

# SoFi: Sovereign Deterministic Finance with Vault-Based Liquidity Anchors

Brandon “Cryptskii” Ramsay\*

## Abstract

SoFi (Sovereign Finance) redefines digital finance by replacing consensus-based infrastructure with deterministic, hash-anchored commitments. This paper formalizes a blueprint for fully sovereign, non-custodial vault-based liquidity systems using Deterministic Limbo Vaults (DLVs) anchored to user identities. Unlike centralized smart contract pools, liquidity in SoFi remains under the originator’s control, discoverable via public storage nodes, and executable by cryptographic proof alone—no trusted intermediaries required. Liquidity aggregation occurs through coordinated vault advertisement, composable Smart Commitments with external hash commitments, deterministic encumbrance accounting (preventing collateral double-pledge), and clockless liveness guarantees via pre-authorized fallback transitions. Critically, routing, verification, discovery-quorumming, deterministic re-routing, and user-intent enforcement are implemented inside the SoFi SDK (STK), not delegated to trusted third parties; external route computation is optional and non-authoritative. Storage nodes serve purely as indexing and data availability infrastructure with no signing authority or custodial role. This document includes explicit storage node specifications intended to be used directly as an implementation blueprint.

## 1 Introduction

Traditional DeFi architecture requires users to deposit assets into smart contract pools, creating three fundamental risks: custody centralization, smart contract vulnerabilities, and MEV extraction by validators. SoFi eliminates these risks through a novel architecture where liquidity providers maintain sovereign control over their assets while still enabling efficient price discovery and trade execution.

The key innovation is the Deterministic Limbo Vault (DLV): a cryptographic construction where funds unlock deterministically when specific hash-based conditions are satisfied. No trusted party holds keys, no multi-signature coordination is required, and no consensus mechanism validates trades. Instead, mathematical verification of hash commitments provides absolute certainty about trade settlement.

### 1.1 Design Goal: Make Wrapper-Projects Unnecessary

Many chain-based designs attempt to patch systemic problems using application-layer financial policy: pooled custody risk, validator rent extraction, and oracle governance disputes. SoFi makes these patches unnecessary by enforcing the relevant economic invariants directly in vault logic:

- **Principal safety:** mutually exclusive fulfill/refund branches on the same parent state (Section 5.3).

---

\*Inventor of DSM (Decentralized State Machine).

- **Zero-fee execution:** no global mempool or validator ordering market; infrastructure paid via subscription for data availability (Sections 7 and 8).
- **Cross-margin without custody:** deterministic encumbrance commitments that prevent double-pledging and define priority (Section 3.3).
- **Oracle-free settlement:** endogenous settlement signals derived from protocol-owned executed tape with anti-distortion rules and explicit cost (Section 4).
- **Route failure resilience:** deterministic RouteSets with bounded user intent (minOut/maxFee) enabling rapid fallback without renegotiation (Section 3.5).

If these invariants are enforced by construction, wrapper structures become redundant (Section 13 and the direct comparison in Section 12).

## 1.2 Why Routing Is Built Into the SDK (STK)

A common failure mode in decentralized markets is quietly reintroducing trust by outsourcing “route selection,” “best execution,” or “verification” to a privileged service. SoFi avoids this by treating routing and verification as part of the local deterministic client responsibility:

- The SDK (STK) performs **non-authoritative** discovery, routing, proof assembly, deterministic re-routing, and verification using public data.
- Any external router is **optional compute assistance only**; it cannot force an unlock because vault predicates and deterministic verification reject anything not satisfying the committed TradeIntent.
- This is robust at scale: decentralization is preserved not by trusting routers to behave, but by making routers unable to matter.

This SDK-first posture is a protocol property: it prevents ecosystem collapse into a handful of routing gatekeepers while still enabling open competition on compute quality.

# 2 Vault-Based Sovereign Liquidity

## 2.1 Decentralized Vault Commitments

Users commit liquidity into DLVs, cryptographic constructions anchored to DSM bilateral relationships:

$$\text{Vault}_i = \text{DLV}(R_{\text{pre}}, \text{CTPA}, \text{UnlockLogic}, \text{Funds})$$

Each vault is:

- Anchored to the user’s identity via DSM genesis commitment
- Governed by deterministic unlock conditions (no Turing-complete execution)
- Discoverable via DSM storage node indexing
- Executable purely via hash verification—no signatures from storage nodes
- Maintained within the vault owner’s bilateral relationship chains

**Critical clarification:** Storage nodes do **NOT** have signing authority over vaults. They serve only:

1. Index vault metadata for discovery
2. Store and serve vault state data
3. Provide data availability for stitched receipts

All vault unlocking is deterministic verification performed by the trading parties themselves.

## 2.2 Storage Node Indexing and Discovery

Vault metadata is published to DSM storage nodes for discovery:

$$\text{IndexKey} = H(\text{TOKEN}_A, \text{TOKEN}_B, \text{CTPA}, \text{VaultID})$$

Storage nodes maintain searchable indices allowing discovery without pooled custody. The indexing layer is informational—nodes never control or sign transactions involving vaults.

Deterministic discovery flow:

1. The SDK queries a **quorum** of storage nodes: “Find vaults for TOKEN A/TOKEN B”.
2. Nodes return vault advertisements plus proofs (inclusion, freshness anchors).
3. The SDK deduplicates, validates proofs, and filters satisfiable vault candidates.
4. The SDK computes a bounded RouteSet satisfying the committed TradeIntent (Section 3.5).
5. The SDK assembles stitched receipts and executes bilaterally; storage nodes only relay and store.

## 2.3 Deterministic Unlock Mechanism

The vault unlock key is derived deterministically:

$$sk_V = H(L \parallel C \parallel \sigma)$$

Where:

- $L$  is the lock configuration (initial state commitment)
- $C$  is the condition set (pricing invariants, bounds, encumbrance rules, RouteSet membership, etc.)
- $\sigma$  is the stitched proof-of-completion showing conditions are satisfied

Key property: prior to  $\sigma$  existing, computing  $sk_V$  is infeasible (requires hash preimage). Once a valid  $\sigma$  exists (constructed by anyone satisfying the predicates), settlement executes deterministically.

No party “approves” the trade—the predicates either verify or they do not.

### 3 Smart Commitments and Coordination

#### 3.1 Structure of Smart Commitments

Smart Commitments are deterministic state transition predicates:

$$C = \{\Delta_{\text{in}}, \Delta_{\text{out}}, \text{invariants, external commitments, encumbrances, intent bounds}\}$$

Unlike VM-based smart contracts, Smart Commitments are:

- **Bounded in evaluation:** explicit finite evaluation budget, no unbounded recursion/loops
- **Purely functional:** predicates over committed inputs
- **Composable:** coordinated via external commitments rather than synchronous multisig
- **Deterministic:** same inputs produce the same verification result

#### 3.2 External Commitments for Multi-Vault Coordination

External commitments enable atomic coordination without coordinator signatures:

$$\text{ExtCommit}(X) = H(\text{"DSM/ext"} \parallel X)$$

Multiple vaults can reference the same  $X$ :

$$\begin{aligned} \text{Vault}_A &: \text{unlock if } H(\text{hop}_A) = h_A \wedge \text{ExtCommit}(X) \text{ exists} \\ \text{Vault}_B &: \text{unlock if } H(\text{hop}_B) = h_B \wedge \text{ExtCommit}(X) \text{ exists} \\ \text{Vault}_C &: \text{unlock if } H(\text{hop}_C) = h_C \wedge \text{ExtCommit}(X) \text{ exists} \end{aligned}$$

Where  $X$  binds the committed TradeIntent, RouteSet, and the execution result:

$$X = H(\text{route-set proof} \parallel \text{vault states} \parallel \text{final balances} \parallel \text{nonce})$$

Atomicity: either all required vaults observe a valid external commitment and unlock, or none do.

#### 3.3 Deterministic Encumbrance Commitments (Cross-Margin Without Custody)

Cross-margin is only safe if collateral cannot be double-pledged. SoFi enforces this using deterministic encumbrance accounting committed in vault state.

Let a vault maintain an encumbrance set  $\mathcal{E}$  describing all live claims on its reserves. The vault commits:

$$E = H(\text{"DSM/enc"} \parallel \text{VaultID} \parallel \text{Canonicalize}(\mathcal{E}))$$

Rules:

1. **Uniqueness:** claim identifier  $e_j \in \mathcal{E}$  can be consumed at most once, proven by inclusion in parent and removal in successor.
2. **Priority:** each  $e_j$  encodes deterministic priority (e.g., lexicographic key (VaultID, parent tip,  $e_j$ )). Conflicts cannot both validate because only one successor per parent tip is accepted (Tripwire; Section 7.2).

3. **Non-equivocation:** presenting two encumbrance sets for the same parent is a fork; honest verifiers reject the losing branch.

External commitments bind to encumbrance commitments to prevent collateral reuse across simultaneous routes:

$$X = H(\text{“DSM/route-set”} \parallel \dots \parallel E_A \parallel E_B \parallel \dots)$$

### 3.4 Routing and Path Computation

Routing is deterministic graph search over public vault metadata and proofs. The core rule: routing is **non-authoritative**. Authority is the committed TradeIntent and vault predicates.

The SDK:

1. Queries storage-node quorums (Section 2.2).
2. Verifies proofs, filters candidates.
3. Computes a bounded set of admissible routes (RouteSet; Section 3.5).
4. Assembles proof skeletons and stitched receipts for chosen route.
5. On hop unavailability, retries another route within the precommitted RouteSet without renegotiation.

External routers are optional compute accelerators and cannot override intent bounds.

### 3.5 Deterministic Route-Sets and Intent Bounds (Fast Re-route)

Distributed markets must handle temporary unavailability (offline endpoints, delayed delivery, stale index views) without sacrificing safety or user intent. SoFi solves this by committing to a **RouteSet** rather than a single route.

**TradeIntent.** The trader commits:

$$\text{TradeIntent} = \{\text{tokenIn}, \text{amountIn}, \text{tokenOut}, \text{minOut}, \text{maxFee}, \text{maxHops}, k\}$$

where:

- minOut is the user preset allowance for price difference (slippage) with no external oracle required.
- maxFee bounds total fees across all hops.
- maxHops and  $k$  bound complexity to prevent proof bloat and route spam.

**RouteSet.** Let  $\mathcal{R} = \{r_1, \dots, r_k\}$  be canonicalized admissible routes, where each route specifies hops and required encumbrance claims:

$$r_i = \langle (\text{VaultID}_1, \Delta_1, e_1), (\text{VaultID}_2, \Delta_2, e_2), \dots \rangle$$

**Committed Route-Set External Commitment.** Bind intent and route-set:

$$X = H(\text{“DSM/route-set”} \parallel \text{TradeIntent} \parallel \text{Canonicalize}(\mathcal{R}) \parallel \text{nonce})$$

Each hop validates:

$$\text{Member}(r, \mathcal{R}) \wedge \text{ExtCommit}(X) \wedge \text{Consume}(e_j)$$

**Intent bounds enforce the allowance.** Any admissible route must satisfy:

$$\sum \Delta_{\text{out}} \geq \text{minOut} \quad \text{and} \quad \sum \text{fee} \leq \text{maxFee}$$

Thus re-routing cannot degrade execution beyond the user bound. Percent-style slippage is compiled into minOut deterministically:

$$\text{minOut} = \lfloor \text{quoteOut} \cdot (1 - \epsilon) \rfloor$$

**Why this removes router trust.** If a router proposes a route not in  $\mathcal{R}$ , vault predicates reject it. If it proposes an in-set route violating bounds, it is non-admissible and fails verification. The SDK can also enforce deterministic route ordering to make selection auditable.

## 4 Protocol-Owned Executed Tape and Oracle-Free Perpetual Settlement

### 4.1 Executed Tape Commitments

Each pair  $(A, B)$  defines an executed tape updated on each accepted trade impacting the pair:

$$T_{n+1}^{A/B} = H\left(T_n^{A/B} \parallel \text{TradeDigest}_{n+1}^{A/B}\right)$$

where TradeDigest is a canonical digest of the trade (pair, amounts, vault IDs, parent tips, and relevant external commitments). This tape is not a global chain; it is a deterministically verifiable sequence discoverable via storage nodes and validated by inclusion proofs.

### 4.2 Anti-Distortion Requirement

SoFi enforces anti-distortion via:

1. **Self-trade exclusion:** vault policies can require counterparty diversity (distinct genesis anchors) for tape contribution.
2. **Bounded impact:** tape influence of any single trade capped by trade-size and reserve-ratio predicates.
3. **Cost coupling:** meaningful tape movement requires real trades, real fees, and real reserve movement across independent vaults, making distortion economically punitive and auditable.

### 4.3 Deterministic Funding and Liquidation Predicates

Perpetuals require settlement, funding, and liquidation. In SoFi, these are predicates over committed state:

- **Settlement reference:** bounded function of tape receipts (e.g., median of last  $m$  trades from independent vaults).
- **Funding:** deterministic transfers based on inventory imbalance and tape deviation.
- **Liquidation:** predicate over margin, encumbrances, and settlement reference; liquidation branch becomes valid and mutually exclusive when satisfied.

A perpetual instrument is a DLV implementing these predicates.

## 5 Composability and Sovereignty

### 5.1 Sovereign Execution

Vaults are never transferred to third parties. Funds remain locked until precommitted conditions are met:

$$S_n = H(S_{n-1} \parallel \Delta_n), \quad \Delta_n \Rightarrow \text{DLV unlock valid}$$

Each party verifies:

1. Hash adjacency (proper DSM chain extension)
2. Inclusion proofs (vault state in Per-Device SMT)
3. Invariant satisfaction (pricing formulas hold)
4. Encumbrance commitment satisfaction and claim consumption
5. External commitment existence (if referenced)
6. RouteSet membership and intent bounds (if used)
7. Token conservation (balances sum correctly)

No consensus, no voting, no trusted intermediaries.

### 5.2 Programmable Market Logic

Vault owners define unlock conditions at creation:

- Pricing invariants: constant product, stableswap, or custom formulas
- Trade size limits and reserve-ratio safety bounds
- Tape-bounded price bounds (oracle-free constraints)
- Intent bounds: minOut and maxFee constraints
- Iteration-budget-based expiry (clockless)
- External dependencies: shared commitments for multi-vault atomicity
- Encumbrance rules: explicit cross-margin and priority constraints
- Fee structure: fixed, proportional, dynamic, or tiered

### 5.3 Pre-Commit Forking for Mutually Exclusive Outcomes

Vault owners can prepare exclusive branches at the same parent:

$$C_{\text{fulfill}}^{\text{pre}} = H(h_n \parallel \text{fulfill} \parallel e_1), \quad C_{\text{refund}}^{\text{pre}} = H(h_n \parallel \text{refund} \parallel e_2)$$

Tripwire guarantees only one successor can be accepted. This enables conditional escrow, clockless refunds, and deterministic multi-path outcomes.

## 6 Scalability and Aggregation

### 6.1 Synthetic Liquidity Grids

Instead of central pools, liquidity is aggregated via vault indexing and atomic composition:

1. User specifies TradeIntent (includes minOut/maxFee/maxHops/ $k$ ).
2. SDK discovers vaults via storage-node quorums and validates proofs.
3. SDK computes bounded RouteSet and commits external commitment  $X$ .
4. SDK assembles stitched receipts for chosen route.
5. If any hop is unavailable, SDK selects next admissible route in  $\mathcal{R}$  without changing intent.
6. Vaults unlock when ExtCommit( $X$ ) exists and predicates verify.

Properties:

- Zero custody risk
- Parallelizable verification
- Merkle-verifiable audit trail
- No coordinator signatures
- Deterministic reroute within preapproved bounds

### 6.2 Example: Route-Set Atomic Execution

Alice trades TOKEN A for TOKEN C. SDK computes two admissible routes and commits:

$$\text{TradeIntent} = \{A, 1000, C, \text{minOut} = 880, \text{maxFee} = 10, \text{maxHops} = 3, k = 2\}$$

$$\mathcal{R} = \{r_1 : A \rightarrow B \rightarrow C, \quad r_2 : A \rightarrow D \rightarrow C\}$$

$$X = H(\text{"DSM/route-set"} \parallel \text{TradeIntent} \parallel \text{Canonicalize}(\mathcal{R}) \parallel \text{nonce})$$

For  $r_1$ :

$$\text{Vault}_{A \rightarrow B} : \text{Member}(r_1, \mathcal{R}) \wedge \Delta_A = -1000 \wedge \Delta_B = +950 \wedge \text{ExtCommit}(X) \wedge \text{Consume}(e_1)$$

$$\text{Vault}_{B \rightarrow C} : \text{Member}(r_1, \mathcal{R}) \wedge \Delta_B = -950 \wedge \Delta_C = +900 \wedge \text{ExtCommit}(X) \wedge \text{Consume}(e_2)$$

If  $B \rightarrow C$  is temporarily unavailable, SDK executes  $r_2$  under the same  $X$  so long as it satisfies minOut/maxFee. No renegotiation occurs.

## 6.3 Professional Liquidity Provider Vaults

Professional LPs may operate always-on vaults:

- Large reserves, competitive fees
- Persistent availability for rapid receipt handling
- Many pairs and parameterized vault policies
- Reputation based on observed execution reliability

LPs retain sovereignty and can withdraw liquidity instantly without governance approval.

## 7 Security Model

### 7.1 Threat Analysis

**Storage node compromise:** nodes cannot steal funds, execute unauthorized trades, or modify vault conditions. They can only:

- censor advertisements (mitigated by multi-node replication and quorum queries),
- delay discovery (mitigated by querying many nodes),
- serve false data (detected by proof verification).

**Routing manipulation:** external routers (if used) can suggest suboptimal routes or withhold information, but they cannot:

- force unlocks,
- violate minOut/maxFee bounds,
- bypass RouteSet membership constraints.

Default routing runs in the SDK, making routers optional.

**Vault owner compromise:** if a device is compromised, attacker can withdraw funds within vault logic. Recovery procedures (capsules, tombstones, succession receipts) mitigate device loss and compromise by enabling forward-only recovery without trusting third parties.

### 7.2 Double-Spend and Fork Exclusion

Tripwire guarantees fork exclusion. Assuming EUF-CMA signatures and collision-resistant hashing, generating two distinct receipts consuming the same parent that both verify is negligible. Implications:

- vault funds spent once per parent tip,
- conflicting trades cannot both be accepted,
- conflicting encumbrance-claim consumptions cannot both be accepted.

### 7.3 Frontrunning and MEV Mitigation

SoFi mitigates MEV by:

1. precommitment of TradeIntent and RouteSet,
2. hash-based unlock conditions,
3. external commitments binding multi-vault execution,
4. bilateral settlement rather than public mempool broadcasting,
5. no validator ordering authority.

### 7.4 Clockless Liveness: Refunds Cannot Become Stuck

Safety without liveness is unacceptable. SoFi provides clockless liveness using an iteration budget counter  $\beta$  committed in state:

$$\beta_{n+1} = \beta_n - 1$$

Vault logic enforces:

- **Fulfill branch** valid if proof completes before  $\beta = 0$ .
- **Refund branch** valid if  $\beta = 0$  and fulfill proof is absent.

Since both branches share the same parent tip, only one can be accepted, guaranteeing forward progress without reliance on wall-clock time.

## 8 Economic Model

### 8.1 Vault Owner Incentives

Liquidity providers earn fees specified in vault unlock conditions:

$$\text{fee}_{\text{received}} = f(\Delta_{\text{trade}}, \text{reserves}, \text{tape index}, \text{encumbrances})$$

Fee structures can be:

- fixed per trade,
- proportional (e.g., 0.3%),
- dynamic based on reserve ratios or tape volatility signals,
- tiered by volume,
- auction-like via explicit predicate constraints.

Vault owners retain 100% of fees—no protocol tax, no governance dilution.

## 8.2 Storage Node Economics

Storage nodes are compensated via subscription:

- periodic subscriptions for data availability,
- pricing scales with storage/bandwidth usage, not per transaction,
- staking and provable slashing for misbehavior,
- competitive node market determines pricing.

Nodes have no claim on vault trading fees.

## 8.3 Trader Experience

Traders benefit from:

- no custody risk,
- deterministic settlement with bounds,
- atomic multi-hop execution,
- no gas fees or validator tips,
- fast deterministic reroute within preapproved RouteSet.

Costs:

- vault fees (to LPs),
- optional external compute fee (only if user chooses it),
- subscription for data availability.

# 9 Storage Node Specification (Implementation Blueprint)

This section defines the **exact** responsibilities, data model, and required interfaces for DSM/SoFi storage nodes. These nodes are deliberately non-authoritative: they do not validate trades, do not sign vault state transitions, and cannot create valid receipts. Their only roles are **indexing**, **data availability**, and **retrieval** of already-committed objects plus their associated proofs.

## 9.1 Non-Goals and Hard Constraints

A storage node *must not*:

- modify or synthesize vault conditions,
- generate receipts or signatures that authorize spending,
- reorder trades or act as a settlement coordinator,
- become a trusted truth source.

A storage node *must*:

- serve immutable objects by content hash,
- maintain searchable indices for discovery,
- provide inclusion/freshness proof material required for client verification,
- be horizontally scalable and content-addressed.

## 9.2 Node Data Model

All node-served objects are content-addressed by a hash  $h = H(\text{CanonicalBytes}(\cdot))$ . Nodes store objects in an immutable blob store and maintain separate index structures.

**Object classes.** A conforming node stores the following object classes:

1. **VaultAdvertisement** (discoverability record; mutable via versioning)
2. **VaultState** (state snapshots or deltas; immutable)
3. **Receipt** (stitched receipt objects; immutable)
4. **ExternalCommitment** (ExtCommit payload; immutable)
5. **ProofBundle** (inclusion and freshness proof material; immutable)
6. **TapeDigest** (executed tape digests for pair indexing; immutable)

**Canonical encoding.** Nodes and clients must share a single canonical byte encoding for each object class. The hash commitment is always over canonical bytes. Nodes are not permitted to store “equivalent” alternate encodings under the same logical identifier.

**Immutability and versioning.** If a record must change (e.g., updated advertised liquidity), it is published as a new object version with a new content hash. Indices map logical keys to *latest known version hashes*, but do not erase prior versions.

## 9.3 VaultAdvertisement Schema

A `VaultAdvertisement` is the discoverability surface for a DLV. It is a *statement about an existing vault state*, not an authorization.

Required fields:

- `ad_version`: monotone integer
- `vault_id`: globally unique identifier for the vault
- `owner_genesis`: owner identity anchor
- `token_a`, `token_b`: trading pair identifiers
- `ctp_a_id`: token policy anchor (or policy namespace)
- `state_ref`: content hash of the referenced `VaultState`
- `reserves_a`, `reserves_b`: advertised reserves (optional to hide in privacy modes)

- `fee_policy`: deterministic fee parameters (bounded predicate parameters)
- `unlock_summary`: deterministic predicate summary or hash of predicate bytes
- `enc_commit`: current encumbrance commitment hash  $E$
- `tape_pair_id`: identifier for the pair tape  $(A, B)$
- `freshness_anchor`: reference to freshness proof material (Section 9.6)

**Advertisement acceptance rule.** Nodes accept an advertisement if it is well-formed and references retrievable objects by content hash. Nodes do not evaluate whether the advertisement is *economically honest*; clients verify state/proofs.

## 9.4 Index Structures

Nodes must implement at minimum the following indices:

**(I1) Pair index.** Maps  $(\text{token}_a, \text{token}_b, \text{ctpa\_id}) \rightarrow$  a set of `vault_ids` (or advertisement hashes).

**(I2) Vault head index.** Maps `vault_id`  $\rightarrow$  latest known `VaultAdvertisement` hash and referenced `VaultState` hash.

**(I3) Content-addressed object store.** Maps content hash  $h \rightarrow$  object bytes.

**(I4) Receipt index.** Maps:

- `vault_id`  $\rightarrow$  set of receipts that reference that vault
- `owner_genesis`  $\rightarrow$  receipts referencing that identity anchor
- `ext_commit_hash`  $\rightarrow$  receipts that depend on that external commitment

**(I5) Tape index (per pair).** Maps  $(\text{token}_a, \text{token}_b, \text{ctpa\_id}) \rightarrow$  ordered list of `TapeDigest` hashes. Ordering is defined *deterministically* by the `TapeDigest` itself (see Section 9.5), not by node discretion.

## 9.5 Deterministic Tape Ordering Rule

Nodes must not invent ordering. Instead, each `TapeDigest` carries a deterministic order key:

$$\text{orderKey} = H(\text{"DSM/tape-order"} \parallel \text{pair\_id} \parallel \text{receipt\_hash})$$

The per-pair tape list is stored sorted by  $(\text{tape\_height}, \text{orderKey})$  where:

- `tape_height` is a monotone integer embedded in `TapeDigest` and validated by clients as consistent with the previous committed tape head they accept,
- `orderKey` breaks ties deterministically.

If a node observes two competing tape digests claiming the same `tape_height` with different predecessor references, it stores both; clients reject the losing branch via their own verification rules.

## 9.6 Proof Bundles: What Nodes Must Serve

Clients must be able to verify everything without trusting the node. Nodes therefore must serve explicit proof material sufficient for verification.

**Required proof material for discovery responses.** For each returned `VaultAdvertisement`, nodes must provide a `ProofBundle` containing:

- **State inclusion proof:** proof that the referenced `VaultState` is included under the owner’s committed SMT root (or equivalent authenticated structure).
- **Owner root reference:** the root hash and the corresponding committed anchor reference that binds the root into the owner’s chain.
- **Advertisement-to-state binding:** proof that `state_ref` equals the hash of canonical bytes of the `VaultState` served.
- **Encumbrance binding:** proof that `enc_commit` equals the encumbrance commitment computed from the `VaultState` encumbrance set.
- **Optional freshness proof:** proof that this advertisement is not stale relative to a node-served “latest known owner anchor” (clients may additionally query multiple nodes for corroboration).

**Receipt retrieval proofs.** When serving a `Receipt`, nodes must provide any referenced objects and proofs:

- referenced `VaultStates` (pre and post),
- referenced `ExternalCommitment` payload (if any),
- any inclusion proofs needed to validate these objects against committed roots.

## 9.7 Node Interfaces

A conforming node must expose two categories of interfaces:

1. **Content-addressed retrieval** by hash
2. **Indexed queries** for discovery and tape access

Interfaces can be implemented over HTTP/2, HTTP/3, or a binary RPC transport. The *semantics* below are required.

### 9.7.1 Content Retrieval

`GetObject(h)` Inputs:

- `h`: content hash

Outputs:

- `object_class`

- `canonical_bytes`

Error conditions:

- `NOT_FOUND`: unknown hash
- `TOO_LARGE`: exceeds configured max object size

### 9.7.2 Discovery Queries

`QueryVaults(token_a, token_b, ctpa_id, min_liquidity, limit, cursor)`

Outputs:

- ordered list of `VaultAdvertisement` hashes (or full ads),
- for each, a `ProofBundle` as in Section 9.6,
- `next_cursor` if more results exist.

Deterministic ordering rule:

$$\text{rankKey} = H(\text{"DSM/vault-rank"} \parallel \text{vault\_id} \parallel \text{state\_ref})$$

Node returns results sorted by:

$$(\text{liquidity\_score desc, fee\_score asc, rankKeyasc})$$

where `liquidity_score` and `fee_score` are functions defined by the SDK for ranking; nodes may provide raw fields, but the SDK remains authoritative.

### 9.7.3 Receipt and ExtCommit Queries

`QueryReceiptsByVault(vault_id, limit, cursor)`

Returns receipt hashes plus `ProofBundles` necessary to retrieve referenced objects.

`QueryReceiptsByExtCommit(ext_hash, limit, cursor)`

Enables fast discovery of all receipts associated with a route-set commitment  $X$ .

`GetExtCommit(ext_hash)`

Returns the `ExternalCommitment` payload bytes bound by `ExtCommit(X)`.

### 9.7.4 Tape Queries

`QueryTape(pair_id, from_height, limit)`

Returns `TapeDigest` hashes and associated receipt hashes for the requested height interval. Ordering must satisfy Section 9.5. Nodes store forks; clients decide.

## 9.8 Ingestion Interfaces (Publish)

Nodes must allow clients to publish objects for availability and indexing. Publish does not mean “approve”.

`PutObject(object_class, canonical_bytes)` Node computes content hash and stores if within policy. Returns:

- `h`: computed content hash

Node must deduplicate by hash (idempotent).

`PublishAdvertisement(ad_bytes)` Stores the advertisement and updates indices (I1, I2). If `ad_version` is not monotone for the `vault_id`, the node stores it but does not update “latest” pointer; clients may still fetch older versions by hash.

`PublishReceipt(receipt_bytes)` Stores receipt, updates receipt indices (I4), and optionally updates tape indices (I5) if receipt carries a `TapeDigest` reference.

`PublishExtCommit(ext_bytes)` Stores external commitment payload and updates ext-commit index.

## 9.9 Replication and Availability Guarantees

Nodes are expected to be replicated. The protocol assumption is not “one node is honest” but “clients query many nodes and verify proofs.”

**Client quorum rule (SDK).** The SDK must:

- query at least  $q$  nodes for discovery,
- accept an object only if its hash matches and proofs verify,
- treat inconsistent answers as expected and simply increase query breadth.

**Node replication rule.** Nodes should implement opportunistic replication of popular objects (vault ads, recent tape digests, receipts referenced by many queries). Replication is content-addressed: correctness is guaranteed by the hash.

## 9.10 Resource Accounting and Subscription Gate

Nodes are funded by subscription for data availability rather than per-transaction fees. Implementation requires:

- a deterministic **accounting key** that maps each stored object to a paying identity (e.g., `payer_genesis`),
- an **allow/deny gate** that rejects publishes exceeding paid quota.

**Quota model.** A node must enforce a quota over:

- total stored bytes for a payer,
- monthly served bytes (bandwidth),
- max object size,
- max publish rate (to prevent spam).

**Deterministic enforcement.** Enforcement decisions must be reproducible given the same accounting state: publish accepted if and only if:

$$\text{quotaRemaining}(\text{payer}) \geq \text{cost}(\text{object})$$

Nodes must expose:

`GetQuota(payer) → (storage_remaining, bandwidth_remaining, reset_policy)`

so SDK can predict publish acceptance.

## 9.11 Operational Limits and Performance Targets

These targets are required for practical DEX UX at scale:

- **Discovery p95 latency:**  $\leq 250$  ms for `QueryVaults` with limit 50 (excluding wide-area network extremes).
- **Object retrieval p95 latency:**  $\leq 250$  ms for objects up to 256 KiB.
- **Max single object size:** 2 MiB (hard reject) to prevent abuse and keep retrieval fast.
- **Index update time:** advertisement publish to index visibility  $\leq 1$  second on a healthy node.

Nodes should support horizontal sharding by hash prefix for object store and by (`pair_id`) for tape indices.

## 9.12 Security and Abuse Controls

Nodes must implement:

- per-identity publish rate limits,
- per-IP rate limits for unauthenticated queries,
- request cost accounting (query complexity caps),
- ban lists for abusive peers (policy, not protocol).

These controls are non-authoritative: they protect node resources, not protocol correctness.

## 9.13 Deterministic Error Codes

Nodes must return deterministic error codes:

- `NOT_FOUND`
- `INVALID_FORMAT`
- `TOO_LARGE`
- `RATE_LIMITED`
- `QUOTA_EXCEEDED`
- `UNSUPPORTED`
- `INTERNAL_ERROR`

SDK maps these to user-visible meanings without ambiguity.

## 10 Implementation Considerations

### 10.1 Vault State Representation

A vault state includes:

$$\text{State}_n = \left\{ \begin{array}{l} \text{genesis}_{\text{owner}}, \text{DevID}_{\text{owner}}, \\ \text{reserves}_A, \text{reserves}_B, \\ \text{unlock conditions, fee structure,} \\ \text{encumbrance commitment } E, \mathcal{E}, \\ \text{tape references } (T^{A/B}), \\ \text{parent tip } h_n, \\ \text{Per-Device SMT root } r_{\text{owner}} \end{array} \right\}$$

Transitions:

$$h_{n+1} = H(\text{State}_n \parallel \Delta_{n+1} \parallel \sigma)$$

### 10.2 Routing Algorithm Sketch (SDK-Resident)

function findRouteSet(tokenIn, tokenOut, amountIn, minOut, maxFee, maxHops, k):

1. Query a quorum of storage nodes for vault candidates
2. Verify inclusion proofs and freshness anchors
3. Build liquidity graph and evaluate satisfiable predicates
4. Compute up to k admissible routes with  $\leq$  maxHops
  - a. For each route compute expected out and total fees
  - b. Enforce: out  $\geq$  minOut and fees  $\leq$  maxFee
  - c. Bind each hop to required encumbrance claim IDs
5. Canonicalize RouteSet
6. Commit X = H("DSM/route-set" || TradeIntent || RouteSet || nonce)
7. Return RouteSet + proof skeletons

### 10.3 Client Verification Flow (SDK-Resident)

function verifyAndExecute(route, routeSetCommitX, tradeIntent, vaults):

1. Fetch vault states via storage-node quorum
2. Verify inclusion proofs (vault state in owner's authenticated root)
3. Verify encumbrance commitment and claim availability
4. Verify route is Member(route, RouteSet) committed by X
5. Recompute each hop and check invariants
6. Enforce intent bounds: totalOut  $\geq$  minOut and totalFees  $\leq$  maxFee
7. Verify ExtCommit(X) existence and binding to enc commits
8. Verify signatures on canonical commit bytes
9. Verify claim consumption (encumbrance removal) is valid
10. Publish stitched receipts for the route
11. On hop failure/unavailability, select next admissible route in RouteSet and retry

Property	Traditional DeFi	SoFi
Custody	Pooled in contracts	Sovereign per user
Execution	Global consensus	Bilateral verification
Liquidity	Concentrated pools	Distributed vaults with atomic composition
Finality	Probabilistic	Deterministic
MEV	Validator extraction	Minimal (no mempool, pre-commits)
Fees	Per transaction (gas)	Subscription-based (data availability)
Programming model	Turing-complete smart contracts	Bounded predicates
Upgradability	Governance votes	Individual choice
Composability	Contract calls	External commitments + encumbrance binding
Censorship risk	Validator discretion	Cryptographic proofs + quorum discovery
Cross-margin	Pool-driven rehypothecation risk	Encumbrance-committed, non-custodial
Perpetuals	Oracle dependence	Tape-derived settlement + anti-distortion
Reroute	Tx fails / repriced manually	Native (RouteSet + minOut/maxFee)

## 11 Comparison to Traditional DeFi

## 12 Comparison to Flying Tulip

This section maps Flying Tulip-style guarantees to SoFi-native invariants. SoFi does not import Flying Tulip’s project structure; it enforces the underlying guarantees as protocol constraints implemented in vault predicates and validated by the SDK.

## 13 Why Wrapper-Projects Become Redundant Under SoFi

Flying Tulip-style designs compensate for constraints introduced by consensus chains: pooled custody, validator ordering, global fee markets, and externally governed oracle feeds. SoFi removes those constraints at the substrate layer and therefore achieves the same economic guarantees as verifiable invariants:

- **Principal protection** becomes a DLV property via fulfill/refund branching under Tripwire (Section 5.3).
- **Unified margin** becomes safe via deterministic encumbrance commitments preventing double-pledging and defining priority (Section 3.3).
- **Zero-fee execution** becomes native because there is no global fee market or validator ordering (Section 7.3); infrastructure is subscription-funded (Section 8.2).

- **Oracle politics** are avoided by tape-derived settlement with explicit anti-distortion rules (Section 4).
- **Route reliability and best execution** are achieved by SDK-resident RouteSets and intent bounds, not by trusting third-party routers (Section 3.5).

## 14 User Experience: How the SDK Hides Determinism and Proof Complexity

SoFi is engineered so protocol complexity does not leak into user experience. The system is deterministic and verifiable, but the interface exposes only the minimal decisions a human should be asked to make. The SDK (STK) absorbs the rest.

### 14.1 The User Contract: A Minimal Surface With Hard Guarantees

The UI presents:

1. tokenIn, amountIn, tokenOut,
2. allowance expressed as minimum output minOut,
3. fee ceiling maxFee,
4. complexity bound (maxHops,  $k$ ).

Guarantee:

*Either you receive at least minOut within maxFee, or the trade does not execute.*

### 14.2 Trade UX Pipeline

The UI shows one “Swap” action. Internally, the SDK runs discovery, routing, proof assembly, verification, publish, and deterministic reroute under a single committed TradeIntent and RouteSet.

### 14.3 Deterministic Failure Semantics

The SDK classifies failures into deterministic categories (no admissible routes, hop unavailable, encumbrance conflict, predicate rejection, stale index, budget exhausted) and maps these to human-readable messages.

### 14.4 Caching and Prefetch

The SDK minimizes perceived latency by caching authenticated roots, incrementally verifying only deltas, prefetching vault states during quote, parallelizing independent proof checks, and bounding hop count and alternatives.

### 14.5 Why “Built Into the SDK” Matters

SDK-resident routing and verification prevents the emergence of trusted routing gatekeepers: external routers remain optional compute helpers whose outputs are checked against committed bounds.

## 15 Conclusion

SoFi enables a sovereign liquidity system where participants retain control over assets until deterministic execution occurs. Storage nodes remain non-authoritative indexing and availability infrastructure; all correctness is enforced by client-side deterministic verification. With DLVs, external commitments, encumbrance commitments, route-set rerouting, and tape-derived settlement, SoFi enforces guarantees that wrapper projects attempt to approximate using policies layered on top of consensus chains. The storage node specification in Section 9 is intended to be implemented directly as part of a complete SoFi blueprint.

Flying Tulip Feature/Guarantee	Goal It Tries to Achieve	SoFi Native Mechanism (No Wrapper)
Perpetual Put / Redemption Right	Principal safety under failure modes	Pre-commit forking with fulfill/refund exclusivity under Tripwire (Sections 5.3, 7.2); deterministic unlocks (Section 2.3); clockless liveness via iteration-budget fallback (Section 7.4).
Unified Cross-Margin System	Reuse collateral across positions without moving funds	Deterministic encumbrance commitments (Section 3.3) bind obligations to a committed claim-set; consumption proofs prevent double-pledge; deterministic priority under Tripwire (Section 7.2).
Oracle-less Perps via Internal Spot Data	Avoid external oracle governance	Protocol-owned executed tape (Section 4.1) plus anti-distortion rules (Section 4.2); funding/liquidation as verifiable predicates (Section 4.3).
Zero-fee trading via subsidy/hardening	Remove execution friction and MEV tax	No global mempool or validator ordering market (Section 7.3); infrastructure monetized via subscription for data availability (Section 8.2); trades execute by deterministic proof (Sections 2.3, 10.3).
Best execution packaging / route reliability	Hide fragmentation and route failures	SDK-resident routing/verification (Section 3.4); deterministic RouteSet fallback (Section 3.5); user intent bounds (minOut/maxFee) enforced by predicates (Sections 3.5, 10.3).
Incentives via buybacks / managed alignment	Align LPs and ecosystem growth	LPs define and keep 100% of fees in vault predicates (See