The Vault Contract Ecosystem

for Revenue Distribution and Digital-Asset Securities

Technical Whitepaper

A Reconciliation-First Control Plane for Deterministic, Auditable Proposal Execution on EVM

Jason Aubrey
Co-Founder & Chief Architect
GigaStar Technologies, LLC

v1.5 2026-07-24

Contents

1 Abstract

This whitepaper presents the GigaStar Vault Ecosystem, a production smart-contract and integration architecture for revenue-sharing digital securities in the Creator Economy. The system coordinates issuance, recurring revenue allocation and distribution, instrument conversion, and secondary-market transfers while preserving a public, independently visible record of proposal authorization and execution.

The architecture separates an off-chain control plane from an authoritative on-chain data plane. A governance kernel (the Vault) manages quorum-gated proposals and routes execution to specialized manager contracts. Paged uploads and monotonic execution cursors bound transaction cost and allow large workflows to progress across multiple transactions. `CallTracker` provides application-level idempotency and deterministic replay semantics for off-chain automation when transaction outcomes are uncertain because of timeouts, stale nodes, process restarts, or lost receipts.

The current architecture entered production in March 2026 as a successor to the prior GigaStar system that had operated since May 2023. As of this publication, the combined platform has supported approximately 30,000 Investors, 67 active instruments, 420,000 revenue transfers, and approximately $1.5 million in cumulative creator revenue distributions. The smart-contract system was independently assessed by CertiK, is source-verified on Polygon PoS, and is supported by unit, integration, static-analysis, and storage-layout compatibility checks.

This paper describes the architecture, primary workflows, protocol invariants, security and trust boundaries, public audit surfaces, deployment model, and production validation evidence. It is intended both as a technical integration primer and as a broader description of the system's design rationale.

2 Audience, Scope, and Non-Goals

The primary audience includes regulated vendor partners, technical reviewers, security assessors, blockchain and financial-systems engineers, and stakeholders evaluating GigaStar's operating architecture. The paper focuses on the blockchain and blockchain-adjacent components required to understand how proposals are authorized, executed, reconciled, and publicly verified.

The following boundaries apply:

- **Technology scope**: The paper describes the Vault smart-contract ecosystem, its integration contract with off-chain request producers, and the public records produced by those workflows.
- **Compliance scope**: The architecture is designed to support regulated workflows, but this paper does not replace offering documents, legal advice, transfer-agent procedures, or entity-specific compliance policies.
- **Security scope**: Public trust assumptions and protocol controls are described. Sensitive operational details, including key custody and incident-response procedures, are intentionally omitted.
- **Availability scope**: The protocol provides deterministic reconciliation and bounded progress; it does not guarantee availability of the underlying blockchain, RPC providers, stablecoins, or other external infrastructure.
- **Investment scope**: This document is a technical publication, not an offer to sell securities or investment advice.

3 Introduction

The Creator Economy continues to expand as Creators monetize digital work, generate recurring revenue, and build financial relationships with their audiences. Small Creators seek capital to grow, large Creators seek ways to diversify, while Fans and Investors seek regulated and transparent ways to participate. Enabling these interactions at scale, safely, efficiently, and with clear auditability requires a structured approach to revenue-sharing securities and their lifecycle operations.

GigaStar introduced a new asset class with a simple model: Creators sell a portion of their future revenue while Fans and Investors buy or trade those revenue rights. The security represents a claim on future cash flows, not ownership of the underlying digital asset itself. This distinction enables modular financing arrangements while preserving the Creator's digital asset ownership and control.

From an engineering perspective, the core challenge is operational: issuance, recurring revenue allocation, and

transfer events must execute deterministically, remain verifiable under partial progress, and tolerate real-world failure modes such as timeouts, retries, and gas-bounded execution. The Vault Ecosystem is designed to provide these properties through on-chain governance, bounded execution, and a reconciliation-first orchestration model.

3.1 History & Motivation

GigaStar's legacy system was built during a period of uncertainty around many factors including market maturity, regulatory expectations, technological direction, investor preferences, and a quick go-to-market strategy. Web3 offered a transparent ledger, a distributed authorization layer, and a low-cost mechanism for distributing revenue to a global Investor base. These benefits had to be balanced against practical user constraints, including a heterogeneous population of existing Web3 users, newcomers with limited Web3 familiarity, and users preferring a Web2-centric experience.

The legacy system entered production in May 2023 and demonstrated that regulated revenue-sharing workflows could operate at meaningful scale on a public ledger. By this publication, GigaStar had distributed approximately $1.5 million cumulatively to Investors, with the cumulative total increasing as monthly distribution amounts vary. The system also operated through ecosystem-wide transitions, including Circle's migration from bridged USDC.e to native USDC on Polygon PoS[1]. The legacy design prioritized EVM[2] portability, wallet compatibility, user familiarity, and low gas fees via Polygon PoS[3] and ERC-721[4]. However, ERC-721 imposed long-term limits around fungibility, liquidity, and secondary-trading mechanics. It also surfaced operational requirements unique to recurring revenue distribution at scale, including stablecoin transitions, off-ramp constraints, and contract maintenance.

The Vault Ecosystem is a replacement architecture designed to support secondary market trading, stronger governance, deterministic execution, and independent public workflow audits. This new system replaces ERC-721 usage with ERC-20[5] and ERC-1155[6] for both secondary market and maintenance benefits. The system supports a ledger in a single ERC-1155 contract with an ERC-20 pass-through wrapper per ERC-1155 Token ID.

4 System Overview

4.1 Hybrid Architecture

The Vault Ecosystem spans on-chain contracts and off-chain services. This paper primarily focuses on on-chain components, with off-chain components described where necessary to define inputs, outputs, and operational workflows.

- **On-Chain Layer**: EVM contracts responsible for securities issuance and ownership management, recurring revenue distributions, proposal-oriented governance, role-based access control, activity telemetry, Investor and Issuer safeguards, and upgradeability.
- **Off-Chain Layer**: External integrations, data pipelines, governance workflows, transaction coordinators, services for frontend, backend, and durable data storage.

The system is on-chain where appropriate for governance, revenue management, and public transparency, and off-chain where appropriate for Web2 integration and faster development cycles.

4.2 Roles

The Vault Ecosystem is role-gated. The key roles are:

- **Public**: Read-only observers (for auditing and transparency)
- **Agent**: Privileged automation for proposals (creating, uploading, sealing, and executing)
- **Voter**: Approves or rejects proposals by quorum
- **Admin**: Configuration and upgrade authority

4.3 System Topology

The system topology shows many entities and concerns where each could be addressed in much detail. However, for brevity much of this paper will focus on the solution stack described in the next section.

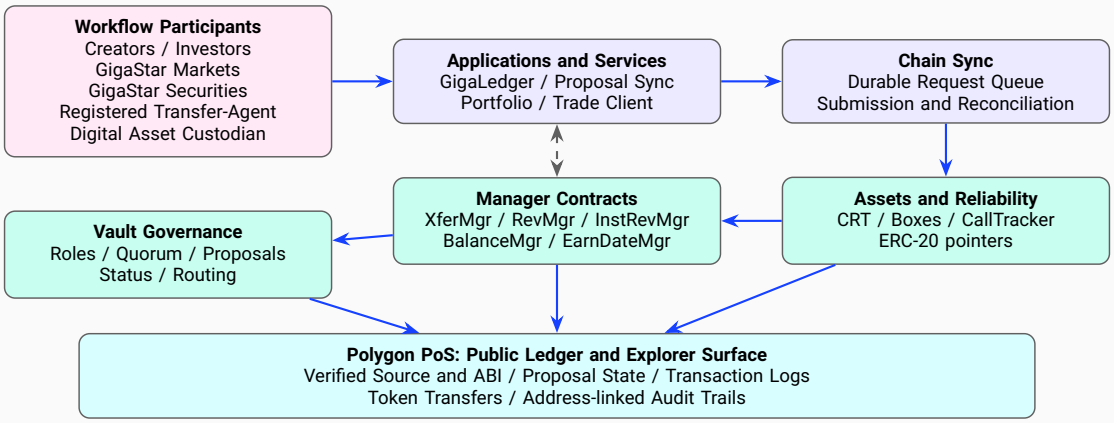

Figure 1: System Topology

4.4 Trust Boundaries

The system is designed around a clear trust boundary as seen from a vertical slice of the solution stack:

- **Control-plane (off-chain)**: proposes and drives workflows. Transactions may experience uncertain outcomes (timeouts, dropped receipts, restarts) and therefore require deterministic retry and reconciliation.
- **Data-plane (on-chain)**: the authoritative state machine. Contracts enforce proposal lifecycle rules, role gating, upload integrity, bounded execution, and public auditability - in addition to attributes intrinsic to the underlying network such as a distributed ledger and cryptographic proofs.

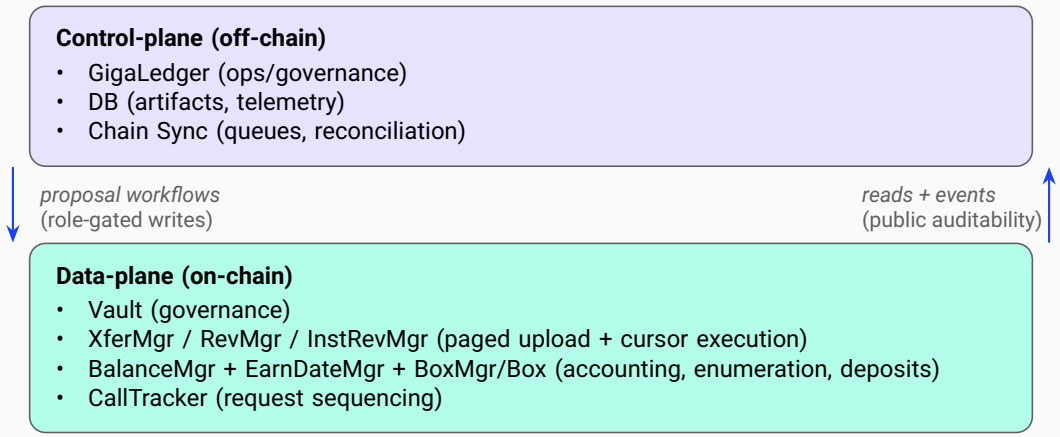

Figure 2: Reconciliation-first control-plane and the authoritative on-chain data-plane.

5 Contract Architecture

The Vault Ecosystem is composed of EVM contracts deployed with explicit cross-contract references, operating as a mesh of specialized services. At release v1.5, the EIP-170[7] contract-code limit of 24 KiB remains a material deployment constraint and has influenced the modular decomposition as shown in Appendix A. Future network changes will likely alter implementation tradeoffs, but they do not affect the architecture or verified deployment described in this paper.

5.1 Primary Contracts

- **Vault**: Governance kernel providing proposal lifecycle, role tracking, and manager contract routing
- **Crt**: Channel Revenue Token (ERC-20[5] and ERC-1155[6]) supporting a scalable, fungible security token

5.2 Manager Contracts

- **XferMgr**: Instrument transfer proposals (native, ERC-20[5], ERC-1155[6])
- **RevMgr**: Revenue transfer proposals
- **InstRevMgr**: Instrument revenue proposals
- **BalanceMgr**: Tracks owner balances
- **EarnDateMgr**: Instrument and earn-date indexing
- **Erc20PtrMgr**: Tracks an Erc20Ptr contract per instrument
- **BoxMgr / Box**: Manages a deposit address per revenue stream for segregated audit trails

5.3 Reliability Substrate

- **CallTracker**: Deterministic retries for off-chain callers via (caller, seqNum, reqId), replay acknowledgment, and call result persistence

5.4 Off-Chain Access

Contract state is publicly accessible via the blockchain ledger. It can be inspected from block explorers, custom applications, GigaStar's Portfolio web application for Investors, or the internal GigaLedger application for operations and governance. This state provides full transparency to interested parties.

During phases of issuance, trading, or revenue distribution, proposals are created by operations staff and their actions provide inputs upstream of a privileged Agent automation role. Voters review proposals via the GigaLedger and approve or reject them. Upon approval, backend services execute proposals to change on-chain state, such as distributing revenue or transferring securities.

6 Design Principles and Alternatives

The Vault Ecosystem is organized around the following principles:

- **Public Auditability**: Public view functions and events enable independent workflow verification
- **Quorum-Gated Proposals**: Material actions are authorized by multiple Voters
- **Bounded Work**: Large payloads are uploaded in pages for gas-bounded progress; execution is cursor-based to guarantee resumability
- **Determinism by Protocol**: Uncertain outcomes due to a failure in request-response pipelines are handled by a protocol of reliability mechanisms that resolve to a certain outcome (e.g. replay-safe requests, idempotency, request identifiers, sequence numbers, cached results, replayable events)
- **Modularity**: The system is decomposed across multiple implementation contracts both for code hygiene and because of EVM runtime bytecode constraints as demonstrated in Appendix A.

6.1 Proposal-Driven Governance

Proposal-driven governance provides a verifiable control surface for high-impact state transitions (issuance, revenue allocation, and transfers). The intent is to make every material change:

- Explicitly authorized (quorum-voted)
- Replayable and resumable under gas limits
- Independently auditable by public observers (full transparency)

Alternative: Single-signer actions. A single-signer model directly executes state mutations without a transparent quorum. While simpler, it reduces audit structure, complicates post-incident attribution, and increases blast radius under key compromise.

6.2 Paged Upload and Sealing

Operational payloads can be large (e.g. many transfers, many owners, many earn-date allocations). Uploading in pages enforces a bounded per-transaction cost and allows robust retries. Sealing makes the payload immutable so that governance decisions can refer to a stable artifact.

Alternative: Single-transaction upload. A single upload transaction is simpler but fails for many realistic batch sizes under EVM gas constraints. Pages also amortize gas fees and processing time by grouping line items into a single transaction.

Alternative: Off-chain payloads only. On-chain payloads allow a proposal request to be independently verified via Web3 methods and improve auditability due to the inherent immutability and distributed nature of the underlying ledger.

6.3 Cursor-Based Execution

Execution is modeled as a resumable state machine. Cursors provide monotonic progress and deterministic retry under partial completion and out-of-gas conditions.

Alternative: Execute-all-or-revert. Transaction gas bounds do not yet support the required throughput for most proposals. Sharding proposal payloads to fit gas bounds reinvents proposals with less elegance.

6.4 Modular Manager Contracts

EIP-170[7] contract-size constraints materially influence implementation which would otherwise require a dynamic library that would have a competing set of complexities and constraints. Separating proposal types across managers focuses each contract on a smaller set of invariants and provides logical and organizational boundaries for logic and data.

Alternative: Monolithic manager. A monolithic contract with a dynamic library can reduce cross-contract hops but increases upgrade risk, auditing complexity, and size-limit pressure.

6.5 CallTracker Exists On-Chain

`CallTracker` addresses a real-world integration gap: off-chain automation operates under uncertain transaction outcomes (e.g. stale nodes, timeouts, network flaps, process restarts). By making retry semantics explicit on-chain via (`caller`, `seqNum`, `reqId`), the protocol becomes deterministic even when the network is not (inspired by ARPANET-era reliability[8,9]).

Alternative: Purely off-chain idempotency. Purely off-chain idempotency fails under ambiguous “did it execute?” conditions. For example, pre-calculating a transaction hash before submission could in theory allow the transaction status to be known under uncertain conditions but due to nuances of latency and stale nodes, a false negative could occur leading to a request replay and duplicate action. `CallTracker` avoids this scenario via a first-class signal for determinism-by-protocol and on-chain idempotency.

6.6 Explicit Public Audit Surfaces

While many systems may use public read functions and events as simply "debug output", this system considers these to be a product surface for Investors, regulators, counterparties, and internal operations. The goal is to make key workflows publicly verifiable without privileged access, providing full transparency.

Many projects point to the blockchain as an implicitly public ledger and claim transparency. But in reality such workflows are often opaque by default for complex workflows. Transparency as a feature delivers value beyond the norm to ensure workflows have inputs and outputs that are simple to decode.

Alternative: Explorer-only visibility. Block explorers provide potential transparency but their utility is often limited or disabled without a proper public audit surface.

7 Securities

7.1 Securitization

In this paper, an instrument is a revenue-sharing security. The platform is designed to support workflows for offerings conducted under Regulation Crowdfunding through registered intermediaries and related regulated entities[10,11]. GigaStar Technologies provides the technology platform; GigaStar Markets operates as a funding portal; and GigaStar Securities operates as a broker-dealer. Other regulated and operational relationships participate where required but are outside this paper's disclosure scope.

7.2 Instrument Lifecycle

An instrument progresses through phases of the following lifecycle:

$$\text{Issuance} \rightarrow \text{Holding Period} \rightarrow \text{Roll/Conversion} \rightarrow \text{Trading}$$

After Issuance, revenue is distributed based on the terms of the security/instrument - extending from the holding period through the trading phase.

Securities purchased in a Regulation Crowdfunding transaction are generally subject to a one-year resale restriction[10]. Eligibility to transition from a non-tradable issuance instrument to a tradable instrument is enforced through off-chain compliance controls. After eligibility is established, the protocol supports conversion into a tradable instrument. The tradable instrument may contain units from multiple issuances, each converted into economically equivalent fungible units.

For example, over a period of time, a Creator may have 3 issuances (ABC.1, ABC.2, ABC.3) and each instrument eventually rolls into ABC.0 - a pool of fungible ABC revenue sharing units (RSUs). The roll may include a conversion to split/merge shares to create equivalent revenue sharing rates/percentages.

8 Definitions and Notation

This section defines recurring terms used normatively throughout this paper.

8.1 Actors and Roles

- **Creator**: A monetized content creator who may become an Issuer of revenue shares via GigaStar.
- **Investor**: A revenue share owner via either the primary or secondary market.
- **Public**: Any observer with read-only access to on-chain state and events for transparency and auditability.
- **Agent**: A privileged automation role responsible for proposing, uploading, sealing, and executing workflows.
- **Voter**: An authorized role that approves or rejects proposals by quorum.
- **Admin**: A privileged role for configuration changes and upgrades.

8.2 Proposal Terms

- **Proposal**: A governance-gated request to transition on-chain state using a structured payload and lifecycle.
- **Paged Upload**: Payload ingestion via multiple transactions, each bounded by gas limits.
- **Seal**: A lifecycle transition that makes a proposal payload immutable for governance review and execution.
- **Cursor-Based Execution**: Resumable execution where each call advances a stored cursor until completion.
- **Terminal Status**: A final proposal state after which no further state mutations occur for the proposal.

8.3 Reliability and Sequencing

- **Retry Key**: The tuple `(caller, seqNum, reqId)` used to make on-chain outcomes deterministic under retries.
- **caller**: The address submitting a state-mutating transaction (typically an Agent).
- **seqNum**: A per-(contract, caller) monotonically increasing sequence number enforcing FIFO ordered writes.
- **reqId**: A request identifier (UUID) used to uniquely identify the intended operation under replay.
- **Replay**: A resubmitted request that behaves idempotently with respect to the original compact result.
- **Reconciliation**: The process of recovering correct state by querying authoritative on-chain state and (when required) replaying prior events.

8.4 Normative Language

The keywords **MUST**, **MUST NOT**, **SHOULD**, and **MAY** are to be interpreted as described in RFC 2119[12].

9 Core Invariants

This section lists system invariants treated as normative correctness conditions. They serve as a checklist for audits, reviews, and future refactors.

9.1 Governance and Status Invariants

- `I1` **(Role-gated writes)**: All state-mutating operations are restricted to appropriate roles (Agent/Voter/Admin), with Public limited to read surfaces.
- `I2` **(Proposal status as a DAG)**: Proposal status transitions are a directed acyclic graph (DAG) and as such terminal proposal states are final (no reactivations).
- `I3` **(Payload immutability after seal)**: Once a proposal is sealed, its payload and relevant headers are immutable; governance decisions apply to a stable artifact.

9.2 Upload and Execution Invariants

- `I4` **(Append-index correctness)**: Paged upload enforces strict append indexing via `iAppend` and `total`. Missing or duplicate pages are rejected deterministically.
- `I5` **(Cursor monotonicity)**: Execution cursors only move forward. If a proposal's status is terminal (e.g. Executed) then a request replay must not advance the proposal state.
- `I6` **(Bounded progress)**: Each execute call performs bounded work and returns typed progress, enabling resumption without requiring unbounded gas.

9.3 Accounting and Transfer Invariants

- `I7` **(Conservation / consistency)**: Token transfers and balance mutations preserve defined conservation rules (e.g. instrument supply, per-source balance constraints, or accounting totals).
- `I8` **(Allocation sums)**: For instrument revenue, per-owner allocations must reconcile to the per-(instrument, earn date) totals within defined rounding rules.
- `I9` **(No duplicate effects under retry)**: Under the `CallTracker` protocol, at most one state-changing call can be accepted for a given (`caller, seqNum`); replays return the original result with a block number to retrieve related events from the historical ledger.

9.4 Auditability Invariants

- `I10` **(Public verifiability)**: For each proposal workflow, sufficient public on-chain read surfaces and events exist to independently verify: proposal payload, lifecycle status, both proximate and terminal outcomes.

10 Proposal Model & Lifecycle

10.1 Proposal Types

The primary proposal types covered in this paper are:

- **Instrument Revenue** (`PropType.InstRev`): Instrument related Investor revenue tracking
- **Token Transfers** (`PropType.Xfer`)
  - **Instrument Transfers**: Ownership changes from issuance or trades
  - **Revenue Transfers**: Revenue sharing payments to Investors

10.2 Proposal Lifecycle By Role

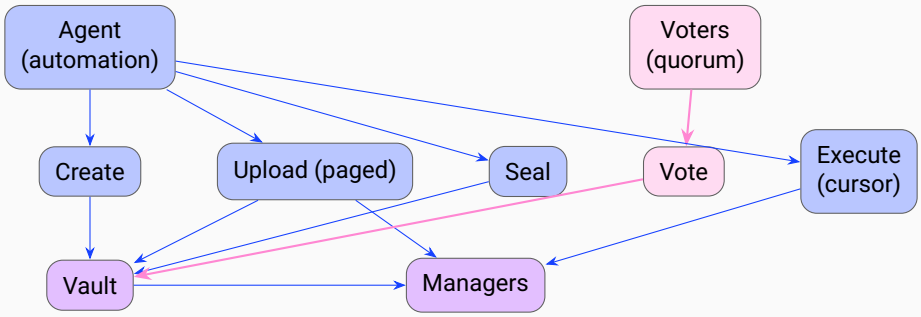

Figure 3: Role-oriented proposal workflow

GigaLedger uses crypto-signing for request authenticity, authentication, authorization, and attribution:

- **Proposal Request**: Each proposal request is signed similar to a Web3 tunnel through the Web2 stack.
- **Proposal Voting**: Votes are cast via off-chain signing (EIP-712[13]) and relayed to the EVM to centralize on-chain concerns (gas, connectivity, etc). Vault also supports direct on-chain signing.

10.3 Proposal Lifecycle By Status

A proposal generally transitions from status `Pending` to `Executed` but may deviate to another terminal state.

Figure 4: Status-oriented proposal workflow

The status model is explicitly partitioned to allow consumers to treat a proposal as terminal without enumerating every terminal reason.

10.4 Manager Patterns

Manager contracts implement two recurring patterns:

- **Paged Upload**: Strict append-indexing using `iAppend` and `total` for deterministic payload ingestion
- **Execution Cursor**: Stored cursors enable proposal progress and resumption, with typed return codes for progress and completion

11 Primary Workflows

The following workflows contain various checks and simulations to fail fast where practical. However, request sequencing is assumed to reduce algorithmic complexity and ensure prerequisite state exists per proposal. Core behaviors are delegated from Vault to manager contracts. Multiple instruments are supported per proposal.

11.1 Workflow A: Instrument Transfer

Prerequisites: Off-chain actions necessitate instrument transfers (e.g. issuance or trades). An issuance respects the security terms and a trade respects the outstanding instrument supply.

This workflow supports multiple instrument transfers and the proposal sequence:

- **Create**: Agent creates a transfer proposal in Vault
- **Upload**: Agent uploads the proposal payload via pages of lines (i.e. transfer details)
- **Seal**: Agent seals proposal
- **Vote**: Voters approve or reject the proposal via a quorum of `Vote.Yay` or `Vote.Nay`
- **Execute**: Agent executes via Vault; Vault delegates to XferMgr until completion

Minor Note

Transfers of native coins (e.g. ETH) and other tokens (ERC-20[5] and ERC-1155[6]) are supported to handle edge cases such as reversing incorrect upstream deposits that would otherwise be permanently locked/lost.

11.2 Workflow B: Instrument Revenue

Prerequisite: Instrument revenue was not previously allocated for the given key (instrument, earn date). The ownership snapshot per key is valid and complete with respect to the outstanding instrument supply.

This workflow allocates revenue per Instrument and per related Investors for an earn date via the proposal sequence:

- **Create**: Agent creates an InstRev proposal in Vault
- **Upload InstRev**: Agent uploads instrument revenue lines in pages
- **Upload Owners**: Agent uploads ownership snapshots per key (instrument, earn date)
- **Seal**: Agent seals proposal
- **Vote**: Voters approve or reject the proposal via a quorum of `Vote.Yay` or `Vote.Nay`
- **Execute**: Agent executes via Vault; Vault delegates to InstRevMgr until completion

Minor Note

If instrument revenue was previously allocated for the given key then it is considered an adjustment that is non-destructive and appended to the ledger history for public observation (i.e. full transparency).

11.3 Workflow C: Revenue Transfer / Distributions (RevDist)

Revenue distributions are modeled as transfer proposals with a revenue-distribution flag. This mode is constrained to ERC-20[5] (due to contract size limitations), and it commonly follows the execution logic of an InstRev proposal with Vault delegation to RevMgr.

A key safety property in RevDist mode is a per-source running balance simulation anchored from BalanceMgr, which reduces the likelihood of executing underfunded batches and provides deterministic failure modes when inputs are inconsistent. This provides an extra layer of safety against upstream protocol violations that are not expected during normal operation. This is defensive handling of upstream inconsistencies, not a substitute for correct request production.

12 Reliability Substrate: CallTracker Contract

12.1 Uncertain Outcomes and Retries

Automation driving on-chain writes must tolerate uncertain outcomes arising from RPC failures, mempool behavior, network interruptions, and process restarts. Correctness therefore depends on deterministic behavior across multiple reliability layers.

An off-chain request may conflict with on-chain state because its sequence number is stale or premature, or because the request has already been accepted. These are expected recovery conditions rather than exceptional protocol failures: authoritative on-chain state prevails, and the off-chain producer must reconcile before proceeding.

12.2 Deterministic Retry Key

`CallTracker` provides deterministic retry semantics keyed by:

- **reqId**: Request identifier (UUID)
- **caller**: Address submitting requests, such as an Agent
- **seqNum**: Expected sequence number (monotonic)

12.3 Protocol Properties

For each calling account and target contract, `CallTracker` establishes the following protocol properties:

- **Request identity**: A `reqId` identifies one logical request for its full lifetime.
- **Sequence uniqueness**: A sequence number already associated with one request identifier cannot be reused for a different request identifier.
- **Request uniqueness**: A request identifier cannot be moved to another sequence number.
- **Replay availability**: A valid request may be replayed at any time after the first submission.
- **At-most-once logical effect**: A replay of an accepted or completed request returns the stored result rather than applying the logical mutation again.
- **Monotonic progress**: Paged uploads and cursor-based execution advance without silently moving backward.
- **Recoverability**: The stored result and referenced transaction-log range are sufficient to reconstruct the original protocol outcome.

These properties require conforming behavior from off-chain request producers. In particular, producers must durably persist request identity and sequence state, reconcile before allocating a new sequence number, and treat protocol errors as integrity signals rather than bypass conditions.

12.4 Deterministic Replay

If a call is replayed with the same key, the system sets a replay flag and returns a compact `CallRes` tuple for recovery and reconciliation:

- **reqId**: Request identifier (UUID)
- **rc**: Call return code, often relates to status/progress
- **lrc**: Line return code, often relates to a line/item
- **count**: Item count, often relates to items processed
- **blockNum**: For replay to find the original transaction

`CallRes` is both stored in contract state and emitted as an event for observability. The tuple is compact to limit storage usage and intentionally does not duplicate the full event payload. A producer can recover the related events by querying the historical transaction-log range beginning at `blockNum` and matching the `reqId`. Because an individual RPC node may temporarily expose stale view state, producers reconcile the stored result, transaction receipt, and historical logs rather than relying on one immediate view call.

12.5 CallTracker Sequencing

After the configured confirmation policy is satisfied, the ledger provides the authoritative outcome. `CallTracker` makes replay a first-class protocol operation so that off-chain ambiguity does not create duplicate logical effects.

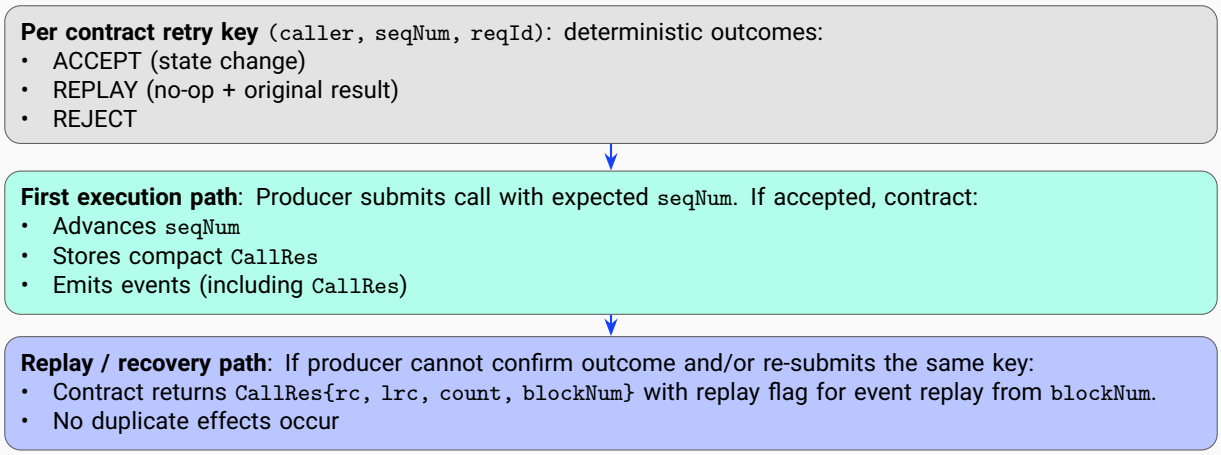

Figure 5: CallTracker sequencing

This table defines the normative action taken by `CallTracker` under the cross-product of `seqNum` and `reqId`. It is intended to be read as a total function over request inputs.

	Request Factors		Outcome	
Case	**SeqNum**	**ReqId**	**Action**	**Description**
000	$<$ expect	New	REJECT	Cannot replay, unknown (seqNum, reqId)
001	$<$ expect	Same	REPLAY	Replay prior CallRes for (seqNum, reqId)
010	$=$ expect	New	ACCEPT	Expected request, state change, new CallRes
011	$=$ expect	Same	REJECT	Invalid identity mapping
020	$>$ expect	New	REJECT	Future sequence number
021	$>$ expect	Same	REJECT	Invalid identity mapping

Figure 6: CallTracker request decision matrix for (caller, seqNum, reqId) reconciliation.

Column descriptions:

- **Case**: Factor-coded case ID (mixed-radix) over the cartesian product enumeration
- **New** or **Same**: Whether a (caller,seqNum,reqId) combo is the same as accepted previously
- **Sequence comparison**: The supplied `seqNum` is less than, equal to, or greater than the next expected value
- **Request mapping**: Each (`caller`, `reqId`) maps to one `seqNum`, and each (`caller`, `seqNum`) maps to one `reqId`
- **Advance rule**: Only an accepted new request advances the sequence; replay and rejection leave it unchanged

A `seqNum` must increase monotonically per caller (FIFO) for a write to occur otherwise it is a replay or error. Reuse never advances a `seqNum` as it is always either idempotent (replay) or an error.

12.6 Ambiguous-Outcome Recovery

A request producer treats submission as an intermediate state rather than proof of completion. When a response or receipt is lost, it reconciles the durable off-chain request record against the on-chain sequence and `CallTracker` result. A matching completed request is committed locally; partial work resumes from the recorded cursor; a request that is absent may be resubmitted with the same identity. Conflicting sequence or request identities stop automatic processing and require recovery logic or manual intervention.

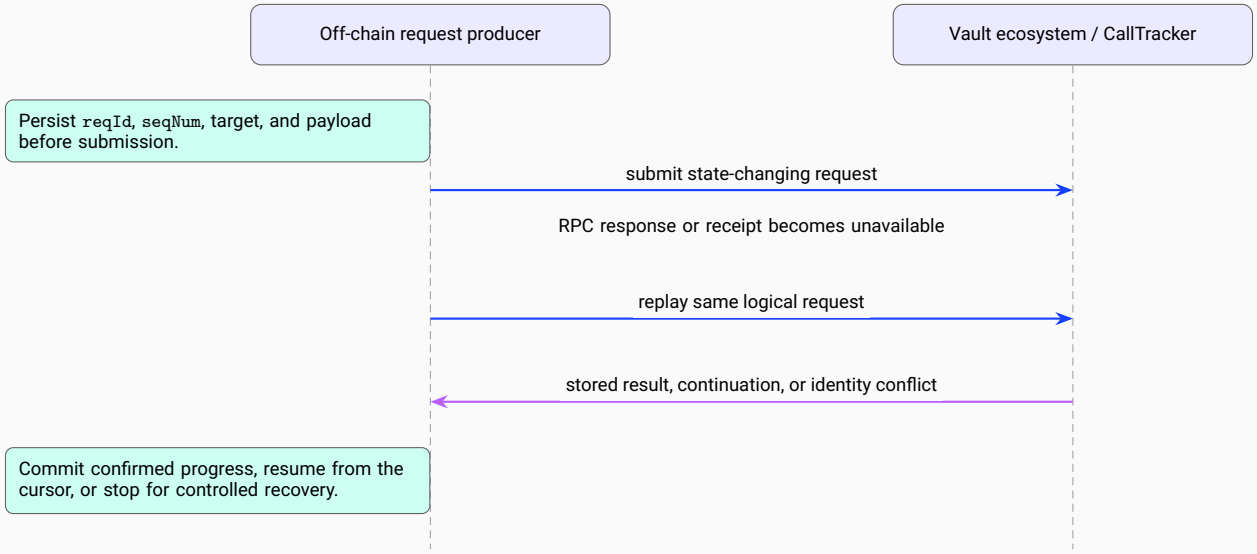

Figure 7: Recovery from an ambiguous transaction outcome without duplicating the logical effect.

13 Request Producers: Interface Contract

This section defines the normative contract for off-chain request producers (e.g. a Chain Sync service) that drives proposals. This control plane must be deterministic and reconciliation-first.

13.1 Queue Discipline

Request producers should ensure requests are executed in FIFO order:

- Dependent proposals should ordinarily be voted on only after prior proposals are terminal
- Exceptions may exist for unrelated proposals, but consistently FIFO simplifies correctness

Each off-chain request/proposal contains a UUID identifier (`reqId`) and every contract write call in the proposal workflow sequence has an identifier that is a deterministic descendant of the proposal's `reqId`.

Each queue must have one active request producer and one active on-chain Agent identity within a sequence domain, and:

- Throughput can be increased by segmenting unrelated requests into separate queues
- More than one agent per queue risks race conditions and replay errors
- A single agent can manage multiple queues, but this may not justify the increased complexity

13.2 Concurrency Model (Control-Plane)

A **thread** is defined as a FIFO-ordered stream of state-mutating calls that share a sequence domain. In the Vault Ecosystem, the primary sequence domain is **per contract, per caller**: each contract maintains a monotonic `seqNum` for each caller address.

Implication 1: Per-contract sequencing. A request producer must durably track `seqNum` independently for each contract it writes to (e.g. Vault, XferMgr, RevMgr, etc). Two calls with the same `caller` sent to different contracts occupy different sequence domains and therefore have independent ordering constraints. After a restart, durable off-chain state must be reconciled against authoritative on-chain state before new sequence numbers are allocated.

Implication 2: Single-writer per thread. For correctness and simplicity, each (contract, caller) thread should have exactly one active writer at a time. Multiple writers increase the probability of replay/sequence conflicts and can degrade operational determinism.

Implication 3: Parallelism via sharding. Throughput can be increased safely by sharding work across:

- distinct contracts (e.g. XferMgr vs InstRevMgr),
- distinct caller addresses (multiple Agent keys),
- distinct workflow queues whose calls do not share on-chain preconditions.

Practical Threading. Since proposals may depend on state in other proposals and each proposal supports high-throughput via paged payload uploads, in the near-term a single Agent thread will process all requests. This simplification eliminates dependency concerns in this layer and pushes the concern up the stack to the GigaLedger where it can be managed by application workflow constraints. However, if/when concurrency scaling warrants the complexity, the implications above can be used as a guide for increasing throughput via more threads (e.g. 1 Agent per proposal type).

Thread safety and recovery. When a producer restarts or loses receipt visibility, it must reconcile by querying chain state (headers, proposal status) and, when required, by replaying events from the `blockNum` surfaced by `CallTracker` for deterministic recovery.

13.3 Sequence Number Tracking

Each contract tracks a monotonically increasing sequence number (`seqNum`) per caller that is incremented after each accepted state-changing call. This `seqNum` enforces a FIFO ordering per caller.

13.4 Request Identity Tracking

Each request contains a UUID identifier (`reqId`) where tracking is **per contract**. If the key (`caller, seqNum, reqId`) matches a prior call then no state change occurs and the `CallRes` for the original call is replayed (it may be a summary and can be used to recover prior events).

This behavior enables idempotency in an off-chain client via requesting events from a prior block. Idempotency is intentionally sharded between on-chain and off-chain mechanisms to eliminate both the on-chain complexity and expense of storing historical event state in the contract which is implicitly tracked by the underlying ledger.

13.5 Off-Chain Database Commits

Off-chain clients must not mark a request as complete solely because a transaction was submitted. Progress must only be committed when the chain state confirms the related transactions after the configured confirmation policy is satisfied and relevant contract state has been reconciled.

14 Integration Contract and Public Observability

A short operational summary of the primary proposal workflows:

14.1 Instrument and Revenue Transfers

- **Create**: `CallRes` yields `pid`
- **Upload Items**: All pages uploaded; proposal header `uploadedAt` is non-zero
- **Seal**: Vault status becomes `Sealed`
- **Execute**: Poll execution calls until a terminal Vault status such as `Executed`

14.2 Instrument Revenues

- **Create**: `CallRes` yields `pid`
- **Upload InstRev**: All pages uploaded; proposal header `uploadedAt` is non-zero
- **Upload Owners**: Per-key snapshots complete and consistent with InstRev totals
- **Seal**: Vault status becomes `Sealed`
- **Execute**: Poll execution calls until a terminal Vault status such as `Executed`

14.3 Generated Contract API

To provide consistent abstractions and help ensure proper protocol sequences for on-chain and off-chain interactions, an off-chain API is generated from both the Solidity code and related ABI. This API ensures a dynamic type-safe binding between the EVM contracts and off-chain services that improves confidence in contract-interface alignment both during past development and going forward with future enhancements.

This generated API contains features such as deployment, upgrading, artifact tracking, telemetry, request and sequence identifier tracking, gas bumping, database-to-chain interop, reliable mechanisms, and log instrumentation for operational concerns.

14.4 Public Audit Surfaces

Public auditability is a first-class design requirement. Public actors can query the entire proposal lifecycle, inspect payload summaries, and independently verify outcomes using view functions and events. For a full interface reference see the published contract code as described in subsection 16.2.

14.4.1 Vault: Proposal Lifecycle and Governance

Representative public reads include:

- `getLastPropId`, `getProp`, `getPropStatus`
- `getQuorum`, `getVotes`
- Manager address getters (e.g. `getXferMgr`, `getRevMgr`)

14.4.2 XferMgr: Transfer Auditing

Public reads enable verification of:

- Proposal header (token info, flags, timestamps)
- Upload completeness and execution cursor progress
- Transfer lines via paged reads of full or lite views
- Failure counts and per-line statuses via lite views and events

14.4.3 InstRevMgr and RevMgr: Revenue Auditing

Public reads enable verification of:

- Instrument revenue lines keyed by (instrument, earn date)
- Ownership snapshots and per-owner allocations (paged)
- Corrections and their deltas
- Execution progress via headers and telemetry events

14.4.4 BalanceMgr and EarnDateMgr: Accounting and Enumeration

Public reads enable:

- Balances by token address or obfuscated owner identity
- Enumeration for audit navigation by instrument, earn date, or both

14.4.5 Event-based Telemetry

Events provide streaming observability for:

- Proposal lifecycle (created, sealed, executed, terminal outcomes)
- Upload progress (instrument revenue uploaded, owners uploaded, transfers uploaded)
- Execution progress (allocated, processed, errors)
- Idempotency signals (acknowledgment of first execution versus replay)

14.4.6 Explorer-Level Transparency Example

PolygonScan exposes verified source, decoded call inputs, token transfers, event logs, and address-linked transaction history for the deployed contracts. Figure 8 shows one successful Vault execution that transferred approximately 16,709 USDC across 71 ERC-20 transfer records in a single transaction[14]. The decoded input also exposes the execution sequence number, request identifier, and proposal identifier. Investors can follow recipient addresses and subsequent transfers directly through the public ledger from their wallet deposits.

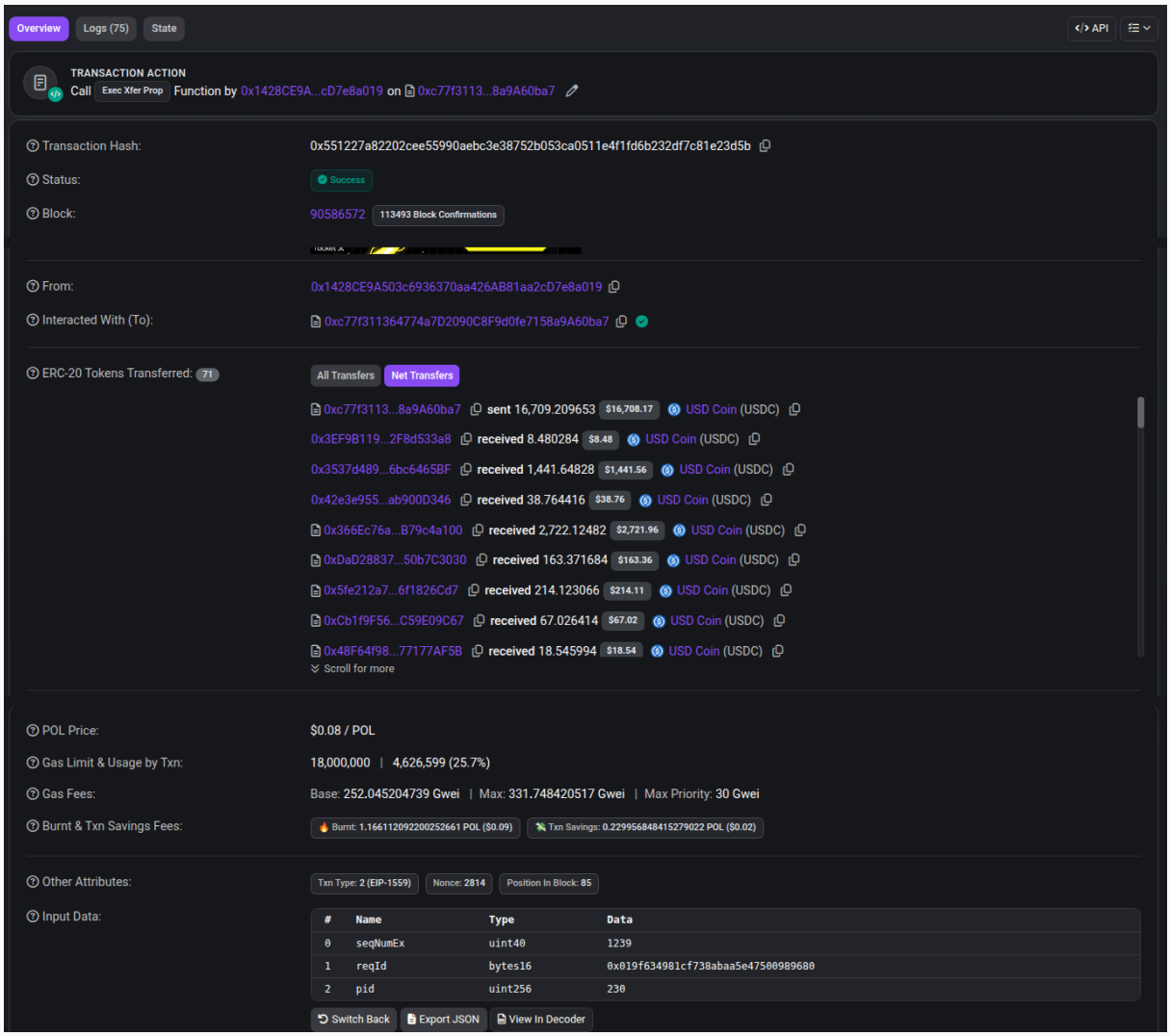

Figure 8: PolygonScan transaction view for an approximately 16,709 USDC Vault revenue transfer. The image is a cropped composite of the public transaction page; click the figure to open the transaction.

This figure demonstrates the transparency of an execution step for a transfer proposal. The figure links directly to the public PolygonScan transaction.

15 Security and Trust Model

This section defines the protocol authority model, how permissions are partitioned across roles and contracts, and the invariants intended to protect custody, governance integrity, and accounting correctness.

15.1 Roles

Creator / Admin The Creator/Admin role represents platform governance and operational authority with elevated privileges. It is intended for trusted operators subject to internal controls. Creator/Admin is responsible for system configuration, emergency actions, and upgrade authorization. The Creator role is meant to be a transient role used during initial deployment/setup and then released in preference of the Admin role.

Agent Agents are operational actors responsible for proposing and executing workflows. Agents initiate proposals, upload proposal data, and execute approved proposals. Agents are not intended to unilaterally redirect user value; they operate within protocol constraints and governance decisions related to compliance policies.

Voter Proposal execution is contingent on a quorum of voters who are governance actors. This quorum-gated model mitigates individual role compromise by requiring an attacker to control a quorum of voter accounts.

Contract Roles (Manager Contracts) In specific cases, contracts in the system are privileged by address to call specific internal accounting and custody endpoints. This facilitates modular design via feature delegation.

BoxMgr The BoxMgr contract has the Box owner role to control box-local administration and outbound asset movements from that box, subject to protocol constraints (e.g. pushes initiated by the system).

15.2 Authority Boundaries

The protocol separates responsibilities into three distinct layers:

1. **Governance decisioning (Voters)** decides whether actions are permitted
2. **Operational execution (Agents)** perform automation to create proposals and execute approved proposals
3. **Custody and accounting enforcement (Vault and Managers)** enforces on-chain invariants and ensures only authorized contracts can mutate balances and move assets

This separation is intended to reduce the risk that any single actor can both authorize and execute value-sensitive actions without oversight.

15.3 Permission Matrix

Permissions summarized by capability:

Capability	Creator/Admin	Agent	Voter	Vault	RevMgr	XferMgr	InstRevMgr	BoxMgr
Configure contract registry / depends	Yes	No	No	No	No	No	No	No
Authorize upgrades	Yes	No	No	No	No	No	No	No
Create proposals	Yes	Yes	No	No	No	No	No	No
Vote on proposals	No	No	Yes	No	No	No	No	No
Execute approved proposals	No	Yes	No	Yes	Yes	Yes	Yes	No
Upload proposal data	No	Yes	No	No	No	No	Yes	No
Operational mitigation (prune xfers)	No	Yes	No	No	No	Yes	No	No
Update balances / claim tracking	No	No	No	No	Yes	Yes	No	No
Move assets from Vault custody	No	No	No	Yes	No	Yes	No	No
Move assets from Box custody	No	No	No	Yes	No	No	Yes	Yes

Figure 9: Capability-level permission matrix

15.4 Key Security and Accounting Invariants

Custody invariants: Sensitive asset movement paths are routed through designated custody contracts and managers, limiting direct value movement to well-defined code paths. Operational mitigations (e.g. pruning transfers) are designed to ensure proposal progress without enabling redirection of value to arbitrary destinations.

Governance invariants: Proposal approval is determined by Voters; execution is performed by Agents only after proposals reach an approved state. Vote transitions are designed to remain internally consistent under the governance model, including when voter membership changes before a proposal reaches a terminal status.

Accounting invariants: Paging and batching logic is designed so proposal totals are not inflated by multi-page uploads. Balance claim and unclaim logic is bounded to prevent overflow conditions and preserve internal balance consistency. Token transfers are intended to rely on well-behaved assets consistent with protocol assumptions.

15.5 Operational Controls and Safety Valves

The protocol includes operational safety valves intended to reduce the impact of problematic inputs:

- **Paged uploads and execution:** Large data sets (e.g. many transfers or ownership entries) can be uploaded and processed incrementally, enabling resumability in the presence of gas limits or intermittent failures.
- **Transfer pruning/skip mechanism:** Agents can mark certain transfers as skipped to mitigate known-bad destinations or operational hazards. Skipping is designed as a non-redirection control; intended recipients remain payable through future proposals once the blocking issue is resolved.

It is often more practical to withdraw and recreate a proposal rather than use these escape-hatch features.

15.6 Implementation Alignment and Release Discipline

At the implementation level, state-changing entry points are role-gated through shared authorization checks and contract-to-contract permissions. Because function-level mappings evolve as the codebase develops, a versioned, release-specific control mapping is maintained for each tagged deployment. This enables reviewers and auditors to verify that the deployed bytecode corresponds to the documented role and trust model for that release.

15.7 Threat Model and Residual Risk

The public threat model focuses on protocol behavior and deliberately excludes sensitive operational key-management details. The principal protected properties are governance integrity, accounting integrity, authorized value movement, replay safety, upgrade integrity, and the availability of sufficient public evidence to reconstruct material workflows.

The architecture is designed to make integrity failures detectable and recoverable where possible, not to claim that all operational or external risks can be eliminated. Institutional controls, regulated-party procedures, monitoring, and incident response complement the on-chain controls but remain outside this publication's security-detail scope.

Scenario	Protocol response	Residual exposure / recovery
Compromised operational Agent	Role gating, quorum-dependent proposal states, bounded execution, and public telemetry limit unilateral action.	Authorized operational functions can still be abused; monitoring, revocation, and institutional controls remain necessary.
Compromised Voter or governance participant	Quorum requires multiple approvals; votes and proposal transitions are publicly observable.	A compromised quorum can authorize harmful actions. Governance membership and key controls remain external trust assumptions.
Compromised upgrade authority	Upgrade entry points are role-gated and deployed implementations are publicly verifiable.	Upgradeability is an explicit trust boundary; operational authorization and review controls are not fully disclosed in this paper.
Lost receipt, timeout, or process restart	CallTracker identity and sequence rules allow deterministic replay and on-chain and off-chain reconciliation.	Off-chain producers must follow the protocol and preserve durable request identity.
Partial batch execution or gas exhaustion	Monotonic cursors and bounded transactions permit continuation without restarting completed work.	Availability depends on future transaction inclusion and affordable gas.
RPC fault, stale response, or event-indexing gap	Authoritative contract state, transaction receipts, and historical logs can be queried from the ledger; providers can be changed.	A provider outage can delay progress; the protocol does not guarantee third-party availability.
Chain reorganization	Confirmation policy and sequence continuity detect or surface divergent assumptions before subsequent work is committed.	Deep or unusual reorganizations may require conditional recovery or manual intervention.
Malformed or non-conforming asset	Token-specific validation, safe transfer patterns, and proposal constraints reduce exposure.	External token contracts and stablecoin issuers remain third-party dependencies.

Figure 10: Public threat scenarios, protocol responses, and residual risk.

16 Deployment And Bootstrapping

16.1 Bootstrapping Concerns

Because contract logic is optimized for bounded execution and modularity, production readiness depends on explicit bootstrapping and operational controls:

- Correct wiring of contract addresses (as a mesh of services)
- Correct role assignments and rotation playbooks (Admin, Voter, Agent)
- Token approvals and operator approvals required for ERC-20[5] and ERC-1155[6] transfers
- Proper migration of token ownership from the legacy system
- Proper creator role revocation
- Monitoring for lifecycle events, execution progress, and replay signals

The technology is designed to support compliance through workflow auditability, separated authority, regulated-party participation, and explicit operational controls. GigaStar Technologies operates as the technology provider, while GigaStar Markets and GigaStar Securities perform regulated functions within their respective scopes[11,15,16]. Legal and compliance responsibility is not reduced to smart-contract behavior alone.

16.2 Deployed Contracts

Each contract has published and verified source code per proxy and logic contract. PolygonScan exposes verified source, ABI, transactions, events, read calls, write-call composition, and proxy implementation relationships for each deployment[17].

Contract	Address
BalanceMgr	0x0e8ccab66952cf92e57ec71448d02cf9a84739d3
BoxMgr	0xd1d9165f6f3a09592947845b7f77a245cf5543d2
Crt	0xfea6c7d87231b584efeeb4f11d86ee630e7663c9
EarnDateMgr	0xa3bcc98a11e70de17fcad70302b13049a7f05082
Erc20PtrMgr	0x0cf7d8f63116ff3775895605931d3f1aa351c4b7
InstRevMgr	0xac0abb5c5e7dd179b843b8cd5f529db30bfb24c7
RevMgr	0x6211ecf97c45e0c8a551df8f24c010b7c859b230
Vault	0xc77f311364774a7d2090c8f9d0fe7158a9a60ba7
XferMgr	0xbfbebf3564a317fe01ca66fde0c6837dfc916e8d

Figure 11: Polygon PoS mainnet contract addresses

As there are many instruments, only the primary contract addresses are provided here for brevity. The ERC-20 address per instrument can be derived from the above.

16.3 Sweeping

To facilitate both a system migration and instrument rolls, each instrument supports sweeping. Legacy instruments contain a legacy deposit address that is conditionally swept forward such that funds are migrated to a Box to prepare for a proposal. Decoupling allows agreements to be updated asynchronously while legacy addresses are retired.

Before an Instrument Revenue proposal, the set of related instruments is used as input for a depth-first traversal (DFS) of disjoint directed acyclic graphs (DAGs). Funds are swept conditionally based on whether an old address exists, whether an instrument is rolled, and whether an instrument that is a root node in the graph is in the proposal.

17 Production Evidence and Validation

Unless otherwise stated, the figures in this section are approximate operational totals as of July 2026. Counts continue to grow, and the cumulative total increases as monthly distribution amounts vary.

17.1 Operating Scale

Measure	Approximate value	Context
Investors supported	30,000	Platform accounts
Current instruments	67	Active instruments; count continues to grow
Deployed contracts and instrument wrappers	233	Primary contracts, Boxes, instrument wrappers
Proposals executed	300	Governance-controlled workflows
Instrument transfers processed	32,000	Issuance
Revenue transfers processed	420,000	Individual payments
Cumulative revenue distributed	$1.5 million	Cumulative total; monthly distributions vary
Large-distribution recipient count	3,000	Representative scale for one distribution workflow
Ambiguous outcomes reconciled	20	Approx. production cases with retry/reconciliation
Duplicate effects from retried requests	0 observed	Operational result under the CallTracker protocol

Figure 12: Production operating evidence as of July 2026.

Source: GigaStar internal operational records. Values are rounded and continue to change with platform activity.

The legacy platform entered production in May 2023. The Vault Ecosystem entered production in March 2026. Blockchain-adjacent portions of a typical approved workflow ordinarily complete in approximately five to ten minutes. End-to-end business workflows may take longer because they can include multi-party reconciliation and regulated operational activity. Worst-case completion time is intentionally not bounded because blockchain maintenance, congestion, gas markets, stablecoin infrastructure, and node-provider availability are external dependencies.

17.2 Engineering Validation

The v1.5 release uses Solidity 0.8.34, Foundry 1.15.0, and Slither 0.11.5. The release validation record includes:

- 112 passing unit tests across 22 suites, with no failed or skipped tests in the reported run;
- 98% aggregate unit-test coverage;
- system-integration tests that execute representative workflows against a local Forge blockchain;
- Slither static analysis in addition to compiler diagnostics and code review;
- a custom post-build storage-layout tool that highlights upgrade compatibility changes;
- source verification on PolygonScan for proxies and current implementations; and
- production reconciliation evidence, including approximately 20 ambiguous transaction outcomes with no observed duplicate logical effects from retried requests.

The test and coverage figures describe the referenced release and are not a substitute for independent security assessment. Likewise, successful production operation is evidence of exercised behavior, not proof that every possible failure mode has been eliminated.

17.3 Independent Security Assessment

CertiK completed an independent security assessment of the GigaStar smart-contract system on April 16, 2026[18,19]. The assessment covered Solidity contracts and used manual review and static analysis. The public final report records no Critical or Major findings and documents the reviewed code revisions, remediation work, accepted design tradeoffs, and complete dispositions for lower-severity findings. Readers seeking details should consult the public report. At publication, CertiK's SkyNet dashboard assigned the project an A grade.

18 Limitations and External Dependencies

The Vault Ecosystem intentionally combines on-chain guarantees with off-chain regulated and operational processes. Its principal limitations and dependencies include:

- Off-chain request producers follow the `CallTracker` protocol.
- Governance, Agent, and upgrade roles remain explicit trust boundaries even though their actions are constrained and observable.
- The architecture depends on Polygon PoS, RPC providers, block explorers, stablecoins, and conforming token contracts for availability or external behavior.
- A configured confirmation policy reduces reorganization risk but cannot make the underlying network mathematically final at submission time.
- Compliance eligibility, including Regulation Crowdfunding resale restrictions, depends on off-chain controls and regulated-party procedures.
- Public source verification and telemetry improve transparency but do not themselves guarantee that an implementation is defect-free.

19 Conclusion

The GigaStar Vault Ecosystem treats regulated digital-asset operations as a distributed-systems correctness problem, not merely as token issuance. Its proposal model separates authorization from execution; paging and cursors make large workflows bounded and resumable; CallTracker converts uncertain off-chain transaction outcomes into an explicit, replay-safe protocol; and public read surfaces make workflow states publicly accessible.

The architecture has progressed beyond a design proposal: it operates in production, processes recurring revenue and instrument transfers at material scale, has reconciled ambiguous transaction outcomes without observed duplicate effects, exposes verified deployed source, and has undergone independent security assessment. These properties provide a foundation for continued integration with regulated partners while retaining the transparency and deterministic execution benefits of a public EVM network.

20 Contributions and Acknowledgments

Jason Aubrey designed and implemented the principal systems in this whitepaper end-to-end, including:

- the original GigaStar smart-contract system;
- the Vault smart-contract ecosystem;
- the blockchain integration, transaction submission, reconciliation, and recovery services; and
- the associated off-chain persistence architecture and database model.

The GigaLedger web frontend was principally implemented by Byron Moore, with backend development and technical guidance provided by Victor Glava. Additional contributors developed and maintained the web application server and other platform components outside the primary scope of this whitepaper.

The author thanks the GigaStar team and reviewers who provided operational, legal, compliance, product, and technical feedback; the independent security assessors who reviewed the code and documented their findings; and especially his wife for her patience and support throughout the project.

A Contract Runtime Bytecode Size

The production contract implementations vary substantially in size, reflecting the functional decomposition of the Vault Ecosystem. Several implementations consume a significant portion of the EVM runtime bytecode limit, with the `Vault` implementation using more than 96% of the available capacity.

Contract	Runtime Size (B)	% Used	Size Unused (B)
`Vault`	23,669	96.3%	907
`BoxMgr`	20,007	81.4%	4,569
`InstRevMgr`	19,406	79.0%	5,170
`Crt`	16,409	66.8%	8,167
`RevMgr`	15,619	63.6%	8,957
`XferMgr`	15,241	62.0%	9,335
`EarnDateMgr`	8,590	35.0%	15,986
`Box`	7,701	31.3%	16,875
`Erc20PtrMgr`	7,097	28.9%	17,479
`BalanceMgr`	6,697	27.3%	17,879
`Erc20Ptr`	2,058	8.4%	22,518

Figure 13: Runtime bytecode size of production contract implementations exceeding 2 KiB.

Note: Deployment helpers, proxy shells, libraries, and test contracts are excluded from Figure 13. Sizes were generated from the Solidity 0.8.34 production build with optimization enabled for 200 runs. The EVM runtime bytecode limit is currently 24,576 bytes.

References

[1] "What You Need to Know: Native USDC on Polygon PoS," Circle. (2023), [Online]. Available: https://www.circle.com/blog/what-you-need-to-know-native-usdc-on-polygon-pos (visited on 2026-07-22).

[2] "Ethereum Virtual Machine (EVM)," Ethereum.org. (2025), [Online]. Available: https://ethereum.org/developers/docs/evm/ (visited on 2026-01-01).

[3] "Polygon PoS," Polygon. (2025), [Online]. Available: https://polygon.technology/polygon-pos (visited on 2026-01-01).

[4] "ERC-721 Non-Fungible Token Standard," Ethereum.org. (2025), [Online]. Available: https://ethereum.org/developers/docs/standards/tokens/erc-721/ (visited on 2026-01-01).

[5] "ERC-20 Token Standard," Ethereum.org. (2025), [Online]. Available: https://ethereum.org/developers/docs/standards/tokens/erc-20/ (visited on 2026-01-01).

[6] "ERC-1155 Multi-Token Standard," Ethereum.org. (2025), [Online]. Available: https://ethereum.org/developers/docs/standards/tokens/erc-1155/ (visited on 2026-01-01).

[7] "EIP-170: Contract Code Size Limit," Ethereum Improvement Proposals. (2016), [Online]. Available: https://eips.ethereum.org/EIPS/eip-170 (visited on 2026-01-07).

[8] J. Postel, "RFC 793: Transmission Control Protocol," IETF, Tech. Rep. 793, 1981. [Online]. Available: https://www.ietf.org/rfc/rfc793.txt (visited on 2026-01-01).

[9] W. M. Eddy, "RFC 9293: Transmission Control Protocol (TCP)," IETF, Tech. Rep. 9293, 2022. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc9293 (visited on 2026-01-01).

[10] "Regulation Crowdfunding: Guidance for Issuers," U.S. Securities and Exchange Commission. (2024), [Online]. Available: https://www.sec.gov/resources-small-businesses/small-business-compliance-guides/regulation-crowdfunding-guidance-issuers (visited on 2026-07-22).

[11] "Funding Portals We Regulate," FINRA. (2025), [Online]. Available: https://www.finra.org/about/entities-we-regulate/funding-portals-we-regulate (visited on 2026-01-01).

[12] S. Bradner, "RFC 2119: Key words for use in RFCs to Indicate Requirement Levels," Harvard, Tech. Rep. 2119, 1997. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2119 (visited on 2026-01-01).

[13] "EIP-712: Ethereum Typed Structured Data Hashing and Signing," Ethereum Improvement Proposals. (2017), [Online]. Available: https://eips.ethereum.org/EIPS/eip-712 (visited on 2026-01-01).

[14] "Vault Revenue Transfer Transaction." Public example of a Vault transaction producing multiple USDC transfers, PolygonScan. (2026), [Online]. Available: https://polygonscan.com/tx/0x551227a82202cee55990aebc3e38752b053ca0511e4f1fd6b232df7c81e23d5b (visited on 2026-07-22).

[15] "BrokerCheck Report: Firm Summary (CRD 328312)," FINRA BrokerCheck, Report, 2025, Firm report PDF. [Online]. Available: https://files.brokercheck.finra.org/firm/firm_328312.pdf (visited on 2026-01-01).

[16] "Customer Relationship Summary (CRS) (CRD 328312)," FINRA BrokerCheck, Report, 2025. [Online]. Available: https://files.brokercheck.finra.org/crs_328312.pdf (visited on 2026-01-01).

[17] "GigaStar Vault Proxy on Polygon PoS." Transactions, verified source, ABI, events, and proxy implementation relationship, PolygonScan. (2026), [Online]. Available: https://polygonscan.com/address/0xc77f311364774a7d2090c8f9d0fe7158a9a60ba7 (visited on 2026-07-22).

[18] "GigaStar - Audit: Security Assessment," CertiK, Apr. 16, 2026, Public final report available through the GigaStar CertiK Skynet project page. [Online]. Available: https://skynet.certik.com/projects/gigastar (visited on 2026-07-22).

[19] "GigaStar - CertiK Skynet Project Insight," CertiK. (2026), [Online]. Available: https://skynet.certik.com/projects/gigastar (visited on 2026-07-22).

Revision History

Version	Date	Author	Status	Pgs/Secs	Summary
`v1.0-internal`	2026-01-01	Jason Aubrey	RFC	19/23	Initial draft
`v1.0`	2026-01-06	Jason Aubrey	Public	19/23	Minor revisions, same length
`v1.1`	2026-01-07	Jason Aubrey	Public	20/23	Added to sections: 4, 19, 22
`v1.2`	2026-01-13	Jason Aubrey	Public	20/23	Updated section 19.2
`v1.3`	2026-02-02	Jason Aubrey	Public	20/23	Updated section 12.4 table
`v1.4`	2026-04-27	Jason Aubrey	Public	20/23	Addresses, ERC-20, sweeping
`v1.5`	2026-07-24	Jason Aubrey	Public	27/23	Security assessment, production evidence, threat model, and additional diagrams