coin
   NETWORK_ID
   parameter_snapshot_hash     // sha256 of canonical parameter struct
 solution:
   spend_path                  // Update | Read
   spend_args

 on Update(new_parameter_snapshot_hash,
           governance_proof_groth16,
           aggregate_pubkey,
           bls_aggregate_sig,
           validator_merkle_root, validator_count, epoch):
   digest = sha256(
       DOMAIN_TAG_PARAMETER_UPDATE || PROTOCOL_VERSION || NETWORK_ID ||
       u64_be(epoch) || validator_merkle_root || u64_be(validator_count) ||
       new_parameter_snapshot_hash
   )
   require groth16_verify(
       GROTH16_VERIFICATION_KEY,
       governance_proof_groth16,
       public_inputs = [
           validator_merkle_root, validator_count,
           aggregate_pubkey.x_hi, aggregate_pubkey.x_lo,
           aggregate_pubkey.y_hi, aggregate_pubkey.y_lo,
           digest, NETWORK_ID, epoch
       ]
   )
   require bls_verify(aggregate_pubkey, digest, bls_aggregate_sig)
   new_self = curry_with(
       parameter_snapshot_hash := new_parameter_snapshot_hash
   )
   return [
       CREATE_COIN(puzzle_hash(new_self), 1),
       CREATE_PUZZLE_ANNOUNCEMENT(
           sha256(DOMAIN_TAG_PARAMETER_PUBLISH || new_parameter_snapshot_hash)
       )
```

```
    ]

 on Read():
    return [
        CREATE_COIN(self_puzzle_hash, 1),
        CREATE_PUZZLE_ANNOUNCEMENT(
            sha256(DOMAIN_TAG_PARAMETER_READ || parameter_snapshot_hash)
        )
    ]
}
```

4.7 Genesis Configuration

The five puzzles describe steady-state operation. Genesis is the one-time event that brings them into existence. Genesis state is pre-committed in `NetworkConfig` (§12) and embedded in every client binary; mainnet, testnet, and any future shard have distinct `NetworkConfig` files and are cryptographically distinct deployments (§12).

`NetworkConfig` commits the following at genesis:

- **Genesis network_coin launcher.** A one-time pre-mainnet ceremony launches the `network_coin` singleton from a publicly-burned XCH coin. The launcher ID is committed in `NetworkConfig`. There is no mint authority and no admin key; the singleton is established by a publicly-witnessed coin spend whose construction is reproducible by any party from `NetworkConfig`.
- **Genesis checkpoint_singleton launcher and initial curried state.** Includes `initial_state_root` (the SMT root over a zero-coin L2 state), `initial_validator_merkle_root`, `initial_validator_count`, `initial_exit_ledger_root` (empty), and epoch = 0. The genesis `validator_merkle_root` commits to a pre-registered initial validator set; their pubkeys and the corresponding `registration_coins` are also committed in `NetworkConfig`.
- **Genesis ParameterRegistry launcher and initial parameter snapshot.** All Category-A parameters at launch values.
- **Verification keys.** Curried into the three singletons. Produced by the pre-mainnet MPC ceremony (§14).
- **Genesis validator-set registrations.** Each genesis validator pre-funds the `collateral_amount` of XCH and constructs a `registration_coin` spend such that the resulting coin IDs are deterministic and committable. The set of (BLS pubkey, `withdrawal_puzzle_hash`, `registration_coin_id`) tuples committed in `NetworkConfig` forms the initial `validator_merkle_root`.

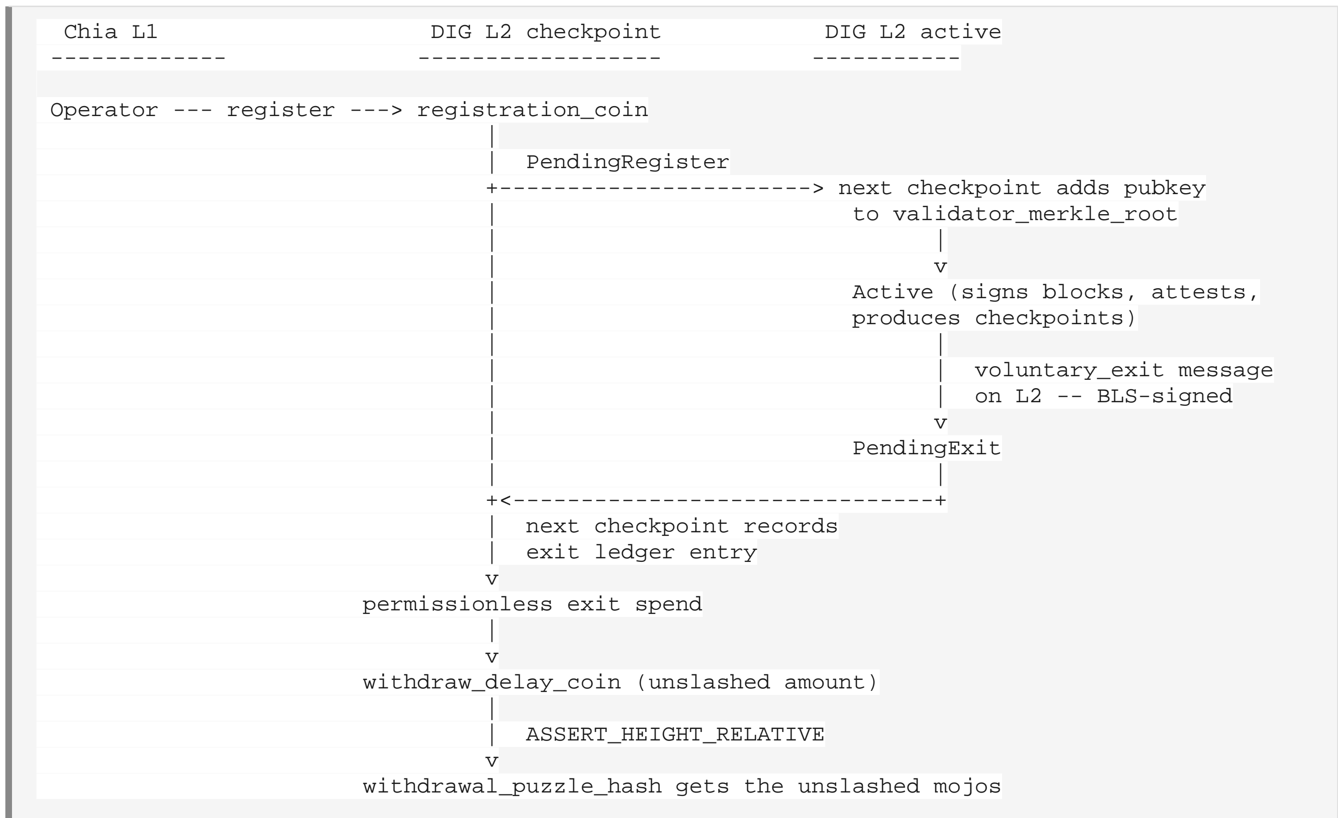
After genesis, the network operates under the steady-state rules. The first checkpoint update (epoch 0 → epoch 1) is signed by the genesis validator set against the curried `initial_validator_merkle_root`, exactly like every subsequent checkpoint. `PendingRegister` activates new validators starting from epoch 1; voluntary and forced exits are processed identically from epoch 0 onward.

The "fair launch" property (no premine) holds because no DIG exists at genesis: every reward, including the first block's subsidy, is minted by protocol issuance under the same rules that apply to every subsequent block (§7.5).

5. Validator Lifecycle

Ethereum lifecycle shape: BLS identity, queued activation, queued exit, delayed withdrawal. The only deviation is that DIG's deposit is a coin on Chia L1, not a Solidity contract on its own L1.

5.1 Phases



```
 Chia L1                     DIG L2 checkpoint           DIG L2 active
-------------               ------------------          -----------

Operator --- register ---> registration_coin
                                 |
                                 |  PendingRegister
                                 +------------------------> next checkpoint adds pubkey
                                 |                           to validator_merkle_root
                                 |                                  |
                                 |                                  v
                                 |                          Active (signs blocks, attests,
                                 |                          produces checkpoints)
                                 |                                  |
                                 |                                  |  voluntary_exit message
                                 |                                  |  on L2 -- BLS-signed
                                 |                                  v
                                 |                          PendingExit
                                 |                                  |
                                 +<---------------------------------+
                                 |  next checkpoint records
                                 |  exit ledger entry
                                 v
                        permissionless exit spend
                                 |
                                 v
                        withdraw_delay_coin (unslashed amount)
                                 |
                                 |  ASSERT_HEIGHT_RELATIVE
                                 v
                        withdrawal_puzzle_hash gets the unslashed mojos
```

5.2 Registration

1. Operator constructs a `network_coin` spend that mints a `registration_coin` curried with their BLS pubkey and withdrawal puzzle hash. Funded by an XCH input matching `collateral_amount`.
2. On L1 confirmation, the `registration_coin` exists. Any DIG fullnode walking the singleton lineage observes it and adds the pubkey to `PendingRegister`, sorted by (`l1_confirmed_height`, `coin_id`).
3. At the next checkpoint, the building fullnode reads `PendingRegister`, applies the activation churn limit (Ethereum's per-epoch formula), and includes the resulting `validator_merkle_root` in the new checkpoint state.
4. Once L1 confirms the checkpoint, the validator is Active.

Activation lag is on the order of one epoch.

5.3 Active duties

- **Block proposal when selected by the proposer-selection rule.** The primary proposer for each slot is deterministic from the previous block's state root, the epoch number, and the slot index within the epoch. If the primary proposer does not broadcast a block within the slot timeout (default 2 × gossip-RTT, governance-mutable), a deterministic fallback proposer for the same slot takes over. The fallback sequence is a permutation of the active validator set under the same seed. A block whose proposer field does not match either the primary or an in-sequence fallback for its slot is invalid.
- **Attestation on every block proposed by another validator,** signing the three Altair participation flags (`TIMELY_SOURCE`, `TIMELY_TARGET`, `TIMELY_HEAD`).
- **Checkpoint signing** at the end of every epoch.

- **Voluntary-exit signing** on request.

Block format, attestation wire format, and proposer-selection rule are in `dig-block` and `chia-l2-consensus`.

5.4 Voluntary exit

1. Validator broadcasts a BLS-signed voluntary-exit message naming their `registration_coin_id`, pubkey, and current epoch. The `registration_coin_id` binds the exit to a specific L1 coin; this matters when re-registration (§5.7) has created multiple coins for the same pubkey over time.
2. Next checkpoint adds an entry to the exit ledger (pubkey → `unslashed_amount`) and includes the new exit-ledger root in checkpoint state.
3. Once the checkpoint is L1-confirmed, anyone may construct the L1 exit SpendBundle. It consumes the `registration_coin`, asserts the exit-query announcement, and creates a `withdraw_delay_coin` with the unslashed amount.
4. After `withdraw_delay_blocks` (~5 days), anyone may spend the `withdraw_delay_coin` to the operator's `withdrawal_puzzle_hash`.

Step (3) is permissionless. The validator does not need to be online. An offline validator's funds will not be stranded.

5.5 Forced exit (slashing)

An L2 offense (§7) lands in the offending epoch's participation flags. The next checkpoint's exit-ledger entry for the validator shows a reduced unslashed amount, and the new `validator_merkle_root` excludes the offender's pubkey. From the L1-confirmed checkpoint onward, the offender can no longer sign blocks or attestations. The remainder of the flow mirrors §5.4 steps 3-4. The operator does not opt in. The network exits them.

5.6 Withdrawal

`withdraw_delay_coin` becomes spendable after `withdraw_delay_blocks` L1 blocks (default ~5 days). Permissionless: the puzzle pays to the curried `withdrawal_puzzle_hash`, so anyone can submit the spend. Funds always go to the operator's address. Protects against censorship of the withdrawal step.

5.7 Re-registration

A previously-exited validator can register again. New `registration_coin` is a fresh L1 coin with a fresh `registration_coin_id`. The L2 treats this as a new validator. Participation flags, inactivity score, and slashing state do not carry over. Matches Ethereum. Blocks re-registration replay attacks.

6. Epoch Geometry and Finality

Three-tier finality. Each tier has its own trust assumption and latency budget.

6.1 The three tiers

Tier	Mechanism	Trust	Latency
Proposed	Single validator signature on a block	Trust the proposer	Block proposal cadence
Soft-finalized	BLS quorum attestation (2k > n), gossiped through the validator set	Trust the L2 majority	Time for a quorum of attestations
Hard-finalized	Chia L1 confirms the `checkpoint_singleton` spend committing the epoch	Trust Chia L1	Next checkpoint plus L1 confirmation depth

6.2 Epoch structure

An epoch is a run of L2 blocks ending in a special checkpoint block. The checkpoint block is the final block of every epoch and is structurally distinct from transaction blocks.

- The checkpoint block contains no transactions. It is a consensus artifact only.
- Its payload is a Merkle root over the block headers of every block in the epoch. A light client can prove any block belongs to the epoch with a logarithmic-size proof against this root.
- Its payload commits the new state root, validator Merkle root, exit-ledger root, epoch number, and (when DFSP is active) four parallel DFSP registry roots: `collateral_registry_root`, `store_registry_root`, `node_registry_root`, and `retrieval_proofs_root` (Part 2 §13). That commitment, digested under §6.5, becomes the `checkpoint_singleton` spend submitted to Chia L1. The BLS aggregate signature is computed over that digest and travels with the SpendBundle as a separate component, not inside it.

Three properties:

1. **Self-describing.** Any fullnode can verify that the epoch's transaction blocks belong to epoch E using only the L2 header chain, before L1 confirmation. L1 provides the unforgeability anchor against L2 majority equivocation.
2. **O(log n) inclusion proofs.** A light client proving a past block belongs to a hard-finalized epoch needs a logarithmic-size hash chain against the checkpoint's header-Merkle root, plus the `checkpoint_singleton` lineage proof.
3. **Rollup data separated from consensus data.** Transactions live in transaction blocks. The rollup is one transactionless block per epoch. Closer to Polygon Heimdall than Ethereum's per-slot post-state commitment.

6.3 Block-time model

An epoch is divided into a fixed number of slots (default `EPOCH_TARGET_SLOTS` = 60, governance-mutable). Each slot has a deterministically-selected primary proposer (§5.3) and a slot-timeout window (default ~10 seconds) within which the primary may broadcast. Block production within a slot is proposer-driven: the proposer broadcasts when its block is ready, not on a strict wall-clock. If no block lands by the slot timeout, the deterministic fallback sequence applies. A block becomes soft-finalized when BLS attestations from a quorum land. Cadence is bounded below by gossip latency and above by the slot-timeout window.

Soft finality within an epoch is fast. It does not wait for Chia L1. This is what an L2 buys over Chia L1: low-latency confirmation for the duration of an epoch.

Epoch boundaries are different. The L2 cannot start the next epoch's block production until Chia L1 confirms the prior epoch's checkpoint (see §6.4). Cross-epoch latency is therefore bounded by Chia L1 confirmation depth, not by L2 gossip.

Target epoch cadence at mainnet launch is approximately 10 minutes (60 slots × 10 s slot timeout), set so that L1 confirmation completes well within each epoch and the L2 is never starved waiting for Chia. Exact cadence is a Category-A governance parameter (§9). Downstream protocols may reference this target cadence when deriving epoch-relative time locks: Part 2's `bond_withdraw_delay_epochs` = 720, for example, corresponds to ~5 days at this cadence and mirrors Part 1's `withdraw_delay_blocks`.

6.4 Epoch lifecycle

Each epoch runs through four stages. L1 confirmation is blocking: the next epoch cannot begin producing blocks until the previous epoch's `checkpoint_singleton` spend is confirmed on Chia L1.

```
Block production --> Checkpoint block --> L1 submission --> L1 confirmation --> Next epoch begins
                                                                      ^
                                                              BLOCKING: no new blocks
                                                              until L1 confirms.
```

- **Block production.** Transaction blocks are produced and attested. Fees accumulate. Participation flags are recorded per attestation. An epoch ends deterministically when the `EPOCH_TARGET_SLOTS`-th slot of the epoch produces a checkpoint block (default 60 slots per epoch with ~10-second slot cadence ≈ 10 minutes; exact values are Category-A parameters in `ParameterRegistry` §4.6). The slot reserved as the epoch's checkpoint slot is structurally distinct: it carries no transactions, only the checkpoint commitment. If the primary proposer of the checkpoint slot fails to broadcast, the fallback proposer sequence from §5.3 applies. The epoch cannot end without a checkpoint block.
- **Checkpoint block.** The final block of the epoch. Contains no transactions. Its payload commits the header-Merkle root over every block in the epoch, the new state root, the new validator Merkle root, the new exit-ledger root, the epoch number, and (when DFSP is active) the four DFSP registry roots (Part 2 §13).
- **L1 submission.** The `checkpoint_singleton` SpendBundle is constructed from the checkpoint block's payload, the Groth16 proof, and the BLS aggregate signature. Submitted to Chia L1 via `chia_query::push_tx`.
- **L1 confirmation (blocking).** The L2 halts new block proposals until Chia L1 confirms the checkpoint spend to a depth of `L1_CONFIRMATION_DEPTH` blocks (default 32, governance-mutable in `ParameterRegistry`). 32 Chia blocks ≈ 16 minutes at Chia's 52-second average block time; this exceeds the practical depth at which Chia L1 reorgs become probabilistically negligible. Once confirmed to depth, the epoch is hard-finalized: the new state is the `checkpoint_singleton`'s new curried state, and the next epoch begins.

If Chia L1 reorgs the checkpoint spend out of the canonical chain before reaching `L1_CONFIRMATION_DEPTH`, the L2 detects the reorg by walking the `checkpoint_singleton` lineage from a deeper L1 anchor and resumes from the most recent depth-confirmed checkpoint. L2 blocks produced after the reorged checkpoint (if any were produced in error) are orphaned. In practice, the blocking model means no L2 blocks exist past an unconfirmed checkpoint; the failure case is bounded to re-submitting the checkpoint spend on the canonical L1 chain.

The blocking model is deliberate. It guarantees that no L2 block ever exists whose ancestor epoch is not L1-confirmed. There is no "unfinalized tail" of orphaned epochs accumulating during a Chia L1 outage. If Chia L1 stalls, the L2 stalls. When Chia L1 recovers, the L2 resumes with no fork-choice ambiguity.

Block cadence within an epoch is therefore bounded above by validator-set liveness and gossip latency. Cross-epoch cadence is bounded by Chia L1 confirmation depth. The two are decoupled: a fast L2 with a slow L1 confirmation is the expected steady state.

6.5 The checkpoint puzzle

`checkpoint_inner.rue`. Three spend paths in CLVM.

Path 1: Checkpoint update. State-changing. Verifies (a) a Groth16 proof against current `validator_merkle_root`, `validator_count`, and a public input bundle containing new state, and (b) a BLS aggregate signature over the canonical checkpoint digest. Recreates the singleton with new state.

Digest (pre-DFSP form):

```
sha256(state_root || validator_merkle_root || validator_count_be ||
       epoch_be || exit_ledger_root || header_merkle_root || network_id)
```

Extended digest (with DFSP active, from Part 2 §13.6):

```
sha256(state_root || validator_merkle_root || validator_count_be ||
       epoch_be || exit_ledger_root || header_merkle_root || network_id ||
       collateral_registry_root || store_registry_root ||
       node_registry_root || retrieval_proofs_root)
```

Network ID prevents cross-network replay. Epoch prevents within-network replay across epochs. The Groth16 circuit verifies the digest opaquely: it has no semantic understanding of what is being hashed, only that the validator set signed it. The same circuit handles both digest forms. The DFSP registry roots ride in the digest only; they are NOT carried in the `checkpoint_singleton`'s curried state (see §4.3).

Path 2: Membership query. Read-only. Emits "pubkey P is a member of the validator merkle root at epoch E".

Path 3: Exit query. Read-only. Emits "registration_coin_id C with pubkey P appears in the exit ledger with unslashed amount U at epoch E". Asserted by `registration_coin` exit SpendBundles; the `registration_coin_id` binds the announcement to a specific coin so that re-registration (§5.7) cannot replay an exit.

6.6 The Checkpoint Groth16 Circuit

The checkpoint circuit is the heart of L1 verification. It compresses the act of "k of n validators signed this state" into a constant-size proof that Chia L1 verifies in a constant-cost CLVM puzzle. This subsection specifies the circuit at a level sufficient for an implementer to construct it.

Proving system. Groth16 over the BLS12-381 pairing-friendly curve. BLS12-381 is selected for three reasons: (1) it is the same curve used by the validator BLS signatures, so the in-circuit and out-of-circuit cryptography share field arithmetic; (2) Chia's `chia_bls` crate already implements BLS12-381 pairing checks usable from CLVM; (3) Ethereum's beacon chain has produced production-grade Rust implementations of BLS12-381 operations (blst, arkworks, halo2-bls12-381) that the DIG implementation reuses.

Groth16 is selected over more modern systems (PLONK, Halo2, FRI-based STARKs) for one reason: verification cost. Groth16 verification is three pairings plus k+1 scalar multiplications in G1 (where k is the number of public inputs); on BLS12-381 this fits in well under Chia L1's per-block CLVM cost budget. PLONK verification is roughly 2× this cost; FRI-based STARKs are 10–100× more expensive to verify. The trade-off is a trusted setup, which DIG accepts and mitigates via MPC ceremony (§14).

Public inputs. Eight field elements, each in Fr (the scalar field of BLS12-381, prime, 255 bits):

Index	Public Input	Width	Meaning

0	`validator_merkle_root`	1 Fr	Sparse-Merkle-tree root over the active validator set at the start of this epoch
1	`validator_count`	1 Fr	Number of active validators (must be ≤ `MAX_SIGNERS` = 20,000)
2-3	`aggregate_pubkey`	2 Fr	G1 point of the signing quorum's aggregate BLS pubkey, encoded as (x, y) in Fp; each Fp element is split across two Fr elements via field-reduction encoding
4	`checkpoint_digest`	1 Fr	sha256 of the canonical checkpoint payload (§6.5), truncated to fit in Fr
5	`network_coin_launcher_id`	1 Fr	NetworkConfig launcher ID, binds the proof to this specific network
6	`previous_epoch_be`	1 Fr	Previous epoch number; circuit constrains the new epoch is previous + 1
7	`current_epoch_be`	1 Fr	Current epoch number

Eight public inputs is well below the 32-input threshold above which Groth16 verification scales linearly. Verification cost is dominated by the three pairings; pairing cost on BLS12-381 in CLVM is fixed.

Witness layout. The circuit takes the following private witness inputs:

```
witness:
 k: u32                                    // number of signers (k ≤ MAX_SIGNERS)
 signers[k]: array of {                    // the k signing validators
   index: u32                              // leaf index in the SMT
   pubkey: G1                              // BLS pubkey of this signer
   path: array<Fr, TREE_DEPTH>             // SMT inclusion path (32 hash siblings)
   path_bits: array<u1, TREE_DEPTH>        // left/right indicator for each level
 }
```

The witness is large but private; only the eight public inputs travel with the proof.

Constraint count and gadget composition. The circuit's constraints split into three groups:

Gadget	Per-signer cost	Total cost (k ≤ 20,000)
Poseidon-based SMT inclusion (`TREE_DEPTH` = 32)	~3,200 constraints	64M constraints worst-case
BLS12-381 G1 addition (aggregation)	~6,500 constraints	130M constraints worst-case

Equality / boolean / arithmetic glue	~50 constraints	~1M constraints

Worst-case constraint count at k = 20,000 is approximately 200M constraints. This is large but tractable on modern proving hardware (a 64-core machine with 128 GB RAM produces such proofs in ~5 minutes using arkworks-style Groth16 implementations). Proving is performed by the validator quorum's designated proposer of the checkpoint slot; it happens once per epoch. Real-world DIG networks will operate at k well below `MAX_SIGNERS`; for k = 1,000 (typical early-mainnet), the circuit instantiation has approximately 10M constraints and produces in under 30 seconds.

The SMT hash function is Poseidon (with prime field BLS12-381 Fr, t=3 rate, 8 full rounds + 57 partial rounds — the parameters used by arkworks-poseidon). Poseidon is selected over SHA256 because in-circuit SHA256 costs roughly 30,000 constraints per hash; Poseidon costs roughly 200 constraints per hash, a 150× reduction in circuit size for the membership tree.

The three constraint families.

1. **Membership (per signer).** For each of the k signers, the circuit verifies that their pubkey's leaf hash, when combined with the path siblings according to the `path_bits`, reproduces the public input `validator_merkle_root`. Each level of the tree contributes one Poseidon hash. The leaf hash is `Poseidon(pubkey.x, pubkey.y, signer_index)`. Concretely:

```
level_hash = Poseidon(leaf_pubkey.x, leaf_pubkey.y, signer.index)
for level in 0..TREE_DEPTH:
   if path_bits[level] == 0:
       level_hash = Poseidon(level_hash, path[level])
   else:
       level_hash = Poseidon(path[level], level_hash)
assert level_hash == validator_merkle_root
```

2. **Majority.** A single arithmetic constraint:

```
assert 2 * k > validator_count
```

In Fr-arithmetic, this is implemented as `validator_count + 1 ≤ 2k` checked by a range proof on (2k - validator_count - 1) ≥ 0.

3. **Aggregation.** The circuit accumulates the G1 sum of the k signing pubkeys:

```
accum = G1::zero()
for i in 0..k:
   accum = G1::add(accum, signers[i].pubkey)
assert accum == aggregate_pubkey   // public input
```

Each G1 addition is the dominant cost (~6,500 constraints under the arkworks parameter representation using projective coordinates with affine-input gadgets).

The proof commits to a specific (`validator_merkle_root`, `validator_count`, `aggregate_pubkey`, `checkpoint_digest`, `network_coin_launcher_id`, epoch transition) tuple. The CLVM puzzle on Chia L1 receives this proof, verifies it (three pairings on BLS12-381), checks the BLS aggregate signature over the `checkpoint_digest` against the `aggregate_pubkey` (a fourth pairing, outside the SNARK), and recreates the `checkpoint_singleton` on success.

Verification cost on Chia L1. Three pairings inside Groth16 verification + one pairing outside (for the BLS aggregate signature check) = four BLS12-381 pairings total. Each pairing on BLS12-381 costs approximately 1 ms in optimized native code; in CLVM, the `chia_bls` primitive is exposed at constant cost. The puzzle is well within Chia's per-block CLVM cost budget for a singleton spend (the empirical cost is dominated by the four pairings, with public-input scalar multiplications adding negligible cost given 8 inputs).

Why circuit caps at MAX_SIGNERS = 20,000 and TREE_DEPTH = 32. `TREE_DEPTH` = 32 supports 2^32 ≈ 4B leaves, ensuring the tree never needs resizing. `MAX_SIGNERS` = 20,000 is a witness-size cap: at k = 20,000 the proving witness is roughly 64 MB and the prover needs ~128 GB of RAM. Both values are committed in `NetworkConfig` (§12) at deployment and can be raised by an explicit redeployment (which is structurally a new network); they cannot drift through governance because they are circuit constants, not runtime parameters.

Same circuit reused for governance. The §9 governance circuit is structurally identical: same Groth16 setup, same `MAX_SIGNERS`, same `TREE_DEPTH`, same gadgets. Only the public-input semantics differ. The trusted setup ceremony's output verification key serves both circuits, so the L1 cost of governance updates is the same as the L1 cost of checkpoints.

A second BLS pairing check, outside the SNARK and inside the CLVM puzzle, verifies the aggregate signature over the checkpoint digest matches the SNARK-attested aggregate pubkey. Constant-time. Well within Chia's per-block CLVM cost budget for a singleton spend.

6.7 State rollup

`state_root` is the root of a depth-32 sparse Merkle tree over every DIG L2 coin alive at epoch close. Committed in `dig-coinstore`.

- **Light-client coin proofs.** Any party verifies a coin's existence at epoch E with a ~1 KiB Merkle proof against the L1-confirmed `state_root` at epoch E.
- **Historical proofs.** Every past `state_root` is recoverable by walking the `checkpoint_singleton` lineage on Chia L1. No "L2 forgot its history" failure mode.
- **No off-chain data assumption.** `state_root` is on-chain. Transaction data is held by DIG fullnodes. If all fullnodes vanished, state remains auditable from L1. Transaction history would not. Standard rollup DA assumption. Mitigation: many independent fullnode operators.

6.8 Signature Domain Separation

Every BLS signature in DIG Network is computed over a domain-tagged payload, not over raw transaction fields. The signing rule is uniform across every signature surface — validator attestations, checkpoint aggregate signatures, voluntary-exit messages, governance votes, and (in Part 2) RetrievalSettlement co-signatures, NodeRegister attestations, SlashEvidence reports, and WithdrawAccrued authorizations:

```
SignedPayload = sha256(
   DOMAIN_TAG ||         // ASCII, transaction-type specific
   PROTOCOL_VERSION ||  // u32, bumped on payload format changes
   network_id ||         // 32 bytes, from NetworkConfig
   canonical_serialize(fields)
)
```

`DOMAIN_TAG` is a transaction-type-specific ASCII string from a fixed registry (e.g. "DIG_CHECKPOINT", "DIG_VOLUNTARY_EXIT", "DIG_DFSP_RETRIEVAL_SETTLEMENT", "DIG_DFSP_NODE_REGISTER", "DIG_DFSP_SLASH_EVIDENCE", "DIG_DFSP_WITHDRAW_ACCRUED"). `PROTOCOL_VERSION` is bumped when a payload's field schema changes; signatures from one schema version are never valid for another. `network_id` binds

the signature to a specific network (mainnet, testnet, shard).

This is the BLS analogue of Chia L1's `AGG_SIG_ME` condition, which binds a CLVM signature to a specific coin and network. DIG's BLS signatures cannot be bound to a coin (they sign protocol-level transactions, not coin spends), but they CAN be bound to a network, a transaction type, and a schema version. The result: a signature gathered on testnet cannot be replayed on mainnet, a signature for one transaction type cannot be repurposed for another, and a signature for a v1 payload format cannot be reused if the payload struct ever changes shape under v2.

The domain registry is part of `NetworkConfig` (§12). Adding a new transaction type adds a tag to the registry; signers and verifiers reject any payload whose tag is not in the registry. Verification cost is unchanged from raw-field signing: one extra sha256 hash before the pairing check.

This complements but does not replace transaction-level replay defenses (epoch binding in checkpoint digests, `registration_coin_id` in voluntary exits, freshness windows on settlements). Domain separation closes the signature attack surface; the higher-level mechanisms close the transaction attack surface. Both are required.

7. Consensus and Slashing

7.1 Block format

- **Header.** Parent hash, block number, epoch number, proposer pubkey, proposer signature, state root, transactions root, timestamp.
- **Transactions.** CLVM SpendBundles. Each validated by `dig-clvm` (delegates to `chia-consensus::run_spendbundle`) for CLVM correctness, condition validity, signature aggregation, and cost.
- **Attestations.** BLS partial signature over (`source_epoch`, `target_epoch`, `head_block_hash`) plus participation flags.

Block validation is tiered. Cheap tier (signature, header well-formedness, parent existence) runs before mempool admission. Expensive tier (CLVM execution, double-spend check against `dig-coinstore`) runs at finalization. Same gas-vs-bandwidth split as Chia's mempool.

7.2 Fork choice

LMD-GHOST modified for finality. Each validator's most recent attestation contributes weight to head selection. Canonical head is the most-attested descendant of the latest hard-finalized checkpoint.

Casper FFG (`source_epoch`, `target_epoch`, justified-then-finalized chain) is not required at Ethereum's density. DIG's hard finality is delivered by Chia L1 confirmation. Within an epoch, fork choice is LMD-GHOST. Across epochs, the L1-confirmed `checkpoint_singleton` lineage is authoritative.

7.3 Slashing offenses

`dig-slashing` enforces four correlation-penalty offenses:

Offense	Detection	Penalty
`ProposerEquivocation`	Two different blocks signed by the same proposer for the same slot	Correlation penalty
`InvalidBlock`	A proposed block fails L2 validation	Correlation penalty

`AttesterDoubleVote`	Two attestations from the same validator with the same target epoch but different roots	Correlation penalty
`AttesterSurroundVote`	Attestation A surrounds B (A.source < B.source < B.target < A.target)	Correlation penalty

Correlation penalty: the more validators in the same offense window, the steeper the penalty. Honest mistake by one validator costs little. Coordinated attack by many costs everything. Same anti-cartel posture as Ethereum.

Participation and inactivity penalties (§7.4) reduce balance but are not classified as slashing offenses; the four offenses above are the only correlation-penalty events.

Each offense is recorded in `dig-slashing` per-epoch offense state. At epoch close, every offender's exit-ledger entry in the next checkpoint shows a reduced `unslashed_amount`. The L1 effect appears at the offender's next `registration_coin` exit spend: the `withdraw_delay_coin`'s mojo value is the reduced amount.

7.4 Participation flags and inactivity leak

`dig-slashing::participation` records three Altair flags per attestation per epoch:

- **TIMELY_SOURCE.** Cited the correct source checkpoint.
- **TIMELY_TARGET.** Cited the correct target checkpoint within the inclusion window.
- **TIMELY_HEAD.** Cited the correct head block within the inclusion window.

Each missed flag earns a participation penalty.

`dig-slashing::inactivity` maintains a per-validator inactivity score. Constants from Ethereum Bellatrix verbatim: `INACTIVITY_SCORE_BIAS` = 4, `INACTIVITY_SCORE_RECOVERY_RATE` = 16, `INACTIVITY_PENALTY_QUOTIENT_BELLATRIX` = 16_777_216. The leak kicks in when the network fails to finalize for an extended window. It drains non-attesting balances and lets the honest set recover finality.

7.5 Rewards

Per-epoch DIG block subsidy splits across five reward roles. The first four go to consensus validators. The fifth goes to DFSP storage providers (Part 2).

- **Proposer reward** for producing the block.
- **Attester reward** for attesting on time (split across participation flags).
- **Inclusion-delay reward** for including others' attestations promptly.
- **Checkpoint signer reward** for the BLS aggregate that closes the epoch.
- **Storage provider reward** distributed to DFSP storage nodes proportional to their per-epoch retrieval proof share (Part 2 §11.3). Active only after `DFSP_ACTIVATION_HEIGHT`; before activation, the storage share is zero and the first four roles consume the full subsidy.

The five role shares are Category-A governance parameters in the `ParameterRegistry` (§4.6, §9). They sum to 100% of the per-epoch block subsidy. The storage-provider share is set high during DFSP bootstrap and declines on a fixed schedule to a steady-state share. The other four shares expand proportionally outside the bootstrap window.

Storage nodes earn DIG from two sources: their share of block subsidy (this section) and client retrieval payments (§8.2 utility use). Both flows are denominated in DIG.

Separate 50/50 fee split on DIG-denominated transaction fees: half to validators (weighted by participation), half burned. Structurally identical to EIP-1559.

Supply effect:

```
DIG_circulating(t+1) = DIG_circulating(t) + subsidy(t) - burn(t)
```

Subsidy follows a Chia-style halving curve with non-zero asymptote. Burn comes from two sources: transaction-fee burn (this section) and collateral burn for stored data (Part 2 §4-§7). At low usage, subsidy dominates and DIG is mildly inflationary. At high usage with significant data volume, burn dominates and DIG can be net-deflationary. Crossover is a function of validator-set size, L2 throughput, and stored data volume, not a parameter.

Protocol-level mint and burn under the coinset model. Both subsidy and burn are L2 protocol operations, not user-submitted spends. Subsidy: each block's application creates fresh DIG coins paying the proposer, attesters, inclusion-delay attestors, checkpoint signers, and (when DFSP is active) the storage-provider pool, in the proportions set by the five role shares; the coin creations are deterministic from the block contents and require no parent coin. Burn: fee inputs to the block consume DIG coins, but only half of the consumed value creates a child coin (paid to validators by participation weight); the other half is destroyed by the absence of a corresponding child. Every validator computes the same set of issuance creations and the same burn arithmetic during block validation; disagreement on subsidy or burn arithmetic is disagreement on the block, and the block fails consensus. The `state_root` commits the resulting coin set, so light clients verify final supply against the L1-anchored `state_root`.

7.6 Appeals

`dig-slashing` supports optimistic-fraud-proof appeals. Before the offender's `registration_coin` is finalized as exited, any party may submit counter-evidence that the offense did not occur. The L2 validator set adjudicates. If sustained, the next checkpoint's exit-ledger entry is rewritten with the original unslashed amount.

Once the `withdraw_delay_coin` exists on L1, the appeals window closes. `withdraw_delay_blocks` is the appeals window plus defense-in-depth budget. ~5 days is enough time to detect and intervene on a wrongful slashing.

Part 3: Economics and Governance

8. Tokens and Economics

Two assets. Clean separation.

8.1 XCH: the staking asset

XCH is Chia L1's native asset. In DIG, XCH plays one role: collateral.

- Validators lock XCH equal to `collateral_amount` into a `registration_coin` on Chia L1.
- XCH is removed from circulating supply for the validator's life.
- On exit, the unslashed XCH (attested by Groth16 + BLS) returns to the operator's `withdrawal_puzzle_hash` after the withdrawal delay.

Target collateral at launch: ~$5,000 USD-equivalent per validator. Creates a demand sink proportional to validator-set size. Governance adjusts `collateral_amount` as the network scales and XCH price moves. The dollar peg is design intent at genesis. Not enforced on-chain.

$5k is a professional commitment, not a hobbyist position. High enough to require financial intent. Low enough that the set can grow to thousands without becoming aristocratic. `MAX_SIGNERS` = 20,000 bounds the network at scale.

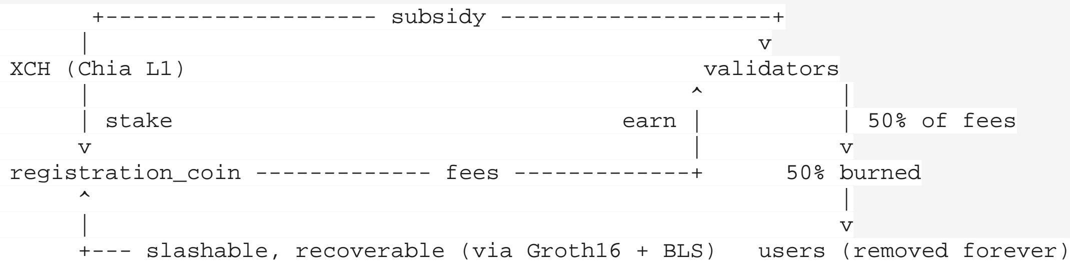
8.2 DIG: the reward and utility asset

DIG is the L2's consensus reward, gas asset, and payment asset for DFSP storage.

- **Issuance.** Block subsidy. Chia-style halving curve to a non-zero floor. Zero premine. Zero genesis allocation. Every DIG is earned by validating.
- **Denomination.** Smallest unit is the dojo. 1 DIG = 1,000,000,000 dojos (10^9). (dojo : DIG :: lamport : SOL.)
- **Consensus use.** Validator block subsidy (paid every block to proposer and attesters). L2 transaction fees (50% to validators, 50% burned).
- **Utility use (Part 2).** Payment asset for storing data on the DFSP protocol. Clients pay DIG to storage providers for capacity, retrieval, and namespace operations. This is the demand side that balances validator issuance.
- **Storage stake asset (Part 2).** DFSP storage nodes stake DIG (not XCH) into a per-node singleton. Storage stake and consensus stake are independent: an operator running both roles satisfies both stake requirements separately.
- **Not a governance token.** DIG voting weight is zero. Governance weight is validator-set membership.
- **Not pre-allocated.** No team allocation, no foundation reserve, no grant pool, no airdrop. Genesis state has zero DIG. First DIG is minted by the first block.

DIG is the L2-native asset of DIG Network. Part 3 (Digstore) references a pre-mainnet placeholder DIG CAT (Chia Asset Token, asset ID `a406d3a9de984d03c9591c10d917593b434d5263cabe2b42f6b367df16832f81`) used by standalone digstore archives during the pre-DIG-Network period. The placeholder CAT is not consensus state and is not the steady-state DIG token; once DIG Network mainnets and a digstore archive registers as a DFSP store, collateral migrates to L2-native DIG under the DFSP `CollateralRecord` primitive (Part 2 §4).

Closed loop:



```
        +-------------------- subsidy --------------------+
        |                                                 v
  XCH (Chia L1)                                       validators
        |                                            ^          |
        | stake                                 earn |          | 50% of fees
        v                                            |          v
  registration_coin ------------- fees -------------+      50% burned
        ^                                                       |
        |                                                       v
        +--- slashable, recoverable (via Groth16 + BLS)   users (removed forever)
```

8.3 Effect on Chia

- **XCH demand.** Every validator locks XCH out of circulation. 1,000 validators × ~$5k = ~$5M XCH removed. 10,000 validators = ~$50M. Governance can raise `collateral_amount` over time.
- **No competition with XCH.** XCH and DIG do not contest each other's economic role. XCH is the stake and store-of-value asset on Chia L1. DIG is the consensus reward and the payment asset for DFSP storage on DIG L2. A storage buyer pays DIG. A validator earns DIG by securing the L2. A staker locks XCH. The three roles are separate.
- **No team rent.** No premine, allocation, or treasury. DIG cannot be the source of a token unlock or insider distribution. Emission curve is public. Economics are determined by validator participation and throughput.
- **Aligned governance.** The validator set is the governance set. Governance is exercised by signing parameter-update proposals with 2k > n BLS majority, verified by the same Groth16 circuit as the checkpoint. Voting weight is participation, not capital.

9. Governance

Governance is narrow. It covers parameters the protocol promises are mutable. It does not cover arbitrary code upgrades. Those happen out-of-protocol via client releases.

9.1 Three categories

Category	Who decides	Where it lives	Cost
A. Protocol parameters (reward quotients, churn, fee burn ratio, slashing thresholds, inactivity constants, extension parameters from protocols built on DIG L2)	Validator set (2k > n)	Curried into the `ParameterRegistry` singleton (§4.6). `collateral_amount` lives in `network_coin` (§4.1) and updates through its own parameter-update path.	One L1 SpendBundle with Groth16 + BLS
B. Puzzle upgrades / hard-fork changes (new CLVM rules, opcodes, validator-record fields)	Operators + validators by client release	Off-chain client version with on-chain trigger	New client deployment, coordinated activation epoch
C. Trust-anchor changes (new Groth16 verification key after new MPC ceremony)	Validator set + operators	New `verification_key_hex` curried into a new network	Effectively a new network

Category A is the on-chain governance surface. B and C are out-of-protocol. Client upgrades stay separate from validator voting. Consensus stays separate from protocol design.

9.2 The governance circuit

Category-A changes are SpendBundles that consume the `ParameterRegistry` (or, for `collateral_amount`, the `network_coin`) and provide:

- A 32-byte `governance_message` digest committing to the parameter change.
- A Groth16 proof against the current validator Merkle root that k signers sign, 2k > n, aggregate pubkey matches.
- A BLS aggregate signature over the digest.

Same shape as the checkpoint circuit. Same `MAX_SIGNERS`, `TREE_DEPTH`, same membership/majority/aggregation gadgets. Public inputs differ. Same trusted-setup output reused. Deploying governance adds no new ceremony.

DIG governance is computationally identical to DIG finality. A parameter change has the same cryptographic strength, same threshold, same on-chain cost as a checkpoint update.

9.3 What governance cannot do

- **No arbitrary code execution.** No on-chain action can run new CLVM, change puzzle logic, or alter the wire format.
- **No validator de-listing.** A validator cannot be removed by vote. Only voluntary exit and slashing remove validators.
- **No treasury.** The 50% fee burn destroys dojos. It is not a treasury.
- **No token issuance.** Governance cannot mint DIG. Emission curve is fixed by client code.

Part 4: Implementation

10. Networking and Binaries

10.1 Binaries

DIG ships five:

- **dig-fullnode.** Canonical L2 chain-state node. Validates blocks, maintains mempool, runs the L1 anchoring loop (walks `checkpoint_singleton` lineage and `registration_coin` set), exposes operator and public JSON-RPC. Never signs. Most peers are fullnodes.
- **dig-validator.** Signs blocks, attestations, checkpoints, and voluntary exits. Holds the BLS signing key in an encrypted file. Does not hold the L1 withdrawal key; the `registration_coin`'s `withdrawal_puzzle_hash` is set at registration and the withdrawal key is controlled independently (see §13). Connects only to its paired fullnode. One binary per BLS pubkey.
- **dig-daemon.** Optional orchestrator. Starts and stops services, unlocks keychain, streams logs. Mirrors `chia_daemon`.
- **dig-introducer.** Bootstrap peer-discovery node. Holds known-good peer addresses.
- **dig-relay.** NAT-traversal helper. Forwards encrypted peer connections across networks without direct inbound.

All five built from one Rust workspace, sharing consensus, gossip, storage, and protocol crates. Key management for all five (and for end-user wallets used by Part 2 DFSP and Part 3 Digstore layers) is handled by `dig-wallet`: a multi-profile mnemonic-backed Chia-style wallet. Different profiles serve different roles (validator L1 withdrawal key, end-user DIG holder, digstore publisher, storage-node operator); the wallet implementation is shared across roles.

10.2 Wire protocols

DIG does not define a new wire protocol. It runs Chia's four protocols as-is: the **networking protocol**, the **peer (full-node) protocol**, the **wallet (light-client) protocol**, and the **serialization protocol**. The message envelope, the Streamable encoding, the handshake, the certificate scheme, and peer discovery are byte-for-byte Chia. The one and only difference is the network itself.

Public interface, not implementation. What DIG conserves is the on-the-wire contract every Chia application already targets: the message types and their field layouts, the Streamable encoding, the handshake fields, and the JSON-RPC request and response shapes. The node behind that contract is DIG's own independent Rust implementation; it does not copy Chia's internals and does not need to. Conforming to the byte-level interface is the requirement, not matching code. Where a Chia behavior is already backwards-compatible, DIG keeps it rather than reinventing it.

The network is the only divergence. Chia's handshake carries a `network_id` field (Chia uses values such as `mainnet`, `testnet11`, `simulator0`). DIG advertises its own `network_id` from `NetworkConfig` (§4.7, §12). Two peers that complete the handshake but advertise different `network_id` values do not peer, so DIG and Chia nodes never cross-connect even though they speak an identical protocol. The constants that travel with a network are DIG's own: the `network_id`, the genesis challenge, the default port, the signing CA, and the DNS-introducer list. This is exactly the mechanism Chia uses to keep its own mainnet, testnets, and simulator apart. DIG is one more network under that same mechanism, not a protocol fork.

Serialization (verbatim). Every DIG object on the wire or under a hash uses Chia's Streamable format unchanged: sized ints and sized bytes in big-endian, `bool` as one byte, `bytes` and `str` as a 4-byte length prefix followed by the

data, `List` as a 4-byte count prefix, `Optional` as a one-byte `0x00`/`0x01` prefix, BLS public keys and signatures as 48 and 96 bytes, and CLVM programs through their own `.parse`. DIG's blocks, attestations, and the checkpoint payload (§6.5) are all Streamable structures, so DIG and Chia tooling serialize and hash them identically.

Networking (verbatim). DIG uses Chia's message envelope (a one-byte message type, an optional two-byte request id, and a Streamable `data` payload) over mTLS WebSockets. Connections are X.509-authenticated and a peer's node id is the SHA-256 of its DER certificate, exactly as in Chia. Heartbeats, the ban rules, the outbound-peer target, the roughly 4 GB message ceiling, and peer gossiping via `request_peers`/`respond_peers` are unchanged. DIG ships its own CA and its own DNS introducers because those belong to the network, not the protocol.

Peer and wallet (verbatim where they have an analogue). Full-node sync (`new_peak`, then `request_block`, then `respond_block`, in batched and long-sync forms), transaction relay (`new_transaction`/`request_transaction`/`respond_transaction` carrying a `SpendBundle`), and the light-client coin-state messages (`send_transaction`, `request_additions`/`request_removals`, `request_puzzle_solution`, `new_peak_wallet`) are reused unchanged. The only Chia messages DIG does not carry are the proof-of-space-and-time ones (signage points, end-of-sub-slot bundles, VDFs, and weight proofs), because DIG is proof-of-stake and has no timelord. In their place, a peer or wallet establishes a peak's validity from the L1-confirmed `checkpoint_singleton` (§6.1, §6.6) rather than from a weight proof. That substitution is a consequence of DIG's consensus, carried over an otherwise-identical protocol; it adds no new framing or encoding.

Node roles. DIG's roles are advertised through the handshake's existing `node_type` field. No field, framing, or encoding is added, and DIG's consensus messages ride in the same one-byte-typed envelope as every other Chia message. Part 2 (DFSP) introduces its storage roles and messages the same way: additional values in the existing fields, carried over the unchanged protocol. Because DIG runs as a separate network, none of these values ever reach a Chia node.

NodeType	Value	Role
FullNode	1	Chia or DIG fullnode
Introducer	5	Chia or DIG introducer
Wallet	6	Chia or DIG wallet
Validator	10	DIG validator
Relay	11	DIG relay
Explorer	12	DIG explorer

Three priority lanes (Critical, Normal, Bulk) enforce per-message rate limits, with shape and quotas inherited from Chia's `rate_limit_numbers.py` and tuned for DIG message volumes.

Additive, backwards-compatible message types. DIG's consensus messages and Part 2's DFSP messages occupy a high, reserved block of message-type values (the envelope's one-byte type, drawn from `ProtocolMessageTypes`) that Chia leaves unused: DIG consensus in 200-219, DFSP in 220-239. Chia's existing values (0-107) are untouched. Because the one-byte type field governs decoding, a parser that receives a type it does not recognize simply ignores the message, exactly as it ignores any unknown type today. The expansion is therefore strictly additive: it extends the interface without breaking it. DIG keeps Chia's existing message handlers unchanged, so a Chia peer's messages are understood as-is and only the new types add handlers. New NodeTypes (above) and new handshake `capabilities` entries are advertised the same additive way, so peers that lack a capability negotiate

down cleanly. An older client never has to understand DIG's additions to use the base interactions.

RPC (same interface). DIG exposes the same local control surface as Chia: a JSON RPC served over both HTTPS and WebSocket, with the same request and response shapes for the calls applications depend on (chain tips, blocks, coin and puzzle-hash lookups, pushing transactions, connection management). Dashboards, indexers, and operator tooling written against Chia's RPC work against a DIG node after repointing the endpoint.

10.3 Propagation and peer discovery

Propagation follows Chia's announce-then-request pattern rather than a separate gossip overlay. A node that advances its peak or admits a transaction sends a short notification (`new_peak`, `new_transaction`); peers request the object only if they want it (`request_block`, `request_transaction`) and ignore the notification otherwise. DIG's block and attestation messages travel the same way, as ordinary Chia protocol messages over the unchanged envelope. Peer discovery is Chia's: the `request_peers`/`respond_peers` exchange (no more than 1000 peers per response), an introducer answering `get_peers_introducer`, and DNS introducers, with the same new and tried peer tables that guard against eclipse attacks.

- Validators connect only to their paired fullnode, never to public gossip endpoints. This minimizes the attack surface against BLS signing keys.
- Fullnodes connect to other fullnodes, introducers, and relays. They are the public face of the network.

Same posture as Chia: signing keys live behind the validating peer, and the validating peer never holds signing keys.

10.4 Bootstrap

1. Resolve introducer addresses, by DNS-introducer lookup or the configured introducer for the network.
2. Open an mTLS WebSocket and complete the Chia handshake, advertising DIG's `network_id`.
3. Obtain an initial peer set via `get_peers_introducer`, then exchange `request_peers`/`respond_peers` with those peers, populating the new and tried tables.
4. Begin sync against the `checkpoint_singleton` lineage on Chia L1, the L1 anchor for DIG state.

The introducer is not part of consensus. It holds peer-list state only. An introducer outage degrades discovery but cannot affect finality or block production.

10.5 Migration for existing Chia applications

The compatibility goal is concrete: a wallet, explorer, indexer, light client, or SDK already written for Chia should reach DIG's base interactions with as little change as possible. Because the wire and RPC interfaces are conserved (§10.2), the protocol layer of such an application does not need to be rewritten. What changes is the network it points at:

- **Set the DIG network_id.** Advertise DIG's `network_id` in the handshake instead of a Chia one. This is the single change that routes the client to DIG rather than Chia.
- **Point at DIG endpoints.** Use DIG's DNS introducer and signing CA in place of Chia's. These are network constants, not protocol changes.
- **Carry over the network constants.** DIG ships its own address prefix and genesis challenge in `NetworkConfig` (§4.7, §12) — the same per-network values a Chia client already parameterizes.

Everything the application actually speaks is unchanged: the Streamable types, the light-client message set (`send_transaction`, the puzzle-hash and coin-id subscriptions, `request_additions`/`request_removals`, `coin_state_update`), the full-node transaction and sync messages, and the JSON-RPC calls. A transaction is a

CLVM `SpendBundle` on DIG just as on Chia (§7.1), so signing, coin selection, and puzzle code carry over directly. In practice an existing Chia wallet talks to a DIG node after a configuration change, not a code change; new code is needed only to use DIG-specific features (the staking and storage messages in the additive range), never for the base interactions.

11. Storage

Fullnode state lives in four stores, each a Rust crate.

- **dig-blockstore.** RocksDB block storage. Six column families (blocks, headers, canonical chain, metadata, checkpoints, attested). Dual-layer canonical-chain indexing (memory-mapped + RocksDB), sharded LRU caches, zstd compression with trained dictionaries after the first 1,000 blocks, async write pipeline, atomic reorg support.
- **dig-coinstore.** RocksDB coin and hint storage. Derived from Chia's `CoinStore` and `HintStore`. Twelve column families. Depth-32 sparse Merkle tree commits global state and produces `state_root` for each checkpoint. Snapshot-and-restore. Rollback and reorg support.
- **dig-mempool.** In-memory transaction pool with admission controls, eviction, and priority-fee ordering.
- **dig-blockstore checkpoint history.** Persistent record of every L1-confirmed checkpoint, keyed by epoch. Enables historical Merkle proofs against any past `state_root`.

Same reorg discipline across all four. Every state-mutating operation is reversible until L1 confirms the checkpoint. At confirmation, the epoch's writes become permanent.

12. The chia-l2-consensus Anchor Crate

Every L1-touching operation routes through `chia-l2-consensus`. It is the only seam between DIG L2 and Chia L1.

The crate owns:

- Rue sources for the five puzzles (`network_coin_inner.rue`, `registration_coin.rue`, `checkpoint_inner.rue`, `withdraw_delay_coin.rue`, `parameter_registry_inner.rue`).
- Rust builders that construct SpendBundles for register, checkpoint, exit, and release.
- Read-only `ConsensusClient::sync` that walks the `network_coin`, `checkpoint_singleton`, and `ParameterRegistry` lineages on Chia L1 and emits the current validator set, checkpoint state, and parameter snapshot.
- `NetworkConfig` constants: launcher IDs, puzzle module hashes, verification key hex, trusted-setup parameters, `MAX_SIGNERS`, `TREE_DEPTH`, `withdraw_delay_blocks`, the signature domain-tag registry (§6.8), and `PROTOCOL_VERSION`.

The crate builds SpendBundles but never broadcasts them. The importing project (`dig-fullnode`, `dig-validator`, or external tooling) calls `chia_query::push_tx` to submit. The L1-touching surface is a small set of pure functions producing serializable bundles. Trivially auditable.

`NetworkConfig` is immutable after deployment. Saved as JSON in deployment artifacts. Loaded on every startup. Changing it produces a new deployment, not an upgrade. Combined with the curried `verification_key_hex`, every DIG deployment is cryptographically distinct from every other. Mainnet, testnet, and future shards are unforgeable against each other.

13. Security Properties

Properties that hold, with the mechanism that enforces each.

- **Validators cannot be slashed more than their stake.** Exit puzzle bounds payout at `original_collateral`.
- **Groth16 proofs cannot be forged.** Trusted-setup soundness plus circuit constraints.
- **The BLS aggregate signature binds the proof to a specific checkpoint message.** Without it, the proof would only prove "some majority exists," not "they signed this state."
- **No cross-network replay.** Checkpoint digest includes the network ID.
- **No epoch replay.** Membership and exit announcements include the epoch number.
- **No condition injection.** `registration_coin`, `checkpoint_inner`, `withdraw_delay_coin`, `network_coin`, and `parameter_registry_inner` accept no solution-provided conditions. SEC-008 invariant in `chia-l2-consensus`.
- **No mid-flight validator removal.** Only voluntary exit and slashing remove validators.
- **No on-chain code upgrade.** Puzzle upgrades require a new deployment.
- **Light-client coin proofs are 1 KiB and stateless.** Verify a past coin's existence with a Merkle proof against the L1-confirmed `state_root`. No DIG node trust required.
- **Validator BLS keys are not co-located with L1 wallet keys.** The validator binary holds the BLS signing key. The L1 collateral is a `registration_coin` whose `withdrawal_puzzle_hash` is independently controlled. Compromise of one does not compromise the other.
- **Signatures cannot be replayed across networks, transaction types, or schema versions.** Domain-tagged signing (§6.8) binds every BLS signature in the protocol to (`domain_tag`, `protocol_version`, `network_id`, fields). A signature on testnet cannot be valid on mainnet; a NodeRegister signature cannot be repurposed as a RetrievalSettlement signature; a v1 signature cannot be reused under v2.

Remaining trust assumptions:

- **Groth16 trusted-setup MPC is honest** (at least one honest participant). Standard.
- **A majority of validators are honest within the inactivity-leak window.** Standard BFT assumption.
- **Chia L1 is honest and live.** Standard for any L2.

No trusted operator, foundation key, escape hatch, upgrade authority, or recovery oracle. Protocol authority is the validator set, expressed on Chia L1.

Part 5: Status

14. Status and Roadmap

DIG is in late prototype, pre-public-testnet.

- **Specs complete.** L1 puzzles, validator lifecycle, consensus, slashing, networking, storage, governance are specified in `docs/resources/specs/` and traced to the Rust crates that implement them.
- **Implementation in progress.** ~27 crates: consensus (`chia-l2-consensus`), block format (`dig-block`), state (`dig-coinstore`, `dig-blockstore`), epoch (`dig-epoch`), slashing (`dig-slashing`), mempool (`dig-mempool`), networking (`dig-gossip`, `dig-protocol`), keys and signing (`dig-keystore`, `chia-bls`), operator surface (`dig-rpc`, `dig-rpc-types`, `dig-service`), and the five binaries.
- **V1 local testnet has run end-to-end.** Full register → propose → attest → checkpoint → exit → withdraw loop, locally.
- **V2 local testnet in preparation.** Improvements identified during V1 are being integrated.
- **Trusted-setup MPC ceremony scheduled before mainnet.** The Groth16 verification key from the ceremony will be curried into `checkpoint_singleton`, `network_coin`, and `ParameterRegistry` for the mainnet deployment. The same ceremony also produces verification keys for the two additional Groth16 circuits introduced in Part 2 (DFSP's retrieval-proof and epoch-transition circuits), so DFSP adds no new trust anchor.
- **Public testnet follows V2 local validation.** Mainnet follows public testnet and the MPC ceremony.

No mainnet launch date. Schedule is gated on V2 testnet stability, MPC ceremony completion, and pre-launch security review of the five CLVM puzzles and the Groth16 circuit.

15. What Comes Next: Part 2 (DFSP)

This paper covers DIG L2 as a consensus substrate. The application is DFSP.

DFSP is a decentralized file-storage protocol implemented as a first-class capability of DIG L2 consensus. It plugs into Part 1 as a strict extension: four new CLVM puzzles in a new `dig-storage-puzzles` crate, none of which touch L1; four new indexed L2 state tables with Merkle roots committed in every block header; one new block-validation method (`DfspBlockApply`); new block-body categories for DFSP deltas and submissions; and reuse of Part 1's BLS aggregation, validator set, and Groth16 circuit verbatim. None of Part 1's five puzzles change.

DFSP introduces four indexed L2 state tables:

- **A collateral registry indexed by storeID.** Holds one `CollateralRecord` per active store. The slashable retrievability bond lives in a per-store singleton coin (`collateral_bond_coin`). The depletable retrieval budget lives as a protocol-managed account balance in the table itself, not as a coin or token.
- **A store registry indexed by storeID.** Commits each store's lifecycle state (UNCOMMITTED, COMMITTED, LIVE, GRACE, EXPIRED) and the current roothash from its `store_lineage_coin` singleton.
- **A node registry indexed by NodeID.** Tracks storage-node stakes (denominated in DIG, not XCH), declared capacity, and status. Per-node stake is held in a `node_stake_coin` singleton; the registry mirrors balance and status for fast lookup.
- **A retrieval proofs table indexed by (NodeID, epoch).** Records each node's per-epoch retrieval activity and the Groth16 retrieval-proof-and-possession submission that attests it.

Every L2 block carries DFSP state transitions as first-class body categories (registry deltas, retrieval settlements, ingestion receipts, retrieval-proof submissions, slash evidence). A new `DfspBlockApply` method runs on every block, applies the deltas to the four indexed tables, and verifies the block header's four DFSP Merkle roots match the updated tables. Validators that disagree on the resulting roots reject the block. Determinism is by construction: identical inputs, identical roots.

The four DFSP roots ride alongside `state_root` in every BLS attestation and fold into the extended checkpoint digest (§6.5) at epoch close. DFSP state is hard-finalized to Chia L1 alongside consensus state via the same Groth16 + BLS attestation.

Assignment of fragments to storage nodes is also a block-header field. Every block carries an `assignment_seed` = Hash(`current_epoch` || `node_registry_root` || `parameter_snapshot`). Any party computes assignments deterministically against the same canonical CLVM library. No routing oracle singleton, no DHT lookup, no gossip-based assignment, no eclipse-attackable peer state.

DFSP introduces two additional Groth16 circuits: a retrieval-proof-and-possession circuit (per-epoch proof of fragments served, with continuous-possession attestation) and an epoch-transition circuit (validator-set attestation over `node_registry_root` advancement at each epoch boundary). Both reuse Part 1's membership-and-majority gadget verbatim. A unified MPC ceremony before `DFSP_ACTIVATION_HEIGHT` produces all three verification keys (consensus, retrieval-proof, epoch-transition) from one setup. DFSP adds no new trust anchor.

Storage nodes are a separate role from consensus validators. Storage stake is denominated in DIG; consensus stake is XCH. Slashing on one role has no effect on the other. An operator may run both, neither, or either.

Part 2 covers DFSP in full: collateral as the storage primitive, the dual-pool (bond singleton + protocol-managed budget balance) economic model, block-derived assignment, first-class L2 RetrievalSettlement transactions, Groth16-compressed per-epoch retrieval-proof submissions, the four new puzzles, the extended L2 block format, the three client SDKs (`dig-write`, `dig-read`, `dig-name`), and the bootstrap pathway.

Part 3 (Digstore) specifies the default payload format that DFSP serves as opaque encrypted archive fragments. A digstore archive can operate standalone (Chia L1 DataLayer singleton as L1 identity anchor) or be registered with DFSP, at which point a `store_lineage_coin` is launched on L2 whose launcher ID becomes the canonical DFSP storeID. Digstore and DFSP coexist by design: Part 3 is the artifact, Part 2 is the distributed service that moves it.

DIG is the Chia-anchored L2 on which a fair-launch, no-premine decentralized storage protocol runs natively in consensus, with the same operational and economic posture as Chia itself.

16. Acknowledgements and References

DIG borrows heavily and with attribution. Upstream sources:

- Chia's full-node, mempool, server, and CLVM implementations (`chia/blockchain/`, `chia/mempool/`, `chia/server/`, `chia/wallet/puzzles/`).
- Chia's wire protocols, used verbatim: the networking, peer, wallet, and serialization protocols (`chia/protocols/`, `chia/server/`). DIG changes none of them; only the `network_id` and its accompanying network constants differ.
- Ethereum's consensus specs (`ethereum/consensus-specs/specs/phase0`, altair, bellatrix). Specifically the Validator container, participation flags, and inactivity-leak formula.
- Polygon's checkpoint module (`0xPolygon/heimdall-v2/x/checkpoint`). Specifically the L1-anchored periodic state-root commitment.

- The Groth16 SNARK (Jens Groth, 2016) as implemented over BLS12-381 in arkworks and audited variants.
- Part 3 (Digstore). The default content-addressable archive format DFSP serves. Single-file archive with URN-derived encryption such that DIG nodes hold and serve fragments without ability to inspect content. Standalone archives anchor identity to Chia L1 via a DataLayer singleton; DFSP-registered archives anchor to a `store_lineage_coin` on L2 instead.
- Companion DIG Network Part 2 (DFSP). Decentralized file-storage protocol implemented as a first-class capability of DIG L2 consensus.

Synthesis: Chia primitives plus Ethereum staking shape plus Polygon rollup pattern plus constant-cost SNARK-verified L1 anchoring plus fair-launch L2-native reward asset with EIP-1559 burn.

Pre-publication draft. Comments and corrections from Chia core developers on the L1 CLVM puzzle surface and the Groth16 verification cost analysis are invited.