Independent Submission
Internet-Draft
Intended status: Experimental
Expires: 1 September 2026

A. Cowles
Quox Ltd

28 February 2026

Verifiable Operations Ledger and Trace (VOLT) Protocol
draft-cowles-volt-00

Abstract

The Verifiable Operations Ledger and Trace (VOLT) protocol defines a minimal, interoperable format for producing tamper-evident execution traces for agentic AI workflows. VOLT records are linked via SHA-256 hash chains and packaged into portable Evidence Bundles that can be verified independently by any conformant implementation.

VOLT functions as a "flight recorder" for AI agent operations: it captures the sequence of events -- messages received, policy decisions evaluated, human approvals granted, tools invoked, and results returned -- with cryptographic integrity guarantees that detect post-hoc modification, deletion, or insertion of records.

The protocol is privacy-first by design. Events carry metadata and content-addressed references rather than raw secrets or sensitive payloads. Evidence Bundles support explicit redaction, optional Ed25519 signatures for non-repudiation, and both rolling and final snapshot modes for long-running workflows.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 September 2026.

Table of Contents

1. Introduction

Autonomous and semi-autonomous AI agents are increasingly deployed to perform consequential operations: modifying production infrastructure, executing financial transactions, managing sensitive data, and orchestrating multi-step workflows across organizational boundaries. The operational logs produced by these systems are typically mutable, platform-locked, and lack cryptographic integrity guarantees. When an incident occurs -- an unauthorized change, a policy violation, or an unexpected outcome -- operators cannot reliably determine whether the recorded trace accurately reflects what happened.

VOLT addresses this gap by defining a lightweight protocol for producing execution traces where each event is cryptographically linked to its predecessor via SHA-256 hash chaining. This creates an append-only integrity chain: any modification, deletion, or insertion of events after the fact is detectable through recomputation of hashes. Events are packaged into self-describing, portable Evidence Bundles that include a manifest, the event chain, content-addressed attachments, and optional digital signatures.

VOLT is designed as a companion protocol to the Agent Envelope Exchange (AEE) [AEE] messaging format and the Agent Orchestration Control Layers (AOCL) [AOCL] governance framework. Together, these three protocols provide a layered architecture for agentic systems: AEE handles message transport, AOCL enforces policies and approval gates, and VOLT records a tamper-evident audit trail of everything that occurred.

It is important to note that VOLT provides tamper evidence, not "truth." If the execution host is fully compromised, an attacker controlling the recorder can emit a consistent but fabricated trace. VOLT detects post-hoc tampering of recorded traces; it does not guarantee that the recorder was honest at the time of recording. Optional signatures and planned future attestation mechanisms strengthen the non-repudiation properties, but the fundamental trust anchor remains the integrity of the recording environment.

This document specifies VOLT version 0.1, covering event recording, hash chaining, Evidence Bundle packaging, verification, privacy and redaction rules, and conformance levels. Features such as deterministic replay, a trace query language, remote hardware attestation, and blockchain anchoring are explicitly out of scope for this version and are noted as future work.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

Run A single end-to-end execution instance, spanning from a user request (or system trigger) through to a terminal outcome. Each run is identified by a unique run_id.

Trace The ordered sequence of events for a single run, forming the complete audit record of everything that occurred.

Event One atomic record describing something that happened during a run: a message received, a policy decision, a tool invocation, a human approval, a model response, or any other observable action.

Ledger Chain The sequence of events linked by prev_hash references, forming an append-only integrity chain. Each event’s hash covers the event content, and each event’s prev_hash points to the hash of the immediately preceding event.

Evidence Bundle A portable, self-describing package containing the trace (as NDJSON), a manifest, optional content-addressed attachments, optional signatures, and optional redaction metadata. Bundles are designed for audit, compliance, and incident reconstruction.

Attachment A content-addressed blob referenced by events via SHA-256 hash. Attachments hold tool outputs, generated artifacts, or other data that is too large or too detailed to embed in event payloads.

Commitment A cryptographic hash representing content: an event hash, an attachment hash, or a run root hash. Commitments enable integrity verification without requiring access to the original content.

Attestation A digital signature over one or more commitments, asserting that a particular actor or runner observed or executed the attested operations. Attestations are optional in VOLT v0.1 but the schema reserves fields for them.

4. Design Constraints

4.1. Privacy by Default

VOLT events MUST NOT include secrets such as API keys, passwords, raw tokens, or private keys. Events SHOULD store metadata and content-addressed references (hashes) instead of raw content. Sensitive payload fields SHOULD be omitted or redacted at source. VOLT is an evidence protocol, and evidence that leaks secrets is a liability rather than an asset.

4.2. Minimal Schema with Extensible Evolution

Every event MUST include a volt_version field. Unknown fields MUST be ignored by verifiers to ensure forward compatibility. Breaking changes to the event schema MUST increment the volt_version value (e.g., from "0.1" to "0.2" for compatible additions, or to "1.0" for incompatible changes).

4.3. Tamper Evidence, Not "Truth"

VOLT guarantees that recorded traces are tamper-evident: any modification after the fact is detectable through hash verification. VOLT does not guarantee correctness of the recorded data if the host environment is fully compromised at the time of recording. The protocol detects changes to the record; it does not attest to the fidelity of what was recorded.

5. Transport and Encoding

VOLT events are stored as Newline-Delimited JSON (NDJSON): one complete JSON [RFC8259] object per line, separated by a single newline character (U+000A). The encoding MUST be UTF-8. Each line MUST be a syntactically complete JSON object; partial objects or multi-line pretty-printed JSON MUST NOT be used in the events file.

The events file is conventionally named events.ndjson, though an alternative filename MAY be specified in the Evidence Bundle manifest via the events_file field. Events within the file MUST be ordered by ascending seq value.

6. Canonicalization

To ensure consistent hashing across implementations, VOLT defines a Canonical JSON serialization. An implementation MUST produce a canonical byte representation of an event object as follows:

1. Serialize as JSON with UTF-8 encoding and no insignificant whitespace (no spaces after colons or commas, no newlines or indentation).

2. Object keys MUST be sorted lexicographically (byte-wise comparison of UTF-8 encoded key strings) at every nesting level.

3. Numbers MUST be represented without exponent notation where possible, and without a trailing ".0" when the value is integral. For example, the number 100 MUST be serialized as 100, not 1e2 or 100.0.

4. Strings MUST be Unicode NFC (Normalization Form Composed) normalized before serialization.

If an implementation’s language or standard library does not provide a canonical JSON serializer, the implementation MUST apply a deterministic key sort and the normalized serialization rules above to produce byte-identical output for identical input objects.

7. Cryptographic Primitives

7.1. Hash Algorithm

Event hashing and attachment hashing MUST use SHA-256 [RFC6234] in VOLT v0.1. Future versions MAY introduce algorithm agility; the hash_alg field in manifests and attachment references is reserved for this purpose.

7.2. Hash Encoding

Hash values MUST be encoded as lowercase hexadecimal strings of exactly 64 characters (representing the 256-bit SHA-256 digest). Implementations MAY display a prefixed form such as sha256:2cf24d... in user interfaces, but the stored value in event fields and manifest fields MUST be the pure 64-character hexadecimal string without prefix.

7.3. Signatures

Digital signatures are optional in VOLT v0.1 but the event and manifest schemas reserve fields for them. When signatures are used:

* The signature algorithm SHOULD be Ed25519 [RFC8032].

* Public key identifiers SHOULD be stable across sessions, using a DID (Decentralized Identifier) or a key fingerprint.

* Signatures MUST be computed over a clearly defined message structure, as specified in Section 13.4.

8. Identifiers and Time

8.1. Identifiers

The run_id field MUST be unique per run. The event_id field MUST be unique within a run. Identifiers SHOULD be UUIDv4 [RFC4122] or ULID (Universally Unique Lexicographically Sortable Identifier). Other globally unique identifier schemes MAY be used provided they satisfy the uniqueness requirements.

8.2. Timestamps

The ts field MUST be an ISO 8601 timestamp in UTC with the "Z" suffix. Millisecond precision is RECOMMENDED. For example: 2026-02-28T19:11:02.123Z. Implementations MUST NOT use local time zone offsets; all timestamps MUST be expressed in UTC.

9. Event Schema

All VOLT events are JSON objects. Every event MUST contain the following top-level fields:

Field	Type	Description
volt_version	string	Protocol version, e.g., "0.1"
event_id	string	Unique identifier for this event within the run
run_id	string	Unique identifier for the run
ts	string	ISO 8601 UTC timestamp with Z suffix
seq	integer	Monotonically increasing sequence number, starting at 1
event_type	string	Dotted path identifying the event kind
actor	object	Who caused or observed the event
context	object	Correlation and cross-protocol linkage
payload	object	Event-type-specific data (privacy-safe)
prev_hash	string	64-char hex; hash of preceding event (64 zeros for genesis)
hash	string	64-char hex; SHA-256 of canonical event without hash field

Table 1: Required Event Fields

9.1. Actor Object

The actor object describes who emitted the event. It has the following fields:

REQUIRED fields:

* actor_type (string): One of "agent", "human", "system", "tool", or "runner".

* actor_id (string): A stable identifier for the actor, such as an agent name, user ID, or system component identifier.

OPTIONAL fields:

* display_name (string): A human-readable name for display purposes.

* org_id (string): Organizational identifier.

* team_id (string): Team identifier within the organization.

* runner_id (string): Machine or host identity, when known, identifying the execution environment.

Example:

```
{
  "actor_type": "agent",
  "actor_id": "quox.agent.routeros",
  "display_name": "RouterOS Agent",
  "runner_id": "runner:vm-prod-01"
}
```

9.2. Context Object

The context object links the event to AEE/AOCL and other external systems, enabling cross-protocol correlation.

REQUIRED fields:

* correlation_id (string): A stable identifier spanning the run. Implementations SHOULD use the AEE correlation ID as the primary linkage value.

OPTIONAL fields:

* parent_event_id (string): For span-like linkage to a parent event within the same run.

* aee_envelope_id (string): AEE envelope identifier.

* aee_message_id (string): AEE message identifier.

* aocl_policy_id (string): AOCL policy identifier relevant to this event.

* aocl_decision_id (string): AOCL decision identifier for a specific policy evaluation.

* workspace_id (string): Workspace or tenant identifier.

* project_id (string): Project identifier.

* tags (array of strings): Freeform tags for categorization and filtering.

Example:

```
{
  "correlation_id": "aee-corr-01HZABC123",
  "aee_envelope_id": "aee-env-456",
  "aocl_policy_id": "policy.prod.write.requires_hitl",
  "tags": ["prod", "write", "hitl"]
}
```

9.3. Payload Object

The payload object carries event-type-specific data. It MUST be safe by default: implementations MUST include metadata (tool names, operation types, timing, status codes) rather than secrets or raw sensitive content. Attachments SHOULD be referenced by hash rather than embedded.

In VOLT v0.1, payloads are metadata-only. Examples of appropriate payload content include: field names present in the original message (payload_keys), token counts, tool names, exit codes, and duration measurements. Full-content payloads (raw message text, raw tool output) are deferred to future versions and will require opt-in redaction support to be conformant.

10. Event Hashing and Chaining

10.1. Hash Computation

The event hash is computed as:

```
hash = SHA-256( CanonicalJSON( EventWithoutHashField ) )
```

Where EventWithoutHashField is the complete event object with the hash field removed. All other fields, including prev_hash, are included in the hash input. The result is encoded as a 64-character lowercase hexadecimal string.

10.2. Genesis Event

The first event in a run (the genesis event) MUST have:

* seq equal to 1.

* prev_hash equal to 64 hexadecimal zeros: "0000000000000000000000000000000000000000000000000000000000000000".

10.3. Chain Rule

For all events where seq is greater than 1, the prev_hash field MUST equal the hash of the event with sequence number seq - 1. Verifiers MUST reject any trace where the chain rule is violated.

10.4. Chained Events Example

The following example shows three chained events forming a minimal trace. Hashes shown are illustrative abbreviated values; real implementations MUST use full 64-character hexadecimal SHA-256 digests.

```
{"volt_version":"0.1","event_id":"evt-001",
 "run_id":"run-abc-123",
 "ts":"2026-02-28T19:12:00.000Z","seq":1,
 "event_type":"run.started",
 "actor":{"actor_type":"system",
          "actor_id":"quox.core"},
 "context":{"correlation_id":"corr-xyz-789"},
 "payload":{"entrypoint":"api.chat",
            "mode":"orchestrated"},
 "prev_hash":"00000000...00000000",
 "hash":"a1b2c3d4...e5f60718"}

{"volt_version":"0.1","event_id":"evt-002",
 "run_id":"run-abc-123",
 "ts":"2026-02-28T19:12:00.050Z","seq":2,
 "event_type":"aee.envelope.received",
 "actor":{"actor_type":"system",
          "actor_id":"quox.aee.gateway"},
 "context":{"correlation_id":"corr-xyz-789",
            "aee_envelope_id":"aee-env-456"},
 "payload":{"channel":"web","size_bytes":1842},
 "prev_hash":"a1b2c3d4...e5f60718",
 "hash":"b2c3d4e5...f6071829"}

{"volt_version":"0.1","event_id":"evt-003",
 "run_id":"run-abc-123",
 "ts":"2026-02-28T19:12:01.000Z","seq":3,
 "event_type":"tool.call.executed",
 "actor":{"actor_type":"runner",
          "actor_id":"runner:vm-prod-01"},
 "context":{"correlation_id":"corr-xyz-789"},
 "payload":{"tool_name":"shell",
   "status":"success","duration_ms":812,
   "attachment_refs":[
     {"hash_alg":"sha256",
      "hash":"e3b0c442...7852b855",
      "content_type":"text/plain",
      "label":"stdout"}]},
 "prev_hash":"b2c3d4e5...f6071829",
 "hash":"c3d4e5f6...07182930"}
```


In this trace:

* Event 1 (genesis) has prev_hash set to 64 zeros and its hash computed from its own content.

* Event 2 has prev_hash equal to Event 1’s hash, establishing the chain link.

* Event 3 has prev_hash equal to Event 2's hash, continuing the chain. It also references an attachment by SHA-256 hash.

11. Standard Event Types

Implementations MAY introduce custom event types, but the following standard types are RECOMMENDED for interoperability. The event_type field MUST be a lowercase dotted string with at least two segments.

11.1. Run Lifecycle Events

Event Type	Description
run.started	Run initialization; first substantive event after genesis
run.completed	Run finished successfully
run.failed	Run terminated with an error
run.cancelled	Run was cancelled by a user or system

Table 2: Run Lifecycle Event Types

11.2. AEE Messaging Events

Event Type	Description
aee.envelope.received	An AEE envelope was received by the system
aee.envelope.sent	An AEE envelope was sent to a recipient
aee.message.parsed	An AEE message was successfully parsed
aee.message.rejected	An AEE message was rejected (validation failure)

Table 3: AEE Event Types

11.3. AOCL Policy and Control Events

Event Type	Description
aocl.policy.evaluated	A policy was evaluated against the current context
aocl.decision.approved	A policy evaluation resulted in approval
aocl.decision.denied	A policy evaluation resulted in denial
aocl.hitl.required	A policy mandated human-in-the-loop approval

Table 4: AOCL Event Types

11.4. Tool Execution Events

Event Type	Description
tool.call.requested	A tool invocation was requested by an agent
tool.call.executed	A tool invocation completed (success or failure)
tool.call.failed	A tool invocation failed with an error

Table 5: Tool Event Types

11.5. Human-in-the-Loop Events

Event Type	Description
hitl.requested	Human approval was requested
hitl.approved	A human approved the requested action
hitl.denied	A human denied the requested action

hitl.timed_out	A human approval request expired without response

Table 6: HITL Event Types

11.6. File and Network Events

Event Type	Description
file.read	A file was read (metadata only: path category, size)
file.write	A file was written (metadata only: path category, size)
net.request	A network request was made (metadata only: method, status, timing)

Table 7: File and Network Event Types

11.7. Model Activity Events

Event Type	Description
model.requested	An AI model inference was requested (metadata: model name, token count)
model.responded	An AI model returned a response (metadata: model name, tokens used)

Table 8: Model Event Types

11.8. Custom Event Types

Custom event types MAY be introduced by implementations. Custom types MUST NOT conflict with the standard prefixes (run, aee, aocl, tool, hitl, file, net, model) unless they are extending those namespaces in a compatible manner. Custom types SHOULD use a vendor namespace prefix, for example:

* quox.marketplace.offer.created

* vendorx.routeros.script.deployed

Verifiers MUST ignore unknown event_type values provided that all required fields are present and valid.

12. Attachments

Attachments are content-addressed blobs referenced by events. They hold data that is too large or too detailed for inline inclusion in event payloads, such as tool standard output, generated configuration files, or sanitized report artifacts.

12.1. Attachment Hashing

Attachment content MUST be hashed with SHA-256 over the raw bytes of the attachment file. The resulting hash is used both as the content address (filename) and as the integrity reference in event payloads.

12.2. Attachment References in Payloads

Events that refer to attachments MUST reference them by hash within the payload object using an attachment_refs array. Each entry in the array MUST include:

* hash_alg (string): The hash algorithm, "sha256" in v0.1.

* hash (string): The 64-character hexadecimal hash of the attachment content.

* content_type (string): The MIME type of the attachment (e.g., "text/plain", "application/json").

* label (string): A human-readable label (e.g., "stdout", "stderr", "config_backup").

Example:

```
{
  "tool_name": "shell",
  "status": "success",
  "duration_ms": 812,
  "attachment_refs": [
    {
      "hash_alg": "sha256",
      "hash": "e3b0c44298fc1c149afbf4c8996fb924...",
      "content_type": "text/plain",
      "label": "stdout"
    },
    {
      "hash_alg": "sha256",
      "hash": "7d865e959b2466918c9863afca942d0f...",
      "content_type": "text/plain",
      "label": "stderr"
    }
  ]
}
```

12.3. Storage Layout

Within an Evidence Bundle, attachments SHOULD be stored under a two-level directory structure using the first two characters of the hash as a prefix directory:

attachments/<first-two-hex-chars>/<full-hash>

For example, an attachment with hash e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855 would be stored at:

attachments/e3/e3b0c44298fc1c14...7852b855

This prefix-based layout prevents file system performance degradation when large numbers of attachments are present.

12.4. Attachment Content Guidance

Good candidates for attachments include: sanitized tool stdout/stderr, JSON tool results with secrets removed, generated artifacts (reports, configurations), and policy evaluation explanations.

The following MUST NOT be stored as attachments by default: raw prompts containing secrets, raw HTTP headers or cookies, raw database dumps, private keys or tokens, full file contents of sensitive system locations, and personal data unless explicitly required and approved by policy.

Implementations SHOULD support a configurable maximum attachment size and truncation with a "truncated" marker in the payload when content exceeds the limit.

13. Evidence Bundles

An Evidence Bundle is a self-contained, portable package that enables independent verification of a VOLT trace. Bundles are designed for audit and compliance evidence, incident reconstruction, workflow accountability, and cross-system portability.

13.1. Bundle Layout

A bundle may be represented as a directory on disk or as a ZIP archive. The contents MUST be identical regardless of container format. The recommended layout is:

```
<bundle_root>/
  manifest.json          # REQUIRED: bundle index and summary
  events.ndjson          # REQUIRED: the event chain

  attachments/           # OPTIONAL: content-addressed blobs
    ab/
      ab12...<hash>
    e3/
      e3b0...<hash>

  signatures/            # OPTIONAL: detached signature files
    sig-1.json

  redactions/            # OPTIONAL: redaction metadata
    redactions.json

  notes/                 # OPTIONAL: human-readable notes
    notes.md
```

13.2. Manifest Format

The manifest.json file MUST be a single JSON object encoded as UTF-8. It serves as the index and summary record for the bundle.

13.2.1. Required Manifest Fields

Field	Type	Description
volt_version	string	MUST equal the event volt_version, e.g., "0.1"
bundle_id	string	Unique identifier for this bundle (UUID or ULID)
run_id	string	The run this bundle covers; ties to event run_id
created_ts	string	ISO 8601 UTC timestamp of bundle creation
hash_alg	string	Hash algorithm; MUST be "sha256" in v0.1
events_file	string	Filename of the events file, typically "events.ndjson"
event_count	integer	Total number of events in the events file
first_event_hash	string	64-char hex hash of the first event (seq=1)
last_event_hash	string	64-char hex hash of the last event (seq=max)

Table 9: Required Manifest Fields

13.2.2. Recommended Manifest Fields

The following fields are RECOMMENDED for production bundles:

* bundle_mode (string): Either "rolling" or "final".

* cutoff_ts (string): For rolling bundles, the ISO 8601 UTC timestamp of the cutoff point.

* correlation_id (string): The AEE correlation ID for cross-protocol linkage.

* producer (object): Information about the system that created the bundle, with subfields name (string), version (string), and optionally build (string, e.g., a git SHA).

* integrations (object): Summary identifiers for AEE and AOCL integration, with optional aee and aocl subobjects.

* redactions_present (boolean): Whether any events in the bundle contain redacted fields.

* attachments_present (boolean): Whether the bundle includes attachment files.

* attachments (array): A summary of included attachments, where each entry contains hash_alg, hash, content_type, bytes (integer), and path (string).

* signatures (array): Signature records as defined in Section 13.4.

* notes (string): Free-text notes about the bundle.

13.2.3. Manifest Example

```
{
  "volt_version": "0.1",
  "bundle_id": "01HZBUNDLE001",
  "run_id": "01HZRUN001",
  "created_ts": "2026-02-28T19:15:00.000Z",
  "hash_alg": "sha256",
  "events_file": "events.ndjson",
  "event_count": 8,
  "first_event_hash":
    "a1b2c3d4...e5f60718",
  "last_event_hash":
    "c3d4e5f6...07182930",
  "bundle_mode": "final",
  "correlation_id": "aee-corr-01HZABC123",
  "producer": {
    "name": "quox-core",
    "version": "0.9.3",
    "build": "git:8f3a3b1"
  },
  "redactions_present": true,
  "attachments_present": true,
  "attachments": [
    {
      "hash_alg": "sha256",
      "hash":
        "e3b0c442...7852b855",
      "content_type": "text/plain",
      "bytes": 4832,
      "path":
        "attachments/e3/e3b0c442...b855"
    }
  ]
}
```


13.3. Rolling and Final Bundles

13.3.1. Rolling Bundles

Rolling bundles provide periodic evidence checkpoints for long-running workflows or to reduce data loss if a run crashes. A rolling bundle SHOULD set bundle_mode to "rolling", include a cutoff_ts timestamp, and set last_event_hash to the hash of the last included event at the cutoff point. Rolling bundles may be superseded by later rolling bundles or a final bundle.

13.3.2. Final Bundles

Final bundles represent the complete record of a run. A final bundle SHOULD set bundle_mode to "final" and include a terminal event (run.completed, run.failed, or run.cancelled) as the last event in the chain. The event chain SHOULD span from seq=1 through the terminal event.

13.4. Signature Records

VOLT v0.1 does not require signing, but defines a standard signature record format so that implementations can add signatures without breaking interoperability. Signatures may be included inline in manifest.json under the signatures array, or as individual files under the signatures/ directory.

Each signature record contains the following REQUIRED fields:

* sig_version (string): "0.1".
* sig_type (string): The signature algorithm; "ed25519" is RECOMMENDED.
* key_id (string): A stable identifier for the signing key (DID or key fingerprint).
* signed_ts (string): ISO 8601 UTC timestamp of signing.
* scope (string): "bundle" in v0.1.
* message (object): The data that was signed, containing run_id, bundle_id, hash_alg, first_event_hash, last_event_hash, and event_count.
* signature (string): Base64-encoded signature bytes.

Example signature record:

```
{
  "sig_version": "0.1",
  "sig_type": "ed25519",
  "key_id":
    "did:key:z6Mkn5Gq...LxXWxabc",
  "signed_ts": "2026-02-28T19:15:05.000Z",
  "scope": "bundle",
  "message": {
    "run_id": "01HZRUN001",
    "bundle_id": "01HZBUNDLE001",
    "hash_alg": "sha256",
    "first_event_hash":
      "a1b2c3d4...e5f60718",
    "last_event_hash":
      "c3d4e5f6...07182930",
    "event_count": 8
  },
  "signature": "MEUCIQDxAbcDefGhIjKlMnOpQrStUvWxYz0123456789..."
}
```


A valid signature indicates that the signer attests the bundle's event chain endpoints and count match the signed message. Signatures do not prove that the underlying host was uncompromised; they provide non-repudiation and stronger chain-of-custody evidence.

14. Verification

Verification is the core value proposition of VOLT: if an Evidence Bundle cannot be verified independently, it is not evidence. A successful verification confirms that the trace has integrity and the bundle is complete.

14.1. What Verification Guarantees

A successful VOLT verification confirms:

* Events are well-formed per this specification.
* Each event hash matches recomputed content.
* Each event links correctly to its predecessor via prev_hash.
* The manifest matches the bundle contents (count and endpoint hashes).
* Any referenced attachments exist and their hashes match.
* Any included signatures (if present) validate correctly.

14.2. What Verification Does Not Guarantee

Verification does not prove:

* The host or runner was uncompromised during execution.
* The tool results are "true" beyond what was recorded.
* The AI model’s reasoning was correct.
* The run complied with external laws or policies not recorded as events.

VOLT is a tamper-evidence and chain-of-custody protocol, not an oracle.

14.3. Verification Algorithm

The following algorithm is normative. A conformant verifier MUST implement all steps unless an optional skip flag is specified.

Step 0 -- Load Bundle: Locate and parse manifest.json as UTF-8 JSON. Validate that all required manifest fields exist per Section 13.2.1. If the manifest is missing or unparseable, the result is ERROR.

Step 1 -- Load Events: Read the events file specified by manifest.events_file (defaulting to events.ndjson). Parse as NDJSON: one JSON object per line. Collect events in file order. If any line is not valid JSON, the result is FAIL with reason INVALID_EVENT_JSON.

Step 2 -- Validate Event Ordering: Ensure events are ordered by seq ascending. In strict mode, FAIL if seq does not start at 1 (SEQ_GAP), if any gap exists (SEQ_GAP), if duplicates are found (SEQ_DUPLICATE), or if the sequence is non-monotonic (SEQ_NOT_MONOTONIC). In permissive mode, warn on gaps but still FAIL on duplicates or decreasing sequences.

Step 3 -- Validate Required Event Fields: For each event, verify that all required fields from Table 1 exist and have the correct types. Missing or invalid fields result in FAIL with reason EVENT_SCHEMA_INVALID.

Step 4 -- Validate Version Compatibility: The volt_version in manifest.json MUST match the volt_version in every event. A mismatch results in FAIL with reason VERSION_MISMATCH.

Step 5 -- Recompute and Validate Event Hashes: For each event: (a) create a copy of the event object with the hash field removed; (b) serialize using the Canonical JSON rules from Section 6; (c) compute the SHA-256 digest of the canonical bytes; (d) compare the computed hexadecimal digest to the stored hash. A mismatch results in FAIL with reason EVENT_HASH_MISMATCH.

Step 6 -- Validate the Chain: For the first event (seq=1), prev_hash MUST be 64 hexadecimal zeros. Failure results in FAIL with reason INVALID_GENESIS_PREV_HASH. For each subsequent event, prev_hash MUST equal the hash of the immediately preceding event. A mismatch results in FAIL with reason CHAIN_BROKEN.

Step 7 -- Validate Run ID Consistency: All events MUST have the same run_id as the manifest's run_id. A mismatch results in FAIL with reason RUN_ID_MISMATCH.

Step 8 -- Validate Manifest Counts and Endpoints: (a) Count events in the file; the count MUST equal manifest.event_count. (b) Confirm manifest.first_event_hash equals the hash of the first event. (c) Confirm manifest.last_event_hash equals the hash of the last event. Any mismatch results in FAIL with reason MANIFEST_MISMATCH.

Step 9 -- Validate Attachments (if enabled): For each event, locate any payload.attachment_refs entries. For each referenced attachment: (a) locate the file at attachments/<first2>/<hash>; (b) read the raw bytes; (c) compute SHA-256; (d) compare to the referenced hash. A missing file results in FAIL with reason ATTACHMENT_MISSING. A hash mismatch results in FAIL with reason ATTACHMENT_HASH_MISMATCH. If attachment verification is disabled via a flag, the verifier SHOULD set attachments_verified to false in the report and warn if attachment references exist.

Step 10 -- Validate Signatures (if present and enabled): For each signature record in the manifest or under signatures/: (a) validate the signature record schema; (b) reconstruct the message object as defined in Section 13.4; (c) verify the signature bytes using the declared algorithm (Ed25519 recommended); (d) confirm the scope is supported ("bundle" in v0.1). An invalid signature results in FAIL with reason SIGNATURE_INVALID. If signature verification is disabled, set signatures_verified to false in the report.

14.4. Verifier Report Format

A verifier MUST output a result. For interoperability, VOLT v0.1 RECOMMENDS a JSON report format.

PASS report example:

```
{
  "result": "PASS",
  "run_id": "01HZRUN001",
  "bundle_id": "01HZBUNDLE001",
  "volt_version": "0.1",
  "hash_alg": "sha256",
  "event_count": 128,
  "first_event_hash": "a1b2c3d4...90",
  "last_event_hash": "f0e1d2c3...08",
  "attachments_verified": true,
  "signatures_verified": false,
  "warnings": []
}
```

FAIL report example:

```
{
  "result": "FAIL",
  "reason": "EVENT_HASH_MISMATCH",
  "details": {
    "seq": 42,
    "event_id": "01HZ-EVT-042",
    "expected_hash": "a1b2c3d4e5f6...",
    "found_hash": "ffee0011aabb..."
  }
}
```

ERROR report example:

```
{
  "result": "ERROR",
  "reason": "MANIFEST_UNREADABLE",
  "details": {
    "message": "manifest.json missing or invalid JSON"
  }
}
```

14.5. Exit Codes

For command-line implementations, the following exit codes are RECOMMENDED:

Code	Meaning
0	PASS -- verification succeeded
1	FAIL -- verification detected integrity violations
2	ERROR -- invalid bundle format or I/O error

Table 10: Verifier Exit Codes

14.6. Standard Failure Reason Codes

Verifiers SHOULD use consistent reason codes to enable automated processing of verification results. The following codes are defined:

Code	Description
MANIFEST_MISSING	manifest.json not found in bundle
MANIFEST_UNREADABLE	manifest.json exists but cannot be parsed
MANIFEST_SCHEMA_INVALID	manifest.json missing required fields
EVENTS_FILE_MISSING	Events file referenced by manifest not found
INVALID_EVENT_JSON	A line in the events file is not valid JSON
EVENT_SCHEMA_INVALID	An event is missing required fields or has wrong types
VERSION_MISMATCH	Event volt_version does not match manifest
RUN_ID_MISMATCH	Event run_id does not match manifest run_id
SEQ_NOT_MONOTONIC	Event sequence numbers are not monotonically increasing
SEQ_DUPLICATE	Duplicate sequence number found

SEQ_GAP	Gap detected in sequence numbers (strict mode)
EVENT_HASH_MISMATCH	Recomputed event hash does not match stored hash
INVALID_GENESIS_PREV_HASH	First event prev_hash is not 64 hex zeros
CHAIN_BROKEN	Event prev_hash does not match preceding event hash
MANIFEST_MISMATCH	Manifest counts or endpoint hashes do not match
ATTACHMENT_MISSING	Referenced attachment file not found
ATTACHMENT_HASH_MISMATCH	Attachment file hash does not match reference
SIGNATURE_SCHEMA_INVALID	Signature record is malformed
SIGNATURE_INVALID	Signature does not verify
UNSUPPORTED_SIGNATURE_TYPE	Signature algorithm not supported by verifier

Table 11: Standard Failure Reason Codes

15. Conformance Levels

To keep implementations comparable, VOLT defines three conformance targets. An implementation may conform to one or more of these levels.

15.1. VOLT-R (Recorder)

An implementation is VOLT-R conformant if it:

* Emits events matching the schema defined in Section 9.

* Computes hash and prev_hash correctly per Section 10.

* Respects the privacy constraints defined in Section 4.

* Uses the Canonical JSON serialization defined in Section 6 for hash computation.

15.2. VOLT-B (Bundler)

An implementation is VOLT-B conformant if it:

* Produces a manifest.json with all required fields per Section 13.2.1.

* Writes events.ndjson ordered by ascending seq per Section 5.

* Stores attachments content-addressed per Section 12.

* Ensures that secrets are excluded from event payloads and attachments.

15.3. VOLT-V (Verifier)

An implementation is VOLT-V conformant if it:

* Implements the full verification algorithm defined in Section 14.3.

* Verifies both event hashes and chain integrity.

* Verifies attachments referenced by events.

* Produces a report per Section 14.4.

* Returns appropriate exit codes per Section 14.5.

16. Privacy and Redaction

VOLT is an evidence protocol. Evidence that captures secrets, personal data, or sensitive operational content by accident is a liability. This section defines the privacy-first logging rules and redaction strategies for VOLT.

16.1. Non-Negotiable Rules

VOLT events and attachments MUST NOT contain:

* API keys, access tokens, or session cookies.

* Passwords or password hashes.

* Private keys, seed phrases, or recovery codes.

* Database connection strings containing credentials.

* Full authorization headers (e.g., "Authorization: Bearer ...").

* Raw secrets from environment variables.

The rule of thumb: if it can grant access, it MUST NOT be recorded.

Events MUST default to metadata and references. Full outputs SHOULD be stored as attachments only when safe and useful. When in doubt, implementations MUST record a summary and content hash rather than raw content.

When anything is omitted or sanitized, events MUST indicate this explicitly via redaction flags such as payload.redacted, payload.inputs_redacted, or payload.outputs_redacted set to true. This prevents silent censorship and keeps audits honest.

16.2. Data Classification

Implementations SHOULD classify data into at least the following categories:

Level	Description
PUBLIC	Safe to store and export (rare for operational data)
INTERNAL	Safe within the organization; not for public export
SENSITIVE	Requires strict redaction controls before storage
SECRET	Must never be stored in VOLT events or attachments
PII	Personal data; may be subject to regulatory requirements

Table 12: Data Classification Levels

16.3. Secret Scanning

Implementations SHOULD include a guard that runs before events and attachments are persisted. The scanner SHOULD detect:

* Token-like strings (JWT patterns, API key prefixes such as sk-, ghp_, AKIA).

* Common credential keys (password=, api_key, secret).

* PEM blocks (-----BEGIN PRIVATE KEY-----).

* AWS access key patterns (AKIA...).

* Common bearer token and cookie patterns.

When a potential secret is detected, the implementation SHOULD strip the value, set payload.redacted to true, and optionally raise an AOCL policy alert. Attachments SHOULD be scanned and sanitized before storage; if an attachment cannot be safely sanitized, it SHOULD be omitted entirely.

16.4. Redaction Flags and Strategy

In VOLT v0.1, redaction is explicit and simple:

* Omit sensitive fields entirely from the payload.

* Replace with boolean flags: "inputs_redacted": true or "redacted": true.

* Optionally include a summary: "inputs_summary": "Write config file to prod".

Example redacted payload:

```
{
  "tool_name": "shell",
  "operation": "write_file",
  "target": "prod/nginx.conf",
  "inputs_redacted": true,
  "inputs_summary": "Write nginx configuration update"
}
```

16.5. Redaction Log

If an implementation performs redaction, it MAY include a redactions/redactions.json file in the Evidence Bundle. This file describes which events had fields redacted and the category of redaction. The format is:

```
{
  "volt_version": "0.1",
  "run_id": "01HZRUN001",
  "items": [
    {
      "event_id": "evt-006",
      "fields_removed": ["payload.inputs"],
      "reason": "secret"
    },
    {
      "event_id": "evt-009",
      "fields_removed": ["payload.response_body"],
      "reason": "pii"
    }
  ]
}
```

Note: VOLT v0.1 does not require cryptographic proof of redaction correctness. That is a planned enhancement for future versions.

16.6. Export Safety Rules

When exporting Evidence Bundles outside the originating system (e.g., as a ZIP for audit handoff):

* INTERNAL and SENSITIVE attachments SHOULD be removed unless explicitly included by policy.

* A summary report of removed items SHOULD be included.

* A verification report SHOULD be included if requested.

* Chain integrity MUST be preserved: events MUST NOT be rewritten during export, as this would invalidate hashes.

If content must be removed after the fact, the export SHOULD be labeled as a "redacted bundle" and MUST NOT be presented as the original full-fidelity record.

17. Threat Model

This section describes the threats VOLT is designed to mitigate, the threats it cannot fully mitigate, and recommended countermeasures. VOLT is an evidence integrity protocol providing tamper-evident traces and portable verification, not perfect truth in adversarial environments.

17.1. Threats VOLT Mitigates

17.1.1. T1 -- Post-Hoc Tampering of Events

Attack: An adversary modifies an event after the run completes (e.g., changing "denied" to "approved").

Mitigation: Event hash validation detects the modification (EVENT_HASH_MISMATCH).

Residual risk: If the adversary re-hashes the entire chain and no signatures are present, the fabricated chain will appear valid. Signatures eliminate this residual risk.

17.1.2. T2 -- Deleting Events from the Middle

Attack: An adversary removes a tool execution event to conceal activity.

Mitigation: The chain breaks (CHAIN_BROKEN) and/or a sequence gap is detected (SEQ_GAP).

Residual risk: Same as T1 if the adversary rebuilds the chain from the deletion point.

17.1.3. T3 -- Inserting Fake Events

Attack: An adversary inserts a fabricated hitl.approved event to manufacture consent.

Mitigation: The chain breaks at the insertion point unless the adversary re-hashes all subsequent events.

Residual risk: Without trusted signatures, a complete re-hash produces a valid fabricated chain.

17.1.4. T4 -- Attachment Substitution

Attack: An adversary replaces a tool output attachment with a sanitized or fake version.

Mitigation: Attachment hash validation detects the substitution (ATTACHMENT_HASH_MISMATCH).

Residual risk: If the adversary also modifies the referencing event and re-hashes the chain, see T1.

17.1.5. T5 -- Repudiation of Approvals

Attack: An actor claims "that approval wasn’t mine."

Mitigation: When actor identity and/or digital signatures are used, repudiation becomes significantly harder.

Residual risk: Without signatures or strong identity binding, VOLT v0.1 provides sequencing evidence but weaker non-repudiation.

17.1.6. T6 -- Silent Redaction / Deceptive Logging

Attack: A recorder omits sensitive tool actions silently without leaving any trace of their existence.

Mitigation: VOLT requires explicit redaction flags; undisclosed omissions remain a governance and detection problem.

Residual risk: If an event was never recorded, its absence cannot be proven by VOLT alone. AOCL policies should enforce required event types for specific operation classes.

17.2. Threats VOLT Does Not Fully Mitigate

17.2.1. T7 -- Fully Compromised Host or Runner

Attack: An attacker controls the runner and emits a clean but fabricated VOLT trace.

Why VOLT cannot solve this alone: The recorder runs in the compromised environment. A compromised host can produce any trace it wishes.

Recommended mitigations: Remote runner attestations (planned), secure enclaves or TPM-backed keys, cross-signing from both orchestrator and runner, a separate logging channel to an append-only store, and strong AOCL enforcement with network segmentation.

17.2.2. T8 -- Signing Key Theft

Attack: An attacker steals a signing key and can sign forged bundles.

Recommended mitigations: Store keys in HSM or TPM hardware where possible; use short-lived keys with regular rotation; maintain key revocation lists; use separate keys per environment (development, staging, production); consider multi-signature requirements for high-risk runs.

17.2.3. T9 -- Incomplete Evidence (Missing Coverage)

Attack: A tool executes without VOLT hooks, so the evidence is incomplete.

Recommended mitigations: Enforce that all tool calls pass through instrumented middleware; use AOCL policies to deny execution if the VOLT recorder is not active; implement CI checks requiring certain event types (e.g., production runs must include hitl.approved).

17.2.4. T10 -- Privacy Leakage via Evidence Bundles

Attack: Sensitive data leaks because it was inadvertently logged or exported without proper controls.

Recommended mitigations: Enable secret scanning before write; enforce strict export controls with role-based access; implement configurable retention with automatic deletion; provide "export-safe" bundle modes that strip sensitive attachments.

18. Security Considerations

This section discusses the security properties, limitations, and operational considerations of the VOLT protocol.

Hash Chain Integrity Guarantees and Limitations. The SHA-256 hash chain provides strong tamper evidence for recorded traces. Any modification to an event -- changing a field value, altering a timestamp, or modifying actor information -- changes the event's hash, which in turn invalidates the prev_hash of every subsequent event. This cascade effect means that tampering with any single event requires recomputing all subsequent hashes. However, without digital signatures, an adversary with write access to the complete bundle can recompute the entire chain and produce a new, internally consistent but fabricated trace. Signatures provide the essential binding between the chain and a trusted identity.

Genesis Event Trust Anchor. The genesis event (seq=1) uses a well-known prev_hash of 64 hexadecimal zeros. The integrity of the entire chain depends on the trustworthiness of this starting point. If an adversary can substitute the genesis event and recompute the chain, the verification will pass. Operators SHOULD treat the genesis event hash as a trust anchor and SHOULD distribute or record it through an out-of-band mechanism when strong assurance is required. Digital signatures over the bundle (covering first_event_hash) mitigate genesis substitution.

Compromised Host Scenario. VOLT's integrity guarantees assume that the recording host is not fully compromised at the time of recording. A compromised host can emit arbitrary events that form a valid chain. VOLT detects post-hoc tampering (changes made after the chain was recorded), but it cannot detect fabrication at the source. Deployments requiring stronger guarantees SHOULD use remote attestation, cross-signing between multiple independent components, TPM-backed signing keys, or a separate append-only logging channel that is not controlled by the same host.

Key Management for Signatures. When Ed25519 signatures are used, the signing keys become high-value targets. Key compromise allows an adversary to produce validly signed forged bundles. Implementations SHOULD store signing keys in hardware security modules (HSMs) or Trusted Platform Modules (TPMs) where available. Keys SHOULD be rotated regularly, and separate keys SHOULD be used for different environments (development, staging, production). Implementors SHOULD maintain a key revocation mechanism and SHOULD consider requiring multiple signatures from independent signers for high-risk production runs.

Privacy Leakage via Metadata. Even when raw content is properly excluded from events, metadata can leak sensitive information. Timestamps reveal activity patterns. Actor identifiers may expose organizational structure. Correlation IDs can be used to link otherwise separate activities. Tool names and operation types may reveal infrastructure details. Implementations SHOULD assess the sensitivity of metadata fields in their deployment context and apply appropriate access controls to Evidence Bundles. The Section 16.6 rules provide baseline guidance for cross-boundary transfers.

Bundle Export Safety. Evidence Bundles exported outside the originating organization carry operational intelligence. Even with secrets removed, the sequence of events, tool names, timing information, and actor identifiers provide substantial insight into operational procedures and infrastructure. Exports MUST be treated as sensitive artifacts. Role-based access controls, audit logging of export operations, and data loss prevention policies SHOULD be applied. Redacted bundles MUST be clearly labeled and MUST NOT be represented as complete records.

Replay and Preimage Resistance of SHA-256. VOLT relies on the collision resistance and preimage resistance of SHA-256 [RFC6234]. As of the time of writing, SHA-256 remains considered secure against practical attacks, with no known feasible collision or preimage attacks. The hash_alg field in manifests and the versioning mechanism in VOLT provide a migration path if SHA-256 is deprecated in the future. Implementations SHOULD monitor cryptographic algorithm recommendations from NIST [FIPS180-4] and be prepared to transition to stronger hash functions if required.

Non-Repudiation with Ed25519. Ed25519 [RFC8032] signatures over bundle commitments (covering run_id, bundle_id, first_event_hash, last_event_hash, and event_count) provide non-repudiation: a signer cannot deny having attested to a specific chain state without claiming key compromise. The strength of non-repudiation depends on the key management practices described above. In VOLT v0.1, signatures are optional; deployments requiring strong non-repudiation MUST enable signing and MUST implement robust key management. Future versions may introduce per-event signatures and multi-party attestation for stronger guarantees.

Denial-of-Service Considerations. VOLT verification requires reading and hashing every event and attachment in a bundle. Maliciously crafted bundles with extremely large numbers of events, very large attachments, or deeply nested JSON structures could be used to consume excessive computational resources on a verifier. Implementations SHOULD impose configurable limits on event count, individual event size, attachment size, and total bundle size. Verifiers SHOULD report ERROR rather than attempting to process bundles that exceed configured limits.

19. IANA Considerations

This document has no IANA actions.

20. References

20.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

20.2. Informative References

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[AEE] Cowles, A., "Agent Envelope Exchange (AEE)", Work in Progress, Internet-Draft, draft-cowles-aee-00, 2026, <https://github.com/AdaminX/AEE-Agent-Envelope-Exchange>.

[AOCL] Cowles, A., "Agent Orchestration Control Layers (AOCL)", Work in Progress, Internet-Draft, draft-cowles-aocl-00, 2026, <https://github.com/AdaminX/AOCL-Agent-Orchestration-Control-Layers-Protocol>.

[FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", FIPS 180-4, August 2015, <https://csrc.nist.gov/publications/detail/fips/180/4/final>.

Acknowledgements

The author wishes to thank the early adopters and reviewers of the VOLT specification within the Quox ecosystem, whose feedback on real-world cryptographic evidence requirements for AI agent operations shaped the design of this protocol.

Author’s Address

Adam Cowles
Quox Ltd
London
United Kingdom
Email: adam@quox.ai
URI: https://quox.ai