TECHNICAL DOCUMENT 001 · PUBLIC METHODOLOGY

A Forensic Specification for AI Agent *Interaction Records*

Threat model, integrity model, and evidence format · v1.1

RELEASED	2026-05-25
DOCUMENT VERSION	v1.1 (release candidate)
SECTIONS	11 total · 9 substantively drafted · 2 forthcoming
ALIGNED TO	build plan v3.1.5 §9
LICENSE	CC-BY 4.0
CHANGELOG	/docs/methodology-changelog.md

Release candidate. This document publishes the methodology infrastructure — section structure, generator pipeline, design-system-conformant typography, and the substantively-drafted Policy Evaluation section (§5). Sections labeled *(forthcoming)* in the table of contents have framing outlines but await substantive drafting against the operator's manual quarterly review cadence (Stanford CIS clinic engagements through full external retainer in Year 2+).

The methodology is intended to be read by forensic experts, regulators, and defense counsel — not marketers. If you cannot defend the methodology, you cannot defend the evidence. Substantive readers are invited to surface holes in §5, the only section currently complete; the remainder will be reviewable as it is drafted.

DOCUMENT OF RECORD · TABLE OF CONTENTS

Contents

§ 1	Threat Model	*substantive*
§ 2	Integrity Model	*substantive*
§ 3	Evidence Format Specification	*substantive*
§ 4	Verification Procedure	*substantive*
§ 5	Policy Evaluation Methodology	*substantive*
§ 6	Detection and Response	*substantive*
§ 7	Retention and Legal Hold	*substantive*
§ 8	Audio Handling	*substantive*
§ 9	Limitations and Honest Disclosures	*substantive*
§ 10	Quarterly Review Log	*(forthcoming)*
§ 11	Worked Example	*(forthcoming v1.2)*

1. Threat Model

This section names the adversaries NuWyre defends against, the capabilities each is assumed to possess, the threats considered in-scope, the threats explicitly out-of-scope, and the trust assumptions on which integrity claims rest. A reviewer who finds the list incomplete should treat NuWyre's evidentiary claims with correspondingly bounded confidence.

NuWyre is **tamper-evident, not tamper-proof.** The distinction is load-bearing for the rest of this document and is restated where each detection mechanism is described.

1.1 Scope statement

NuWyre is the evidence layer for AI agent interactions. Its evidentiary contract is over **what the NuWyre server observed at ingestion time** — not over what an upstream system (the customer's AI agent, a third-party voice platform, an LLM provider) actually did. This boundary determines what is in and out of scope.

In scope

- Tampering with stored events after ingestion: insertion of fabricated events, modification of stored content, deletion of inconvenient events, reordering of the per-organization chain.
- Tampering with evidence bundles after export: byte-level mutation of any artifact in the ZIP, replay of a stale bundle as if it were current, substitution of an attacker-controlled bundle for an authentic one.
- Repudiation defenses: the customer's AI agent claiming "we never said that" about a specific exchange the NuWyre chain recorded; the regulator asking "prove this is the actual record."
- Insider tampering by a NuWyre operator with database write access: detection — not prevention — via the daily Merkle root anchored to the Bitcoin blockchain + mirrored to a public GitHub repository.
- Loss of clock authority: skewed or forged server clocks; adapter- supplied timestamps used to misrepresent ingestion timing.
- Loss or compromise of the ingestion signing key: detection + bounded-blast-radius procedures + rotation methodology.
- Denial-of-receipt: events sent by the customer's adapter for which no NuWyre record is produced; chain-of-custody for ingestion acknowledgements.
- Cryptographic anchor compromise: a single OpenTimestamps calendar server, a single RFC 3161 TSA, or the GitHub anchor repository being compromised. The defense is multi-leg anchoring; the threat model assumes any one anchor may fall.

Explicitly out of scope (V1)

- **Confidentiality of event content beyond at-rest encryption.** V1 is an evidence platform, not an end-to-end encrypted communications system. Event content lives in NuWyre's Postgres database at rest encrypted by Supabase's underlying storage layer; in transit over TLS 1.3. Customer-managed encryption keys land in larger custom-quote engagements where the customer's threat model justifies the operational complexity (operator manual §4).
- **Real-time tamper detection.** Daily Merkle roots provide bounded detection latency — typically within 24 hours of any tamper event, always within the next-daily-root window. A streaming-anchor mode (per-hour or per-event anchoring) is V1.1+ scope. Customers requiring sub-second tamper detection should not rely on V1 as their sole integrity layer.

- **Customer-side conversation fabrication.** A customer staff member who fabricates a conversation before sending it to NuWyre receives a NuWyre signature attesting that NuWyre observed the fabricated bytes at ingestion. NuWyre's integrity claim is "this is what we received and when," not "this is what actually happened in the world." Customer-side integrity is the customer's responsibility; customer-side trust artifacts (call recording chain-of-custody, agent attestation logs, deployment-time audit trails) are complementary to but distinct from NuWyre's evidence.
- **Adversaries with persistent root access to NuWyre infrastructure who can also forge our public GitHub anchor commits AND simultaneously compromise the Bitcoin chain at the relevant block-height.** Defense against the complete-infrastructure-takeover threat model is not claimed. NuWyre defends against threats that can be detected via cross-leg anchor verification; an adversary who simultaneously controls all anchor surfaces produces an undetected tamper. The multi-leg design raises the cost of this attack to nation-state level; it does not zero it.
- **Quantum-resistant cryptographic guarantees.** V1 uses SHA-256 and Ed25519, both vulnerable to a sufficiently capable quantum adversary. NuWyre takes no position on the timeline for practical quantum cryptanalysis; when post-quantum signatures and hashes become NIST-standardized and library-supported, an `event-v2` schema will adopt them. Bundles signed under v1 remain verifiable against the v1 contract; the migration story is "new keys + new schema for new events" not "rewrite history."

1.2 Trust assumptions

Each assumption below is necessary for NuWyre's integrity claims to hold. Where an assumption can be independently verified by the relying party (the regulator, the auditor, the opposing counsel), that verification path is named.

#	ASSUMPTION	INDEPENDENTLY VERIFIABLE?
TA-1	The Bitcoin blockchain provides timestamping resistant to economically-rational adversaries (proof-of-work cost > value of forging a backdated block at the relevant block-height).	Yes — verifier queries the public Bitcoin chain via the operator's chosen node.
TA-2	At least two of three pinned RFC 3161 TSAs (FreeTSA, Sectigo, DigiCert) remain operational + non-colluding for the duration of the relevant retention period.	Yes — verifier validates each TSA's certificate chain against pinned trust roots.
TA-3	The public GitHub anchor repository (NuWyre/anchors) or its Codeberg mirror is independently fetchable by the verifier.	Yes — verifier fetches commits directly; does not trust NuWyre's representation.
TA-4	The NuWyre Ed25519 issuer public key is obtained via a trusted out-of-band channel (the CLI binary's pinned key, this methodology PDF, or a TLS-authenticated fetch from `nuwyre.com`).	Yes — pinned in the verifier binary; key fingerprint published in this document; verifiable against `manifest.signing.key_fingerprint_spki_b64`.
TA-5	NuWyre's Postgres database is operated with point-in-time recovery + offsite backups + access controls per industry-standard operational practice.	Partially — backup integrity is operator-attested; the system-description document at `/legal` enumerates current controls. Verifier does not check backup state.
TA-6	The customer's adapter (whichever voice/AI platform delivers events to NuWyre) accurately reflects what the customer's AI agent actually said and heard.	No — see §1.1 out-of-scope. NuWyre's chain attests to ingestion-time content; not to upstream truthfulness.
TA-7	SHA-256 remains collision-resistant for the duration of the bundle's retention period.	No — depends on cryptographic-community consensus. NuWyre commits to migrating to a stronger hash via `event-v2` if SHA-256 collision-resistance materially degrades.
TA-8	Ed25519 signatures remain unforgeable without the private signing key.	No — same as TA-7; depends on cryptographic-community consensus.

1.3 Adversary model

NuWyre's adversaries fall into three classes by capability. Detection mechanisms for each class are documented in §6.

Class A — External attackers

Capabilities: network position between the customer's adapter and NuWyre ingestion; ability to attempt replay, injection, or forgery of HTTP requests at the ingestion endpoint; no privileged access to NuWyre infrastructure.

Defenses:

- TLS 1.3 with HSTS + certificate pinning at the SDK/CLI level for adapters that consume the published verifier.
- Per-organization API keys gating ingestion; rate limiting + Turnstile- gated public form submission.
- HMAC webhook signatures (where the upstream platform supports them) validated at the adapter boundary; details per §6.
- Idempotency-key handling at ingestion to prevent replay-induced duplicate events from poisoning the chain.

Class B — Compromised customer-side credentials

Capabilities: possession of a valid NuWyre API key + ability to issue arbitrary `/v1/events` requests on behalf of the customer organization.

Defenses:

- The compromised key can inject false events into the customer's chain, but NuWyre's chain-integrity signal remains intact: the injected events ARE in the chain, with valid hashes, signed by the ingestion service. The integrity claim "these are the events NuWyre observed" remains true; what is no longer true is "these are the events the customer's AI agent actually emitted."
- Key rotation procedure (operator manual §6) bounds blast radius: prior-key signatures remain valid for already-anchored events; new events require the rotated key.
- Per-(organization, sequence_number) monotonicity prevents an attacker holding a compromised key from inserting events retroactively into earlier daily roots.

Class C — Insider with NuWyre infrastructure access

Capabilities: read + write access to the NuWyre Postgres database; ability to substitute the ingestion service binary; ability to forge entries in the events table; ability to recompute local hash chains.

Defenses:

- Daily Merkle roots anchored to the Bitcoin blockchain via OpenTimestamps + to RFC 3161 TSAs + to the public GitHub anchor repository. An insider can rewrite the local database but cannot rewrite Bitcoin blocks or unwind GitHub commits without ALSO controlling those external systems.
- Daily-root verification by the customer (or the customer's counsel, or a regulator) detects insider tampering on the next-daily-root cycle. Detection latency is bounded by anchor pipeline timing (typically under 24 hours; always under next-daily-root + anchor-pipeline-window).
- RLS policies + append-only triggers at the Postgres layer make tampering require explicit service-role bypass; the bypass leaves audit-log traces enumerated at `/legal/system-description.pdf`.

1.4 Threat catalog

The threats below are the canonical attack scenarios NuWyre defends against. Each is paired with the detection mechanism + the residual risk the customer should be aware of.

THREAT	DETECTION MECHANISM	RESIDUAL RISK
T1: Event insertion — adversary inserts fabricated events with valid-looking content.	Per-event chain integrity (prev_event_hash + event_hash + Ed25519 signature) + daily Merkle root anchored to Bitcoin via OTS + cross-mirrored at GitHub. An inserted event either lacks a valid prior-event link OR shifts the daily root from its anchored value.	If insertion happens BEFORE the daily-root job AND the insider also forges the daily-root anchor commit AND the OTS receipt for that day, detection requires cross-anchor verification — which the multi-leg design preserves. Sub-24-hour insertion windows: bounded by daily-root cadence.
T2: Event modification — adversary changes the content of a previously-stored event.	Same as T1. The modified event's recomputed event_hash differs from the stored event_hash, breaking the chain at that point + invalidating the per-day Merkle root.	None for anchored days; an attacker who modifies an event in the same UTC day BEFORE the daily-root job runs leaves no Bitcoin trace, but the GitHub anchor + the customer's local copy of the bundle (if exported pre-tamper) still detect it.
T3: Event deletion — adversary deletes a record from the database.	Sequence-number gap detection + chain-break detection (next event's prev_event_hash points at a now-missing event).	An adversary who deletes a tail-of-chain event AND the records of higher sequence numbers AND the daily-root anchor commit can produce a "shortened" but internally-consistent chain — the cross-anchor design detects this when the anchored Merkle root from the next-daily-root cycle no longer matches.
T4: Event reordering — adversary reorders events within a chain.	Per-event sequence_number + per-event prev_event_hash form a tamper-evident sequence; reordering breaks the hash links.	V1's check 3 (hash chain) does NOT enforce file-line ordering of events within a bundle's `events.jsonl` — only the per-event chain links. A future verifier may tighten this.
T5: Backdating — adversary submits an event with a forged earlier `timestamp_iso` to make it appear to predate something.	Server-side `provenance.ingestion_timestamp` is set at the API boundary, NOT trusted from the request body. The DAILY-ROOT ANCHORING WINDOW is keyed on `provenance.ingestion_timestamp`, not `forensic.timestamp_iso`. Backdating the adapter timestamp does NOT change which day the event anchors under. See §2 anchoring window invariant.	An adversary with network position + clock-forging capability against NuWyre's server clock could shift `ingestion_timestamp` itself; defense is NTP + monitored clock-skew at the operator side (system-description document).
T6: Bundle tampering — adversary modifies an exported bundle's bytes after the customer receives it.	Every artifact in the bundle has its SHA-256 recorded in `manifest.artifacts`; the manifest itself is Ed25519-signed; verifier recomputes per-file SHA-256s and re-verifies the signature. A single bit-flip in any artifact surfaces at Check 2 (artifact integrity).	Detection assumes the customer has the genuine NuWyre public key; key-substitution attacks are addressed via TA-4.
T7: Bundle substitution — adversary substitutes a different (but cryptographically valid) NuWyre-signed bundle for the one the customer received.	Customer should record the bundle's `manifest.bundle_id` + `manifest.daily_roots[]` at receipt time; later verification confirms the same bundle is being verified. Out-of-band hash-recording is a customer-side discipline; NuWyre publishes the bundle hash at export time.	Customer-side hygiene-dependent; NuWyre cannot enforce.
T8: Key compromise — adversary obtains the NuWyre ingestion signing key.	Key compromise does NOT retroactively invalidate already-anchored events (their chain + Merkle position is already pinned in Bitcoin + GitHub). Going-forward: rotation procedure per operator manual §6; the rotated key's effective-after-timestamp is published; the verifier's pinned-keys dispatch admits the new key for events with `generated_at` >= effective-after. Old key remains valid for older events.	Window between compromise + detection: an attacker can issue fake events signed with the compromised key; these will pass Check 1 but not Check 9 cross-anchor verification once the customer's next bundle export is verified. Operator manual specifies the customer-notification timeline upon key-compromise discovery.

THREAT	DETECTION MECHANISM	RESIDUAL RISK
T9: OTS calendar compromise — adversary compromises an OpenTimestamps calendar server.	Multi-calendar OTS submissions; verifier accepts any successful Bitcoin anchor from any calendar. Single-calendar compromise does not invalidate the anchor for which Bitcoin attestation has landed.	Compromise BEFORE Bitcoin attestation lands could prevent that calendar from upgrading; defense is multi-calendar submission. Compromise of ALL calendars simultaneously: see §1.1 fantasy-threat-model exclusion.
T10: TSA compromise — adversary compromises one of the three RFC 3161 TSAs.	Spec §11 mandates ≥2-of-3 distinct TSAs verify; single-TSA compromise leaves the threshold satisfied.	Compromise of ≥2-of-3 TSAs simultaneously: would require independent compromise of three commercial timestamping authorities operated by different organizations. Detection: the verifier surfaces partial-verification verdict for the affected day; customer notification required per operator manual.
T11: GitHub anchor repo compromise — adversary forges or deletes commits in NuWyre/anchors.	Codeberg mirror provides cross-anchor verification; Bitcoin OTS receipts pin the daily root independently. GitHub anchor is the EASIEST of the three legs to verify but not the SOLE leg.	A NuWyre-side insider with both DB and GitHub access could produce a bundle that passes Check 7 against a forged anchor commit; Check 5 (OTS Bitcoin) provides independent verification.
T12: Denial-of-receipt — customer's adapter submits an event for which NuWyre produces no record.	`/v1/events` returns a synchronous response containing the assigned `event_id` + `event_hash` + `sequence_number` + `daily_root_pending` indicator. Adapters MUST persist this response; absence of a NuWyre acknowledgement is the operator's signal that no record was produced.	Customer-side adapter hygiene-dependent; NuWyre provides the acknowledgement primitive, not the persistence.

1.5 Detection vs. prevention boundary

NuWyre's promise is **tamper-evident**, not tamper-proof. This distinction is mechanical, not rhetorical:

Prevented at the storage layer: tampering with the *current* state of the database. RLS policies + append-only triggers + per-table CHECK constraints + service-role-gated bypass paths make straightforward UPDATE / DELETE / re-INSERT impossible without explicit operator-side service-role context. The system-description document at `/legal` enumerates every service-role-bypass path with its documented operational justification.

Detected post-tamper: tampering that bypasses storage-layer constraints (insider with service-role context; direct database manipulation; substitution of the ingestion service binary). The daily Merkle root + cross-anchor verification surface tamper events within the next-daily-root window. Detection latency is bounded but not zero.

Not defended against: the complete-infrastructure-takeover threat where the adversary simultaneously controls NuWyre's DB, GitHub account, OTS calendar submissions, AND Bitcoin block production at the relevant block-heights. This is a nation-state-cost threat explicitly excluded per §1.1.

The customer's procurement decision should weigh: **does the expected adversary's capability fit Class A or Class B (defended) or Class C with bounded-window detection (defended within latency window) or the complete-takeover scenario (not defended)?** For most regulated AI agent deployment contexts — TCPA, HIPAA, GDPR, broker-dealer recordkeeping — Class C with bounded-window detection is the relevant threat. NuWyre is appropriate. For threat models that require nation-state-resistant guarantees, NuWyre is one necessary layer but not sufficient on its own.

1.6 Residual risks the customer must accept

The honest list of what NuWyre cannot guarantee:

1. **Pre-NuWyre integrity**. NuWyre attests to what was ingested. The integrity of the upstream pipeline (the AI agent's output, the voice platform's transcription accuracy, the customer's consent-capture system) is the customer's responsibility.
2. **Detection latency**. Daily Merkle roots fire once per UTC day; anchor commits land within the next-daily-root + anchor-pipeline window. A tamper-event followed by recovery of integrity within the same UTC day before the daily-root job runs may evade anchor-level detection. Customer-held bundle copies provide redundant detection.
3. **Bundle hygiene**. Customers receive evidence bundles; NuWyre cannot enforce that customers store them with appropriate integrity controls. Out-of-band hash-recording at receipt time is the customer's discipline.
4. **External-dependency continuity**. Bitcoin chain health, GitHub availability, and TSA continuity are external dependencies. The multi-leg design mitigates single-dependency failures; multi- dependency failures are out-of-scope per §1.1.
5. **Standards-track maturity**. Spec v1.x is the first publicly- ratified version. The conformance fixture suite + reference verifier provide implementation-validated invariants; standards- community review will surface edge cases in production. Customers deploying NuWyre as their sole evidence layer should maintain an alternative integrity surface (off-system backups, a second evidence vendor) until v1.x has matured through additional real-world deployments.

The honest acknowledgment of these limitations is itself part of the methodology. A vendor who claims none of these limitations apply should be treated with skepticism. See §9 for the comprehensive limitations enumeration.

2. Integrity Model

This section describes the cryptographic and operational mechanisms that produce NuWyre's tamper-evident guarantees. Every claim in §1 about adversary detection ultimately reduces to one of the mechanisms documented here. A reviewer who wants to validate NuWyre's integrity claim from first principles should be able to start at the event record and trace forward to an externally-anchored proof in either the Bitcoin blockchain, an RFC 3161 TSA timestamp, or a signed GitHub commit.

The model rests on **multi-leg anchoring**: every daily Merkle root is independently anchored to (a) the Bitcoin blockchain via OpenTimestamps, (b) two or three RFC 3161 commercial TSAs, and (c) a signed commit in the public NuWyre/anchors GitHub repository. A relying party who validates ANY ONE of these legs has cryptographic evidence the root existed at the anchored time. Validating multiple legs raises the cost of a successful tamper to nation-state level per §1.

2.1 Event-level integrity

Every event ingested by NuWyre receives four cryptographic fields under `forensic`:

```
{
  "forensic": {
    "content_hash":          "<sha256-hex of canonical content payload>",
    "prev_event_hash":       "<event_hash of the prior event in this org's chain>",
    "event_hash":            "<sha256-hex linking the prior 4 fields>",
    "sequence_number":       <monotonic integer per organization>,
    "timestamp_iso":         "<adapter-supplied wall-clock time>",
    "timestamp_unix_ns":     "<stringified bigint nanoseconds; preserved across PostgREST>",
    "ingestion_signature":   "<Ed25519 signature, 88-char base64>"
  }
}
```

2.1.1 content_hash

`content_hash` is the SHA-256 of the RFC 8785 (JCS) canonicalization of the event's `content` field — the role, content, content_hash, tool_calls, prompt_hash, system_prompt_hash sub-fields. JCS canonical JSON pins key ordering + number formatting + Unicode handling, eliminating implementation-specific serialization variance that would otherwise produce divergent hashes for semantically- identical content.

Why JCS rather than ad-hoc JSON: a third-party implementer writing their own NuWyre-conformant content_hash function must be able to produce byte-identical output to NuWyre's TypeScript implementation and the Go verifier's implementation. JCS is the single standard both can encode against without coordination.

The KAT-2 (Known Answer Test) golden vector at `packages/evidence/tests/audit-log-export.test.ts:564-650` pins the expected content_hash for a specific input; the Go verifier consumer test at `apps/cli/internal/checks/check9_audit_log_kats_test.go` asserts byte-equivalence. Drift between the TS and Go implementations fails CI.

2.1.2 event_hash

`event_hash` is the SHA-256 of the canonical concatenation of four fields per spec §6.2:

```
event_hash = sha256(
  content_hash      ||
  prev_event_hash   ||
  sequence_number   ||  // stringified decimal
  timestamp_unix_ns ||  // stringified bigint
)
```

The four-field composition is verbatim across the primary-event chain (§6.2) and the audit-log-event chain (§16.2.2) — same implementation, same byte-encoding, same KAT (KAT-3) pinning. The chain integrity invariant is: for any event `E_n` in an organization's chain, recomputing `event_hash(E_n)` from `E_n`'s stored fields and comparing against `E_(n+1).prev_event_hash` must match. Any divergence indicates tampering at `E_n` or `E_(n+1)`.

2.1.3 Genesis sentinel

The first event in an organization's chain has `prev_event_hash = "0000000000000000000000000000000000000000000000000000000000000000"` (64 zero hex characters). The verifier treats this as the canonical chain-start marker; any non-zero `prev_event_hash` on a `sequence_number = 0` event is a chain integrity violation.

2.1.4 ingestion_signature

Each event's `ingestion_signature` is an Ed25519 signature over the canonical concatenation of the event's chain-integrity fields, produced by NuWyre's ingestion service using the issuer signing key. The signature attests "NuWyre's ingestion service observed this exact event with this exact chain position at ingestion time."

The signing key is pinned in the verifier binary (compile-time embedded; key fingerprint published in this document at §2.4). Key rotation produces a new pinned key with an `EffectiveAfter` UTC timestamp; the verifier dispatches to the appropriate key based on the event's `generated_at` per spec §5 + key-management procedure at §2.4.

2.2 Per-organization hash chain

Events for an organization form a single chain spanning all session boundaries within that organization. The chain invariants:

- **Per-(organization_id, sequence_number) uniqueness**: enforced by Postgres UNIQUE constraint + by the `organization_chain_state` table that assigns the next sequence_number atomically via SECURITY DEFINER RPC. Two concurrent ingestion requests for the same organization will receive distinct sequence_numbers; the ingestion RPC's optimistic-concurrency-control loop retries on conflict.
- **Gap-free monotonicity**: sequence_numbers are dense within an organization (0, 1, 2, ... with no gaps). Verifier detects gaps during chain reconstruction.
- **Append-only at the storage layer**: per-table append-only triggers reject UPDATE and DELETE operations on the `events` table except via documented service-role bypass paths (retention sweep — see operator manual §7; tombstone redaction — see §8).
- **Session-boundary spanning**: `identity.session_id` is a column tag on each event but is NOT a separate chain. A single org's chain can interleave events across many concurrent sessions. The spec §6 chain invariant applies at the organization level.

2.3 Canonical encoding (RFC 8785 JCS)

NuWyre uses RFC 8785 (JSON Canonicalization Scheme) for every hash-input serialization. The JCS choice is documented per spec §6.1

- enforced by:

- The TypeScript reference implementation at `packages/schema/src/` uses a JCS-conformant canonicalizer.
- The Go verifier at `apps/cli/internal/checks/` uses a JCS-conformant canonicalizer.
- The cross-language byte-equivalence KAT vectors at `apps/cli/internal/checks/testdata/audit_log_kats_v1.json` pin specific (input, canonical_output_sha256) tuples. CI fails if the Go and TS implementations diverge.

A third-party implementer writing a conformant verifier or writer MUST use a JCS-conformant canonicalizer. Common ad-hoc JSON serializers (Python `json.dumps()` with default settings; JavaScript `JSON.stringify()` without canonical key ordering) will produce divergent hashes.

2.4 Ingestion signing key (Ed25519)

NuWyre uses Ed25519 for ingestion signatures. The choice rationale:

- **Deterministic**: same input + same key produces the same signature byte-for-byte. Eliminates implementation-divergence risk between the TS signer and the Go verifier.
- **Compact**: 64-byte signatures; 32-byte public keys; 32-byte private keys.
- **No parameter choice**: no curve negotiation, no padding scheme, no random-source dependency. Reduces attack surface vs RSA or ECDSA-with-secp256k1.
- **Library maturity**: well-supported in Go (`crypto/ed25519`, standard library) + TypeScript (`@noble/ed25519` audited implementation) + every standard library expected of a third- party implementer.

2.4.1 Pinned public keys

Two pinned public keys ship in the V1 verifier:

- **`issuer-prod-v1`** — production signing key for customer-export bundles. Effective from `2026-05-01T00:00:00Z` (post-Phase-5 deploy bootstrap). SPKI fingerprint published in the verifier's `keys` command output + in this methodology PDF appendix.
- **`issuer-dev-v1`** — development signing key for example-demo + sandbox-preview + audit-log-export fixture bundles. NEVER signs production bundles; the verifier emits a `DEVELOPMENT BUNDLE — verified with dev key, not for production trust` warning when this key is encountered + the operator must opt in with `--allow-dev-key` to fold the warning into pass.

2.4.2 Key rotation procedure

When a production key rotates:

1. The new key is generated in a KMS-backed environment (per Phase 6 Item 1 two-key topology) + the public key SPKI is published.
2. A new pinned-keys release of the verifier ships with the new public key appended to the dispatch table. The old key remains pinned with its now-set `EffectiveBefore` timestamp.
3. New events with `generated_at >= EffectiveAfter[new_key]` are signed with the new key. Old events with `generated_at < EffectiveAfter[new_key]` remain signed with the old key.
4. Verifier dispatches per-event by checking the event's `generated_at` against each pinned key's effective-period window.
5. Bundles spanning a rotation window contain signatures from multiple keys; the verifier validates each signature against the appropriate pinned key. The bundle's `manifest.signing` block declares which key_id signed the manifest itself; events within the bundle may use different keys per their `generated_at`.

2.4.3 Key compromise procedure

If the ingestion signing key is compromised:

1. Operator-side detection (anomalous signatures, unauthorized bundle exports, key-material exposure) triggers immediate rotation per §2.4.2.
2. The compromised key's `EffectiveBefore` is set to the compromise-detection timestamp + verifier ships updated dispatch.
3. Customer notification: per operator manual §6, all customers receive notification within 24 hours of detected compromise + guidance on re-verifying their already-received bundles (already-anchored events remain cryptographically anchored at their Bitcoin/TSA/GitHub roots; future event ingestion uses rotated key).
4. Already-anchored events are NOT invalidated. Their chain + Merkle position + Bitcoin attestation + TSA attestation + GitHub commit are pinned independently of the signing key. The signing key's role is to attest "NuWyre ingested this," which remains attestable for pre-compromise events via the pre-rotation key's verified signatures.

2.5 Daily Merkle root

Once per UTC day, the daily-root cron job at `apps/api/src/app/api/cron/daily-root/route.ts` computes the canonical Merkle root over all events whose `provenance.ingestion_timestamp` falls within the day window.

2.5.1 Merkle construction

Per spec §8:

- Leaves are event_hash values, sorted by `(organization_id, sequence_number)` ASCENDING (canonical leaf-ordering invariant).
- Tree is binary, balanced by zero-padding to the next power of two with the canonical sentinel `0000000000000000000000000000000000000000000000000000000000000000`.
- Internal nodes are `sha256(left_child || right_child)` (raw bytes, NOT hex concatenation).
- Root is the top node after recursive pair-hashing.

2.5.2 Anchoring window invariant

The daily Merkle root anchors every event whose server-side `ingestion_timestamp` falls within the 24-hour UTC window. Adapter- supplied `timestamp_iso` (the event's wall-clock time as the adapter reported it) does NOT affect daily-root membership.

This means:

- A back-dated event arriving late — for example, an event with adapter-claimed `timestamp_iso=2026-04-22T23:59:59Z` that the server first observes at `ingestion_timestamp=2026-04-23T00:06:00Z` — is anchored on **2026-04-23** (the day it was INGESTED), not on 2026-04-22 (the day it claims to have occurred).
- The `forensic.timestamp_iso` field remains on every event for adapter-side temporal reconstruction; it is signed into the event hash chain and is verifiable. But it does NOT determine which daily Merkle root anchors the event.
- The `provenance.ingestion_timestamp` field is **server-stamped** at `/v1/events` ingestion and at the internal adapter pipeline; customer-submitted values in this field are discarded at the API boundary. This is enforced at the request handler layer, not via trust in the customer's claim.

Why this design. NuWyre's evidentiary contract is over what the server OBSERVED at ingestion, not over what an adapter CLAIMS. The anchoring guarantee is "every event the server ingested during [day, day+1) UTC is committed to the daily Merkle root for that day" — a statement that's verifiable against `ingestion_timestamp` because `ingestion_timestamp` is set by NuWyre's server clock. Anchoring against adapter-claimed `timestamp_iso` would allow

an adapter (or compromised api_key) to manipulate which day its events were anchored on, breaking the temporal integrity claim.

Verifier behavior. The Go CLI verifier groups events into days using `provenance.ingestion_timestamp.slice(0,10)` (UTC date prefix) and reconstructs the per-day Merkle tree from those events' `event_hash` values. The reconstructed tree's root must equal the `daily_roots.root_hash` value in the bundle's `daily_roots.json` for the corresponding date. A bundle is considered tampered if these diverge for any anchored day.

Late-arrival handling. Events that arrive at the server AFTER their day's daily-root job has fired are anchored under the day they were ingested (the day the server SAW them). They are NOT silently dropped from the chain — they receive the next available `sequence_number`, contain a valid `prev_event_hash` pointing at the prior event in the org's chain, and are anchored under the daily Merkle root of their actual ingestion day.

2.5.3 Dual-subtree composition (audit-log-export bundles)

Audit-log-export bundles use a dual-subtree daily root per spec §16.3:

```
daily_root = sha256(events_subtree_root || audit_log_subtree_root)
```

The events subtree is computed over primary `events` table records; the audit-log subtree is computed over `audit_log_events` table records. For customer-export bundles with no audit-log events on the day, the audit-log subtree root is the zero sentinel; for operator- only audit-log-export bundles, the events subtree root is the zero sentinel.

This composition lets the same daily_root anchor both primary event integrity and audit-log event integrity in a single Bitcoin attestation, while keeping the two integrity domains cryptographically independent at the subtree level. The KAT-5 vector at the same audit_log_kats_v1.json pins the dual-subtree composition.

2.6 OpenTimestamps Bitcoin anchoring

After the daily root is computed, it is submitted to multiple OpenTimestamps calendar servers. Each calendar returns a `.ots` receipt — a pending calendar attestation that the root was observed at the submission time. The OTS protocol then upgrades these to Bitcoin attestations once the root is folded into a Bitcoin block's Merkle tree (typically within minutes; spec-mandated detection latency budget is documented at the verifier's check 5).

2.6.1 What the OTS receipt proves

A Bitcoin-attested OTS receipt proves: "The Merkle root in this receipt was observed by the OpenTimestamps calendar at time T1, and was anchored to Bitcoin block B at time T2." The relying party can:

- Independently fetch block B from any Bitcoin node.
- Verify the OTS receipt's claimed block hash matches the public Bitcoin chain.
- Verify the root in the receipt matches the bundle's claimed `daily_root` for the anchored date.

A successful chain produces: "NuWyre's daily root for date D was folded into Bitcoin block B at time T, which is independently verifiable against the public Bitcoin chain."

2.6.2 Pending-state semantics

OTS receipts arrive in two states:

- **Pending**: calendar attestation only; Bitcoin block confirmation not yet observed. Verifier surfaces as `warn` (folds to pass under `--allow-pending-ots`).
- **Confirmed**: Bitcoin block attestation present; verifier fully validates against the public chain.

The transition from pending to confirmed typically takes minutes to hours depending on Bitcoin block-time variance + OTS calendar upgrade cadence.

2.7 RFC 3161 TSA timestamping

In parallel with OTS submission, the daily root is submitted to multiple RFC 3161 Time-Stamping Authorities (V1: FreeTSA, Sectigo, DigiCert). Each TSA returns a `.tsr` (timestamp response) + `.chain.pem` (certificate chain) proving the TSA observed the root at a specific time.

The verification requirement per spec §11 is **≥2-of-3 TSAs verify**. Single-TSA compromise leaves the threshold satisfied; the verifier surfaces partial-verification when only 2 verify (still passing) + fail when only 1 verifies. The cert chains are validated against pinned trust roots embedded in the verifier binary.

2.7.1 Why both OTS and RFC 3161

OTS provides anchoring with public-blockchain proof-of-work cost backing. RFC 3161 provides anchoring with commercial-TSA legal- admissibility precedent (decades of court use). The combination delivers:

- **OTS** to a regulator who wants permission-less verifiability against an open chain.
- **RFC 3161** to a court that prefers the historical legal-evidence precedent.

Both legs anchor to the same daily root; the relying party chooses which they trust. A bundle that passes both is anchored under both trust models simultaneously.

2.8 GitHub anchor mirror

The third leg of multi-leg anchoring is a signed commit to the public `NuWyre/anchors` GitHub repository (mirrored to Codeberg). The daily root is recorded as JSON in a date-organized path:

```
daily_roots/<organization_id>/<utc_date>/<bundle_metadata>.json
```

The commit is signed with NuWyre's GitHub PGP key (separate from the ingestion signing key; documented in the system-description PDF at /legal). The verifier's check 7 fetches the commit + validates the claimed `daily_root` matches the customer's bundle.

2.8.1 Anchor-pending state

In the V1 deploy-bootstrap state, daily-roots are computed + submitted to OTS + RFC 3161 immediately, but the GitHub anchor commit is deferred to a manual operator-signed lift (Phase 5+ infrastructure). Bundles produced in this window declare `mirror_status: "anchor-pending"` on the GitHub anchor leg. The verifier surfaces `warn` (folds to pass under `--allow-anchor- pending`).

The anchor-pending state is documented at v1 launch as transparent: operators receive a bundle that explicitly states the GitHub leg is deferred; the OTS + RFC 3161 legs provide full anchor coverage. Production deployment removes the anchor-pending state once the operator-signed lift infrastructure ships.

2.9 Tombstone records for redaction

When a record must be redacted for compliance reasons (PII discovery post-ingestion; legal-hold-cleared post-period; customer-data- deletion request), NuWyre does NOT delete the underlying event row. Instead:

1. The event's `content` field is replaced with a tombstone marker `[REDACTED:<reason>]`.
2. The original `content_hash` is preserved in the row — the redaction does NOT recompute hashes.

3. The chain integrity remains intact: prev_event_hash + event_hash
    - sequence_number + ingestion_signature all stay as originally computed.
4. The tombstone is itself a recorded redaction event in `audit_log_events` with the redactor's identity + timestamp + justification.

This design preserves the cryptographic chain (the row is still verifiable against the daily Merkle root) while allowing compliance-mandated content removal. A bundle export of a tombstoned event reveals the tombstone marker; the verifier accepts the tombstone-content + verifies the chain integrity.

The trade-off: a relying party reading a tombstoned event sees only the marker, not the original content. The original content's content_hash is preserved in the chain — a relying party with a pre-redaction copy of the content can validate it matches the preserved content_hash. The customer-side discipline of preserving pre-redaction content copies (per operator manual §8) is the forensic-completeness substrate.

2.10 Putting it together: full integrity reconstruction

A relying party (regulator, auditor, opposing counsel) reconstructing NuWyre's integrity claim for a specific event would:

1. **Locate the event** in the bundle's `events.jsonl`. Verify its `content_hash` by canonicalizing the `content` payload + SHA-256.
2. **Verify the chain link** by recomputing `event_hash` from the four-field composition + comparing against `events.jsonl[next].prev_event_hash`.
3. **Verify the Merkle proof** in `merkle_proofs.json` for the event: walk the sibling-hash path + confirm the walked root equals the bundle's claimed daily_root for the event's ingestion date.
4. **Verify the daily root** is anchored:
    - Independently fetch the OTS receipt's claimed Bitcoin block + confirm the block's Merkle tree includes the OTS commitment + confirm the daily_root matches.
    - Validate the RFC 3161 TSA receipts (≥2-of-3) against pinned trust roots.
    - Fetch the GitHub anchor commit from NuWyre/anchors + verify the signed-commit + confirm the daily_root matches.
5. **Verify the bundle signature**: confirm `manifest.signing.key_id` is a pinned NuWyre key (effective at the bundle's `generated_at`)
    - verify the manifest's Ed25519 signature.
6. **Cross-reference dates**: confirm the OTS Bitcoin block timestamp is AFTER `manifest.daily_roots[].generated_at` (anchor-can't- predate-submission invariant).

A successful walk through all six steps establishes: "The event in this bundle was ingested by NuWyre at the recorded time, is part of a valid hash chain extending back to genesis, was anchored under the daily Merkle root for its ingestion date, and that daily root is independently attested in the Bitcoin blockchain, by commercial RFC 3161 TSAs, and by a signed GitHub commit."

The complete walk is mechanically executable: `nuwyre verify <bundle.zip>` performs all six steps. The methodology document is the prose form of what the verifier does mechanically. A reviewer who wants to validate the verifier's logic should run the conformance fixture suite (14 fixtures at `docs/spec/fixtures/bundle-format-v1/`) which exercises the full integrity reconstruction across valid + tampered bundle variants.

3. Evidence Format Specification

NuWyre's evidence format is a versioned, publicly-specified bundle that packages everything a relying party needs to verify a set of events end-to-end. The format is the contract: any third-party verifier implementation that reads bundles per this specification produces the same verdict as NuWyre's reference verifier.

The canonical specification document is `docs/spec/bundle-format-v1.md` in the source repository. This methodology section is the prose- summary form of that contract; for binding mechanical detail (exact byte sequences, JCS canonicalization rules, edge cases), the specification document is authoritative.

3.1 Bundle structure

An evidence bundle is a ZIP archive containing a defined set of files. The ZIP is a standard PKZip-compatible archive (`archive/zip` Go library compatible; `zipfile` Python compatible; `unzip` POSIX compatible). The structure for a customer-export bundle:

```
bundle.zip
├── manifest.json              — bundle metadata + per-file SHA-256s + anchor state
├── signature.sig              — Ed25519 over canonical manifest bytes
├── events.jsonl               — one event per line, JCS-canonical JSON
├── merkle_proofs.json         — Merkle proof per event up to daily root
├── daily_roots.json           — per-day Merkle root + per-day anchor metadata
├── evaluations.jsonl          — policy-pack evaluation results per event
├── audio/                     — content-addressed audio files (when present)
│   └── <sha256>.<ext>         — file named for its own SHA-256
├── ots_receipts/              — OpenTimestamps Bitcoin anchor receipts
│   └── <utc_date>.ots         — one per anchored daily root
├── rfc3161_receipts/          — RFC 3161 commercial TSA receipts
│   ├── <utc_date>__<tsa>.tsr      — timestamp response per TSA
│   └── <utc_date>__<tsa>.chain.pem — cert chain per TSA
├── github_anchors/            — GitHub anchor commit references
│   └── <utc_date>.json        — anchor commit metadata per day
├── cover.pdf                  — human-readable summary; methodology pointer
├── verify.md                  — offline verification instructions
└── scenario_index.json        — sandbox/demo bundle scenario metadata (when applicable)
```

The audit-log-export bundle structure substitutes `audit_log_events. jsonl` + `audit_log_subtree.json` for the primary `events.jsonl` + `merkle_proofs.json` (or runs both for combined bundles). See §3.5.

3.2 Bundle types

The `manifest.bundle_type` field is a closed-vocabulary enum selecting bundle-specific verification logic:

- **`customer-export`** — production customer-facing evidence bundle. Signed by `issuer-prod-v1`. Contains real customer events + evaluations + audio (when applicable) + full anchor leg (OTS + RFC 3161 + GitHub).
- **`audit-log-export`** — operator-side audit-log evidence bundle. Contains audit_log_events (not customer events) + dual-subtree composition (events subtree may be empty per spec §16.3.1). Signed by dev key in V1 (production two-key topology forward-bookmarked per §2.4); the verifier emits the development-bundle warning acceptable under `--allow-dev-key` opt-in.
- **`example-demo`** — public example bundle showcasing the format. Contains synthetic events for demonstration. Signed by `issuer-dev-v1`; relying party must opt in with `--allow-dev-key`.
- **`sandbox-preview`** — customer-facing sandbox bundle generated from operator's own redacted event samples for evaluation purposes. Signed by dev key; anchor-pending state (the sandbox bundle is for demonstration, not for production trust).

The bundle-type field is signed into the manifest; an adversary cannot substitute a customer-export bundle's type to evade verification logic.

3.3 manifest.json

The manifest is the bundle's load-bearing metadata document. It declares:

```
{
  "schema_version": 2,
  "bundle_format": "nuwyre-bundle/v2",
  "bundle_id": "<UUID-v4>",
  "bundle_type": "<customer-export | audit-log-export | example-demo | sandbox-preview>",
  "organization_id": "<UUID-v4>",
  "organization_name": "<human-readable name>",
  "generated_at": "<ISO 8601 UTC>",
  "filename": "<canonical bundle filename>",
  "signing": {
    "schema_version": 1,
    "signatures": [
      {
        "algorithm": "ed25519",
        "key_id": "<issuer-prod-v2-ed25519 | issuer-dev-v2-ed25519>",
        "key_fingerprint_spki_b64": "<base64 SPKI fingerprint>",
        "key_purpose": "Ed25519 manifest signature; v2.0.0-rc1+ dual-sig topology"
      },
      {
        "algorithm": "ml-dsa-65",
        "key_id": "<issuer-prod-v2-ml-dsa-65 | issuer-dev-v2-ml-dsa-65>",
        "key_fingerprint_spki_b64": "<base64 SPKI fingerprint>",
        "key_purpose": "ML-DSA-65 manifest signature; v2.0.0-rc1+ dual-sig topology"
      }
    ]
  },
  "anchor_status": {
    "ots_status": "<pending | confirmed | failed>",
    "rfc3161_status": "<not_attempted | partial | verified | failed>",
    "github_status": "<not_attempted | anchor-pending | anchored | failed>"
  },
  "anchors": {
    "opentimestamps": {
      "receipt_path": "<ots_receipts/<utc_date>.ots | null when not_attempted>",
      "status": "<submitted-pending-bitcoin-confirmation | bitcoin-confirmed | failed>",
      "submitted_at": "<ISO 8601 UTC>"
    },
    "rfc3161": [
      {
        "tsa_name": "<freetsa | sectigo | digicert>",
        "receipt_path": "rfc3161_receipts/<utc_date>__<tsa>.tsr",
        "chain_path": "rfc3161_receipts/<utc_date>__<tsa>.chain.pem"
      }
    ],
    "github": {
      "repo": "https://github.com/NuWyre/anchors",
      "commit_sha": "<40-char SHA-1 | null when anchor-pending>",
      "mirror_status": "<not_attempted | anchor-pending | anchored | failed>"
    }
  },
  "daily_root": "<64-char lowercase hex SHA-256>",
  "daily_roots": [
    { "date": "<YYYY-MM-DD>", "root": "<sha256-hex>" }
  ],
  "merkle_subtrees": {
    "events_root": "<sha256-hex; zero sentinel when not applicable>",
    "audit_log_root": "<sha256-hex; zero sentinel when not applicable>"
  },
  "event_count": <integer>,
  "audit_log_event_count": <integer>,
  "artifacts": [
    {
      "path": "<bundle-relative path>",
      "sha256": "<64-char lowercase hex>",
```

```
      "size_bytes": <integer>
    }
  ]
}
```

The manifest is canonicalized via RFC 8785 JCS before signing. The relying party recomputes the canonical form + verifies the signature against the pinned public key (per Check 1 in §4.4).

3.4 signature.sig

For **bundle_format v2** (current; v2.0.0 final landed 2026-05-22 with informative additions at v2.0.1): `signature.sig` is a **JCS-canonicalized JSON multi-signature container** per spec §5 + §18.2 with this shape:

```
{
  "schema_version": 1,
  "signed_artifact": "manifest.json",
  "signatures": [
    {
      "algorithm": "ed25519",
      "key_fingerprint_spki_b64": "<base64 SPKI fingerprint>",
      "key_id": "issuer-prod-v2-ed25519",
      "signature_b64": "<88 base64 chars = raw 64-byte Ed25519 sig>"
    },
    {
      "algorithm": "ml-dsa-65",
      "key_fingerprint_spki_b64": "<base64 SPKI fingerprint>",
      "key_id": "issuer-prod-v2-ml-dsa-65",
      "signature_b64": "<4412 base64 chars no padding = raw 3309-byte ML-DSA-65 sig per FIPS 204 §4 Table 1>"
    }
  ]
}
```

Both signatures MUST independently verify for the bundle to be authentic. Each signature covers the **same** JCS-canonicalized `manifest.json` bytes. The two `signatures[]` entries are positional (Ed25519 at index 0, ML-DSA-65 at index 1) and the verifier cross-validates that `signature.sig.signatures[i].key_id` byte-equals `manifest.signing.signatures[i].key_id` for each `i`.

For **bundle_format v1** (legacy; locked across the v1.0.x amendment series; verifiers continue to accept indefinitely per §3.13.1): `signature.sig` is the raw single Ed25519 signature (64 bytes) over JCS-canonicalized manifest bytes (legacy flat shape; pre-dates the v2 JSON container). v1 verifiers continue to verify v1 bundles forever; v2 verifiers dispatch by the `bundle_format` identifier and verify either path.

Verification (v2; the current path):

1. Read `manifest.json` bytes.
2. Canonicalize via JCS (key sorting + canonical number formatting + canonical UTF-8 encoding).
3. Read `signature.sig` bytes. Parse as JSON (the bytes are themselves JCS-canonicalized JSON, NOT a raw byte concatenation). Validate the container shape per the spec §18.2 schema. Base64- decode `signatures[0].signature_b64` (88 chars → 64 raw bytes for Ed25519) and `signatures[1].signature_b64` (4412 chars, no padding → 3309 raw bytes for ML-DSA-65).
4. Cross-validate `signature.sig.signatures[i].key_id` equals `manifest.signing.signatures[i].key_id` for each `i` (the positional discipline guards against cross-signature smuggling). Look up the pinned public keys by these `key_id` values + verify each key is effective at `manifest.generated_at`.
5. Verify the Ed25519 signature over the canonicalized manifest bytes using the pinned Ed25519 public key.

6. Verify the ML-DSA-65 signature over the same canonicalized manifest bytes using the pinned ML-DSA-65 public key.
7. BOTH signatures MUST verify; if either fails, the bundle is rejected. The verifier's structured JSON output emits an `algorithm_verdicts: [{algorithm, status}, ...]` array populated for BOTH algorithms regardless of which fails per spec §18.10 (operators consuming `--json` output should read this rather than parsing the human-prose error message).

The dual-signature attestation: "NuWyre's issuer keys — one elliptic-curve, one post-quantum — both signed this exact canonical manifest at the recorded time." **Cryptographic forgery (defeating the signing algorithms themselves) requires defeating two schemes built on different mathematical foundations** — elliptic-curve discrete logarithm AND module-lattice problems — providing defense against present-day cryptanalysis and against a future cryptographically-relevant quantum computer (NIST projections place CRQC capability in the 2035-2040+ timeframe, intersecting NuWyre's multi-decade retention horizon per §7). **Key-custody compromise** (extraction of both private keys from NuWyre's signing infrastructure) is a separate threat model addressed by the key-custody controls described in `docs/legal/system-description.md` §7 — defense-in-depth here is at the algorithmic-cryptanalysis layer, not the key-bytes-at-rest layer.

Per-event `forensic.ingestion_signature` (§2.1.4) remains single Ed25519 in both v1 and v2. The manifest-level ML-DSA-65 signature **transitively** protects the event corpus through the daily Merkle root: each event's `content_hash` → `event_hash` participates in the daily root computed at export time; the daily root is embedded in `manifest.json`; the ML-DSA-65 manifest signature covers `manifest.json` bytes. An adversary forging a per-event Ed25519 ingestion signature on a substituted event would change the event_hash, breaking the Merkle-proof reconstruction at Check 4; they cannot change the daily_root without also defeating the ML-DSA-65 manifest signature. Therefore per-event Ed25519 is sufficient for v2 — the post-quantum coverage propagates from the manifest down through the daily root to the event corpus.

3.5 events.jsonl and audit_log_events.jsonl

Event records are stored as newline-delimited JSON (JSONL / NDJSON format). Each line is a complete event object per the `event-v1` schema (`docs/spec/event-v1.schema.json`) or the `audit-log-event-v1` schema (`docs/spec/audit-log-event-v1.schema.json`).

JSONL choice rationale:

- **Append-friendly**: the writer can append events as they're exported without rewriting the whole file.
- **Streamable**: the verifier can stream-parse the file without loading it entirely into memory.
- **One-line-per-record**: byte-level tampering of one event is trivially scoped; a relying party can spot which line the tamper is on.

Each event's `forensic.event_hash` field is computed at ingestion + preserved in the JSONL bytes. The verifier (Check 3) recomputes `event_hash` from the four-field composition (per §2.1.2) + compares against the stored value.

3.5.1 JCS canonicalization for event content

Event content (`event.content` for primary events; `event.actor` + `event.subject` + `event.event_type` for audit-log events) is canonicalized via RFC 8785 JCS before content_hash computation. This pins:

- Object key ordering (lexicographic by Unicode code point).
- Number formatting (no leading zeros, no trailing zeros after decimal, ASCII-only).
- String encoding (UTF-8 with specific escape sequences for control characters).

A third-party implementer writing a conformant content_hash producer MUST use a JCS-conformant canonicalizer. The KAT-2 golden vector at `packages/evidence/tests/audit-log-export.test.ts:564-650` pins specific input →

expected_content_hash tuples for cross-language implementation parity.

3.6 merkle_proofs.json (and audit_log_subtree.json)

The merkle proof file contains a proof per event, enabling the verifier to independently reconstruct the daily root from each event's leaf position:

```
{
  "schema_version": 1,
  "proofs": [
    {
      "event_id": "<event UUID>",
      "leaf": "<event_hash, sha256-hex>",
      "leaf_index": <position in canonical-ordered leaf list>,
      "path": [
        {
          "position": "<left | right>",
          "sibling": "<sha256-hex of sibling node at this level>"
        }
      ],
      "root": "<sha256-hex of reconstructed root>"
    }
  ],
  "root": "<sha256-hex of full Merkle tree root>"
}
```

The verifier walks each event's `path` from leaf to root, combining the leaf with each sibling per the `position`:

- `position: "right"` → `parent = sha256(leaf || sibling)`
- `position: "left"` → `parent = sha256(sibling || leaf)`

Concatenation is raw bytes (NOT hex). The walked root must equal `proofs[i].root` + the top-level `proofs.root` field. For primary events in customer-export bundles, the walked root must also equal `manifest.daily_root` for the relevant date.

For dual-subtree audit-log-export bundles, `audit_log_subtree.json` contains the audit-log subtree proofs; `merkle_proofs.json` contains the (typically empty) events subtree proofs. The daily root is composed: `daily_root = sha256(events_subtree_root || audit_log_subtree_root)` per §2.5.3 + spec §16.3.

3.7 daily_roots.json

Records the daily Merkle root + associated metadata for each UTC day spanned by the bundle:

```
{
  "schema_version": 1,
  "roots": [
    {
      "date": "<YYYY-MM-DD UTC>",
      "root": "<sha256-hex>",
      "event_count": <integer>,
      "ots_receipt_path": "<ots_receipts/<date>.ots | null>",
      "rfc3161_receipts": [
        {
          "tsa_name": "<tsa identifier>",
          "receipt_path": "rfc3161_receipts/<date>__<tsa>.tsr"
        }
      ],
      "github_anchor_path": "github_anchors/<date>.json"
    }
  ]
}
```

The verifier (Check 4 + Check 5 + Check 6 + Check 7) cross-references each daily root against:

- The Merkle proof reconstruction (Check 4).
- The OTS receipt's claimed digest (Check 5).
- The RFC 3161 receipts' signed digests (Check 6).
- The GitHub anchor commit's recorded root (Check 7).

A relying party verifying a multi-day bundle validates each day independently; a per-day failure does NOT invalidate other days' verification.

3.8 evaluations.jsonl

Policy-pack evaluation results, one per line per evaluation:

```
{
  "schema_version": 1,
  "evaluation_id": "<UUID>",
  "event_id": "<event UUID that was evaluated>",
  "rule_id": "<pack_id.rule_name>",
  "pack_id": "<namespace-shortname-vMajor>",
  "pack_version": "<semver>",
  "pack_body_hash": "<sha256-hex per §5.2>",
  "model_id": "<canonical model identifier>",
  "model_version": "<pinned version string>",
  "verdict": "<clean | flagged | error>",
  "severity": "<info | low | medium | high | critical>",
  "reasoning": "<LLM-generated structured reasoning>",
  "cross_validation": {
    "performed": <boolean>,
    "agreement": "<agree | disagree | n/a>"
  },
  "evaluated_at": "<ISO 8601 UTC>",
  "row_hash": "<sha256-hex of the evaluation row's canonical form>"
}
```

Each evaluation row is itself hash-recorded — the `row_hash` enables a relying party to verify the evaluation has not been modified since recording (Check 2 includes evaluations.jsonl per-line verification when present). The `pack_body_hash` enables a relying party to recover the exact pack revision used to produce the evaluation (per §5.2).

3.9 Anchor receipts

3.9.1 ots_receipts/.ots

OpenTimestamps protocol bytes per the OTS specification at https://opentimestamps.org. The format is binary; a verifier parses it via the OTS library:

1. Magic header (`\x00OpenTimestamps\x00\x00Proof\x00...`).
2. Version byte.
3. File hash op (SHA-256 of the daily root).
4. Sequences (one per calendar): operations that fold the daily-root hash into a Bitcoin block's Merkle root.

In pending state, the receipt contains only calendar attestations; in confirmed state, the receipt contains the Bitcoin block height + the proof path to the block's Merkle root.

3.9.2 rfc3161_receipts/___.{tsr,chain.pem}

`.tsr` is the PKCS#7 SignedData timestamp token per RFC 3161. It contains:

- The TSA's signature.
- The hashed message (the daily root).
- The TSA's timestamp claim.

`.chain.pem` is the X.509 certificate chain from the TSA's signing certificate up to a publicly-known root CA (pinned in the verifier binary). The chain enables a relying party to validate the signature's trust root without trusting NuWyre's representation.

V1 ships receipts from three TSAs: FreeTSA (non-profit), Sectigo (commercial), DigiCert (commercial). Spec §11 mandates ≥2-of-3 verification.

3.9.3 github_anchors/.json

Records the GitHub anchor commit metadata:

```
{
  "schema_version": 1,
  "date": "<YYYY-MM-DD UTC>",
  "root": "<daily_root sha256-hex>",
  "repo": "https://github.com/NuWyre/anchors",
  "commit_sha": "<40-char SHA-1 | null when anchor-pending>",
  "commit_sha_format": "<sha1 | sha256; future-proof>",
  "mirror_status": "<not_attempted | anchor-pending | anchored | failed>",
  "anchored_at": "<ISO 8601 UTC | null when not anchored>"
}
```

The verifier (Check 7) fetches the claimed commit from `NuWyre/anchors` (or its Codeberg mirror) + validates the referenced commit contains the daily root for the date.

3.10 audio/ directory

When the bundle contains audio (typically for voice-platform- integration customer-export bundles), the `audio/` directory contains content-addressed audio files:

```
audio/<sha256>.<extension>
```

Where `<sha256>` is the SHA-256 hex of the raw audio file bytes (NOT including the extension in the hash input). The extension indicates the audio format (`.wav`, `.mp3`, `.opus`, etc).

Audio is bound to events via the event's `content.audio_ref.hash` field — the event's content_hash incorporates this audio_ref. Tampering with the audio file (or substituting a different audio file at the same path) changes the file's actual SHA-256, which the verifier detects at Check 2 (artifact integrity).

See §8 for the full audio handling methodology including consent-capture context, retention-independence from events, and tombstone-redaction-without-chain-break.

3.11 cover.pdf and verify.md

`cover.pdf` is a human-readable bundle summary: bundle metadata, event counts, anchor states, methodology pointer. Generated at bundle creation time. Intended as the first-look document for a relying party who wants a non-cryptographic overview before running the verifier.

`verify.md` contains step-by-step instructions for the relying party to run offline verification against the bundle. Includes:

- One-line CLI installation.
- The `nuwyre verify <bundle.zip>` invocation.
- Common flag combinations.
- Expected exit codes.
- Where to seek help (the methodology document; the spec documentation; the verifier's source code).

Both documents are generated per-bundle and recorded in `manifest.artifacts` with their SHA-256s.

3.12 Reproducibility contract

The evidence format is designed for **deterministic regeneration**: the same input events + same policy pack + same pinned model version + same UTC day produce a byte-identical bundle.

This is enforced via:

- JCS canonicalization at every hash-input boundary.
- Pinned model versions for evaluations (per §5.3).
- Deterministic Ed25519 signatures (no random nonce in signing).
- Canonical leaf-ordering for Merkle construction (sequence_number ascending; per spec §8).
- Fixed-format anchor receipts (OTS protocol bytes, RFC 3161 PKCS#7 DER, JSON for GitHub anchors).

The KAT golden vectors at `apps/cli/internal/checks/testdata/audit_log_kats_v1.json` validate cross-language byte-equivalence: TS implementation produces byte-identical output to Go implementation for the pinned vectors. CI fails if either implementation drifts.

A relying party reproducing a bundle from source events should produce the same artifact bytes — the reproducibility is the substrate for "anyone can independently verify the bundle was generated honestly from the claimed source events."

3.13 Versioning and backward compatibility

The format is versioned at multiple levels:

- **`event-v1` schema** at `docs/spec/event-v1.schema.json`. Schema evolution follows semantic versioning (per spec governance §3).

- **`audit-log-event-v1` schema** at `docs/spec/audit-log-event-v1.schema.json`. Independently versioned but currently locked to schema_version 1.
- **`bundle-format-v1` specification** at `docs/spec/bundle-format-v1.md`. Versioned with minor revisions (currently v1.0.17 at time of writing); revision history at the spec's foot.
- **`pack-format-v1`** at `packages/policy/`. Independent versioning for policy packs (per §5.1).

3.13.1 Backward compatibility commitment

NuWyre commits to:

- A bundle generated under spec v1.x.y remains verifiable by any v1.x.* (forward-compatible patch) verifier.
- A bundle generated under spec v1.x remains verifiable by any v1.y verifier where y >= x (forward-compatible minor).
- Major version increments (v2.x) are reserved for incompatible schema changes (e.g., post-quantum signatures); v2.x verifiers retain v1.x verification capability as a deprecated codepath.
- Already-anchored events under v1 remain valid against v1 verification semantics indefinitely.

The compatibility contract is the substrate for the long-term retention story (per §7): a bundle generated in 2026 must remain verifiable in 2046 even if the spec has evolved to v3.

v2.0.0 final landed 2026-05-22 (Phase 7.F.4 promotion gate session 102): post-quantum migration via ML-DSA-65 + Ed25519 dual signing at the manifest level. The v2 amendment is documented at `docs/spec/bundle-format-v1.md` §§18.1-18.10 (the v2 spec amendment co-located with the v1 spec per file-naming-pragmatism decision — multi-version-in-single-file rather than sibling `bundle-format-v2.md`). The reference TS writer (`packages/evidence/src/generate-bundle.ts` + `generate-audit-log-bundle.ts`), Go-native verifier (`apps/cli/internal/checks/check1_v2_dual_sig.go`), and Go-WASM verifier (`apps/cli/web/nuwyre.wasm`) all implement the v2 dual-sig path. **27 conformance fixtures** (14 v1 + 13 v2) at `docs/spec/fixtures/bundle-format-v1/` empirically verify cross- language byte-equivalence per the rc → final promotion criteria at SPEC_GOVERNANCE.md §3.2. New bundles emitted after v2.0.0 publication SHOULD use the v2 path; v1 verification capability is retained indefinitely per the deprecated-codepath commitment above. Customer- facing impact: v2 bundles are quantum-resistant against the signature-layer analog of "harvest-now-decrypt-later" — an adversary who records bundle bytes today and later derives the Ed25519 private key from its public key via a cryptographically-relevant quantum computer would still face the ML-DSA-65 signature, which is designed to resist quantum attack. ML-DSA-65 is NIST FIPS 204 (finalized 2024-08-13 alongside FIPS 203 ML-KEM and FIPS 205 SLH-DSA) — one of two NIST-standardized post-quantum general-purpose digital signature schemes (FIPS 204 ML-DSA + FIPS 205 SLH-DSA).

3.13.2 SPEC_GOVERNANCE.md procedure

Spec amendments follow the procedure documented at `docs/spec/SPEC_GOVERNANCE.md`:

- Patch revisions (v1.0.x) for clarifications + bugfixes; no schema changes; existing bundles continue to verify.
- Minor revisions (v1.x) for additive capabilities + new bundle subtypes; existing bundles continue to verify; new bundles use the new capabilities.
- Major revisions (v2) for breaking changes; require migration story for already-anchored content.

Each amendment is published with: rationale, scope of change, backward-compatibility analysis, fixture-suite update, KAT vector update (if applicable), reference verifier implementation update.

3.14 Where the format lives

The authoritative artifacts:

- **Spec document**: `docs/spec/bundle-format-v1.md` (this methodology section is the prose-summary; spec doc is binding).
- **JSON Schemas**: `docs/spec/event-v1.schema.json` + `docs/spec/audit-log-event-v1.schema.json` (machine-validatable).
- **Conformance fixtures**: `docs/spec/fixtures/bundle-format-v1/` (14 fixtures: 10 customer-export + 4 audit-log-export; 1 valid + tampered variants per category; pinned bundle.zip + declared results.json per fixture).
- **KAT golden vectors**: `apps/cli/internal/checks/testdata/` (cross-language byte-equivalence pinning).
- **Reference verifier**: `apps/cli/` (Go) + WASM build at `apps/marketing/public/wasm/nuwyre.wasm`.
- **Reference writer**: `packages/evidence/` (TypeScript; produces bundles from a `BundleDataSource` abstract data interface).

A third-party implementer reading the spec + fixtures + KAT vectors should be able to produce a conformant verifier without coordination with NuWyre. This is the standards-track posture (per SPEC_GOVERNANCE §5: "multiple independent implementations are a goal").

4. Verification Procedure

This section describes how a relying party (the customer's compliance officer, the customer's outside counsel, a regulator, an opposing expert witness) independently verifies a NuWyre-issued evidence bundle without trusting NuWyre.

The verifier is open source. The cryptographic primitives are standard (SHA-256, Ed25519, OpenTimestamps, RFC 3161 PKCS#7). The external dependencies (Bitcoin chain, GitHub, commercial TSAs) are publicly fetchable. A relying party who completes verification has cryptographic evidence — not vendor attestation — that the bundle's events were ingested as recorded.

4.1 Verifier distribution

Two reference verifier implementations ship at v1:

- **`apps/cli`** — Standalone Go binary. Single static binary, no external runtime dependencies. Embeds the pinned issuer Ed25519 public keys + pinned RFC 3161 TSA trust roots at compile time. Suitable for offline / air-gapped verification environments + for CI integration. Distributed as platform-specific binaries from the NuWyre/cli GitHub releases page (V1.1+ distribution mechanism; V1 ships source-buildable from `apps/cli/cmd/nuwyre`).
- **`apps/marketing/public/wasm/nuwyre.wasm`** — In-browser WebAssembly verifier. Same Go code compiled to WASM. Distributed via the `/verify` page on `nuwyre.com`. Useful for one-shot verification by a relying party who does not want to install a CLI. Network-dependent for OTS/RFC 3161/GitHub fetches.

Both implementations consume the same spec-conformance fixture suite

- produce byte-identical JSON output for any given bundle.

4.2 Installation

The CLI is distributed as a source-buildable Go module. To build from source:

```
git clone https://github.com/nuwyre/cli
cd cli
go build -o nuwyre ./cmd/nuwyre
```

The binary is ~12 MB on Linux amd64. No external runtime dependencies.

Reproducible-build verification: the binary's SHA-256 is published alongside each release in the NuWyre/cli GitHub release notes. A relying party who wants to verify the binary itself is what NuWyre claims it is can rebuild from source + compare SHA-256s. Build reproducibility is a V1.1+ infrastructure commitment.

The WASM verifier requires no installation; navigating to `/verify` on `nuwyre.com` and dragging-and-dropping a bundle ZIP runs verification in the browser via the same Go-compiled-to-WASM code path.

4.3 The verification command

```
nuwyre verify <bundle.zip>
```

Default behavior: runs all 9 checks (enumerated in §4.4) against the bundle. Emits a human-readable report + sets process exit code per §4.6.

Common flag combinations:

```
# Default verification with online anchor checks
nuwyre verify bundle.zip

# Skip external-anchor checks (OTS, RFC 3161, GitHub) — useful in
# air-gapped review contexts.
nuwyre verify --offline bundle.zip

# JSON output for programmatic consumption (CI integration, audit
# scripts). Same checks, machine-parseable output shape per the
# results.schema.json contract.
nuwyre verify --json bundle.zip

# Opt-in flags for V1 deploy-bootstrap state:
nuwyre verify \
  --allow-dev-key \           # accept example-demo/sandbox/audit-log dev-key signatures
  --allow-pending-ots \       # accept OTS receipts without Bitcoin confirmation
  --allow-anchor-pending \    # accept GitHub anchor commits in pending state
  bundle.zip

# Strict OTS mode: refuse pending receipts; require Bitcoin attestation.
nuwyre verify --strict-ots bundle.zip
```

4.4 The 9 checks

The verifier runs nine checks in canonical sequence per spec §14.1. Each check is independent + cumulative — a failure in an earlier check does not prevent later checks from running (the per-check errors are collected; the aggregate verdict is computed at end).

Check 1 — Manifest signature

What it verifies: the `manifest.json` file's signatures in `signature.sig` match the JCS-canonicalized form of the manifest. The verifier dispatches by `bundle_format` EXCLUSIVELY (a v1 bundle that smuggled a `signing.signatures[]` array still routes to v1 verification and fails because v1 `signing` is a flat object). The `bundle_format` and `schema_version` fields MUST match per spec §2 — this coherence check runs at the §2 format-identifier dispatch boundary BEFORE Check 1 step 1.

- **v2 bundles** (current; v2.0.0 final landed 2026-05-22): `signature.sig` is parsed as a JCS-canonicalized JSON multi- signature container per spec §5 + §18.2. BOTH the Ed25519 signature (index 0; same algorithm family as per-event ingestion signatures in §2.1.4) AND the ML-DSA-65 signature (index 1; NIST FIPS 204 post-quantum) must independently verify against the pinned NuWyre issuer keys looked up via `signature.sig.signatures[i].key_id`, which the verifier cross-validates against `manifest.signing.signatures[i].key_id`.
- **v1 bundles** (legacy; locked across the v1.0.x amendment series; verifiers continue to accept indefinitely): the single Ed25519 signature (raw 64 bytes in `signature.sig`) must verify against the pinned issuer key looked up via the flat `manifest.signing.key_id`. The verifier dispatches to the appropriate pinned keys based on the bundle's `bundle_type` + `generated_at` + `bundle_format`.

What PASS guarantees: the manifest's content has not been modified since NuWyre signed it. For v2 bundles, both NuWyre's elliptic-curve key AND its post-quantum key signed the exact canonical manifest — cryptographic forgery (defeating the signing algorithms themselves) requires defeating two schemes built on different mathematical foundations. The signing keys were pinned NuWyre keys at the time of signing.

What FAIL means: the manifest's bytes have been modified after signing OR a signing key is not a pinned NuWyre key OR a signature itself was tampered OR (v2-specific) one of the two signatures does not verify. For v2 the verifier reports BOTH algorithms' verdicts via the structured `algorithm_verdicts: [{algorithm, status}, ...]` array per spec

§18.10 — operators consuming the `--json` output should read this field programmatically rather than parsing human-prose error messages.

What `warn (dev_key)` means: the bundle was signed with the development key, not the production key. For bundle types where dev keys are acceptable under `--allow-dev-key` (example-demo, sandbox-preview, audit-log-export), the warn fires if either signature's key is a dev key. For `customer-export` bundles a dev key in either position is a hard failure regardless of `--allow-dev-key`. For v2 bundles, a mixed prod-ed25519 + dev-ml-dsa-65 shape (or vice versa) is also rejected as inconsistent per the cross-environment-slot discipline at spec §18.6.

Check 2 — Artifact integrity

What it verifies: every file in the bundle's `manifest.artifacts` list has a stored SHA-256 that matches the computed SHA-256 of the file's actual bytes. The verifier iterates the artifacts list + recomputes SHA-256 over each file's bytes.

What PASS guarantees: no file in the bundle has been modified since NuWyre exported it. The bundle is byte-identical to what NuWyre signed at export.

What FAIL means: at least one file's bytes have been modified (tampered or corrupted). The error message identifies the specific file + the declared-vs-computed SHA-256 mismatch.

Check 3 — Hash chain (events.jsonl)

What it verifies: the per-event hash chain in `events.jsonl`. For each event after the first, verifies that the event's `forensic.prev_event_hash` equals the prior event's `forensic.event_hash`. Verifies that the first event's `prev_event_hash` is the genesis sentinel (64 zero hex characters). Verifies that each event's `event_hash` is correctly computed from the four-field composition per spec §6.2.

What PASS guarantees: every event in the events.jsonl chain links cryptographically to its predecessor; no event has been inserted, modified, or deleted from this chain.

What FAIL means: a chain break — either an event's `prev_event_hash` does not match the prior event's `event_hash`, or an event's stored `event_hash` does not match the recomputed hash.

What SKIPPED means (audit-log-export bundles): the events.jsonl is empty + audit-log chain integrity verification is performed at Check 9 instead. Per spec §16.2.2 + §16.3.1 empty-subtree composition, audit-log-export bundles may have empty primary-events chains.

Check 4 — Merkle proof

What it verifies: for each event in the bundle, walks the sibling-hash path from the event's leaf position to the recomputed Merkle root. Compares against the bundle's declared `daily_root` for the event's UTC date. For dual-subtree audit-log-export bundles, verifies the events subtree root (which for operator-only bundles is the zero sentinel; primary event integrity moves to Check 9).

What PASS guarantees: the Merkle tree was constructed honestly + every event is provably a leaf under the declared root. No event can be removed from the tree without changing the root.

What FAIL means: the walked root differs from the declared root. Either the proof path is forged, or the leaf event_hash is tampered, or the declared root is forged.

Check 5 — OpenTimestamps Bitcoin anchor

What it verifies: the `.ots` receipts in `ots_receipts/` are valid OTS protocol bytes; the receipt's claimed digest equals the daily root for the receipt's date; the receipt's Bitcoin attestation (if present) is independently verifiable against the public Bitcoin chain.

What PASS guarantees: the daily root was anchored in a specific Bitcoin block at the receipt's attested time. A relying party fetching the Bitcoin block independently confirms the anchor.

What `warn (pending_ots)` means: the receipt has calendar attestations only; no Bitcoin block attestation has yet landed. This is the normal state for receipts within hours of submission. Operator may opt in with `--allow-pending-ots` to fold to pass.

What FAIL means: no receipt exists for an attempted-anchor day OR the receipt's digest does not match the daily root OR the receipt's Bitcoin attestation does not reconstruct against the public chain.

What SKIPPED means (under `--offline`): the verifier did not attempt Bitcoin network fetches; opt-in by the operator means the external anchor checks are explicitly bypassed.

Check 6 — RFC 3161 TSA timestamps

What it verifies: each `.tsr` file in `rfc3161_receipts/` is a valid PKCS#7 SignedData structure; the receipt's signed digest equals the daily root for the receipt's date; the receipt's signing certificate chains to a pinned TSA trust root. Spec §11 mandates ≥2-of-3 distinct TSAs verify.

What PASS guarantees: the daily root was timestamped by at least 2 of the 3 commercial RFC 3161 TSAs at the receipts' attested times.

What FAIL means: fewer than 2 TSAs verified successfully.

Check 7 — GitHub anchor cross-check

What it verifies: each `github_anchors/<date>.json` file declares a `mirror_status` + (when anchored) a `commit_sha`. The verifier fetches the claimed commit from the public NuWyre/anchors repo + validates it contains the daily root for that date.

What PASS guarantees: the daily root is anchored in a signed public GitHub commit, independently fetchable by anyone with network access.

What `warn (anchor_pending)` means: the bundle was produced in the V1 deploy-bootstrap window where GitHub anchor commits are deferred to manual operator lift. OTS + RFC 3161 anchor legs provide independent attestation. Operator may opt in with `--allow-anchor-pending` to fold to pass.

Check 8 — Ephemeral session attestation

What it verifies: if the bundle declares ephemeral-session signing (post-V1 two-key topology), validates the per-session attestation key was signed by the production manifest-signing key.

What SKIPPED means: the bundle uses single-key signing topology (V1 default). Ephemeral-session attestation is not applicable; check is structurally inapplicable.

Check 9 — Audit-log Merkle (dual-subtree composition)

What it verifies (for audit-log-export bundles):

- The `audit_log_events.jsonl` chain integrity (recomputed event_hashes match stored event_hashes; prev_event_hash links resolve).
- The `audit_log_subtree.json` proofs (sibling-hash walks reconstruct to the declared subtree root).
- The dual-subtree composition: `daily_root = sha256(events_subtree_root || audit_log_subtree_root)`.
- The manifest-declared `audit_log_event_count` matches the actual audit_log_events.jsonl line count.

What PASS guarantees: the audit-log integrity domain is cryptographically verified end-to-end + the daily root anchors both primary-event and audit-log subtrees simultaneously.

What SKIPPED means (for customer-export bundles): the bundle is not audit-log-export type; the dual-subtree composition does not apply (or it applies trivially because the audit-log subtree root is the zero sentinel).

4.5 Reading the report

The human-readable report renders each check's status + summary:

```
$ nuwyre verify --allow-dev-key --allow-pending-ots --allow-anchor-pending bundle.zip

NuWyre Verifier v1.0
Bundle: bundle.zip (4,959 bytes)

  ✓ Check 1: manifest-signature   warn (dev_key — opted into pass)
  ✓ Check 2: artifact-integrity   pass
  • Check 3: hash-chain           skipped (audit-log-export; empty events.jsonl)
  ✓ Check 4: merkle-proof         pass
  ✓ Check 5: opentimestamps       warn (pending_ots — opted into pass)
  ✓ Check 6: rfc3161              pass
  ✓ Check 7: github               warn (anchor_pending — opted into pass)
  • Check 8: ephemeral-session    skipped (single-key signing topology)
  ✓ Check 9: audit-log-merkle     pass

VERDICT: PARTIAL VERIFICATION (exit 1)
  7 check(s) verified; 0 failed; 2 skipped; 3 warn(s) opted INTO pass via --allow-* flag
```

The JSON output (`--json` flag) emits the same information in machine-parseable form per the `results.schema.json` contract. A relying party integrating verification into CI consumes the JSON output programmatically.

4.6 Exit codes

The verifier sets process exit codes per spec §14:

- **0** — PASS. All checks verified (possibly including warns folded via opt-in flags + skipped checks under `--offline`).
- **1** — FAIL or PARTIAL_VERIFICATION. Either at least one check definitively failed (FAIL; terminal) OR at least one check is warned/skipped without operator opt-in (PARTIAL_VERIFICATION; the operator can re-run with the appropriate flag to opt INTO pass).
- **2** — INVOCATION_ERROR. The verifier could not run — typically a malformed CLI invocation, missing bundle file, or unparseable bundle ZIP structure.

CI integrations should treat exit 1 as a verification problem requiring investigation; exit 0 as a clean verification; exit 2 as a tooling problem requiring operator attention.

4.7 Offline mode

The `--offline` flag skips checks 5/6/7 (the external-anchor checks that require network fetches). Useful for:

- Air-gapped verification environments (regulator's evidence-review workstation).
- Pre-deployment validation of bundles in CI without flaky network dependencies.
- Time-bounded verification when network access is slow or intermittent.

Under `--offline`, the verifier emits SKIPPED for checks 5/6/7 and the aggregate verdict reflects the partial verification state. Offline-verified bundles do NOT have anchor-leg cryptographic attestation; they have only manifest signature + artifact integrity

- chain integrity + Merkle proof verification. A relying party wanting full anchor verification must re-run online with network access.

4.8 Building from source

Organizations requiring source-built verifiers (regulatory requirement; security-policy compliance; reproducible-build verification) follow:

```
git clone https://github.com/nuwyre/cli
cd cli
go test ./internal/...          # run the unit + KAT + conformance test suite
go build -o nuwyre ./cmd/nuwyre # produce the binary
./nuwyre verify <bundle.zip>    # use the freshly-built binary
```

The full test suite includes:

- Per-check unit tests at `internal/checks/`.
- KAT (Known Answer Test) golden vectors at `internal/checks/testdata/audit_log_kats_v1.json` validating cross-language byte-equivalence with the TypeScript reference.
- Conformance fixture suite at `docs/spec/fixtures/bundle-format-v1/` exercising 14 fixtures (10 customer-export + 4 audit-log-export; 1 valid + tampered variants per category).

A successful test suite + a clean build produces a verifier whose outputs are byte-identical to the official NuWyre/cli release for any given input bundle. Reproducibility verification is the relying-party-side defense against verifier-binary substitution.

4.9 What verification establishes

A bundle that passes all 9 checks (default flags; pending-OTS + anchor-pending opt-ins acceptable for V1 deploy-bootstrap state) provides the following cryptographic claims:

1. The bundle's manifest was signed by a NuWyre issuer key whose public component is pinned in the verifier binary.
2. Every artifact in the bundle is byte-identical to what NuWyre signed at export.
3. The per-event hash chain is internally consistent + extends back to the genesis sentinel.
4. The Merkle proof for each event reconstructs to the declared daily root.
5. The daily root is anchored in the Bitcoin blockchain via OpenTimestamps (verified against the public chain) OR is in pending state acknowledged by the operator's opt-in flag.
6. The daily root is timestamped by ≥2-of-3 commercial RFC 3161 TSAs.
7. The daily root is recorded in a signed GitHub commit in the public NuWyre/anchors repository (or is in anchor-pending state acknowledged by the operator's opt-in flag).
8. (For audit-log-export bundles) the audit-log chain integrity is verified + the dual-subtree composition matches.

Each of these claims is independently re-verifiable by the relying party. NuWyre's role is to produce the bundle + publish the verifier + maintain the anchor pipeline. The verification is mechanical + reproducible + requires no NuWyre co-operation. This is the operational embodiment of the "tamper-evident, not trust-NuWyre" posture.

4.10 What verification does NOT establish

Per §1 + §9, verification confirms what NuWyre observed at ingestion and that the chain is tamper-evident. It does NOT confirm:

- The truthfulness of the upstream AI agent's outputs (NuWyre attests to ingestion, not to upstream behavior).
- The correctness of any compliance evaluation produced by the policy pack (the evaluation is itself an event with its own chain position; the evaluation's correctness is a separate methodology question per §5).

- Anything that happened before NuWyre's server observed the event at ingestion (clock-skew on the customer's side; voice-platform- side recording integrity; AI-agent-side prompt-faithfulness).

A relying party using verification as part of a regulatory or litigation defense uses the verified bundle as ONE evidentiary component, combined with customer-side hygiene artifacts (call recording chain-of-custody, agent attestation logs, deployment-time audit trails) for complete forensic reconstruction.

5. Policy Evaluation Methodology

This section documents how NuWyre evaluates a stream of EventV1 records against a policy pack. The model is: each pack rule pairs a deterministic **trigger** (object-tree DSL with closed predicates) with an **LLM evaluator** that produces a verdict + severity + reasoning. Triggers are side-effect-free; evaluators are gated to deterministic settings (`temperature=0`, model version pinned).

5.1 Pack format (v1)

A pack is a directory:

```
<pack-id>/
  pack.yaml            -- metadata, rules, trigger trees, evaluator config
  prompts/
    <rule-name>.md     -- LLM prompt template per rule
  schemas/
    <rule-name>.json   -- JSON Schema for the LLM's structured output
  fixtures/
    fixtures.json      -- paired fire / no-fire trigger contexts
```

`pack.yaml` declares: `id` (`<namespace>-<short-name>-v<major>`), `version` (semver), `compatibility` (event_schema_version, pack_format_version), `authoring`, `applies_to.industries`, and `rules[]`.

Each rule has `id` (`namespace.rule_name`), `severity`, `title`, `description`, `trigger`, `evaluator`, and `cross_validation`.

The trigger DSL is a recursive object tree:

- **Boolean groups**: exactly one of `all` / `any` / `not` per node.
- **Field predicates**: an object with a `field` path and exactly one of the 14 closed predicates (`equals`, `not_equals`, `in`, `not_in`, `contains`, `contains_any`, `contains_all`, `exists`, `not_exists`, `greater_than`, `less_than`, `greater_or_equal`, `less_or_equal`, `length_greater_than`, `length_less_than`).
- **Field paths**: dot-separated identifiers with optional `[filter]` (`key=value AND other=value`) or positional (`first` | `last` | integer) segments. No regex, no temporal operators (those are LLM judgment, not trigger conditions).

Loader rejects anything outside this grammar, including same-family cross-validators (see 5.5).

5.2 body_hash

Every pack receives a `body_hash` computed at load time as `sha256_hex(canonicalize(...))` over the canonical JSON of:

```
{
  "pack_format_version": "1",
  "pack": <pack metadata, JCS-canonicalized>,
  "prompts": { "<rule-id>": "<sha256-hex of prompt UTF-8 bytes>" },
  "schemas": { "<rule-id>": "<sha256-hex of schema JSON UTF-8 bytes>" }
}
```

Any change to pack metadata, any prompt template, or any output schema changes the `body_hash`. The hash is recorded on every evaluation row, so the exact pack revision used to produce a result is recoverable years later without trusting the customer-facing version string.

5.3 Evaluator runtime

The evaluator runtime exposes two surface methods (`runSync`, `enqueue`) and is constructed with the rules it will evaluate. In production mode it **refuses to register rules with non-zero `temperature`** — determinism is not an opt-in; it is a registration-time gate.

For each evaluation, the runtime:

1. Loads the rule's prompt template and output schema from the in-memory pack cache (no disk access at evaluation time).
2. Renders the template by substituting a closed set of placeholders (`{{ rule.id }}`, `{{ rule.title }}`, `{{ rule.description }}`, `{{ rule.severity }}`, `{{ session.id }}`, `{{ session.events }}`, `{{ agent.industry }}`, `{{ event.content_hash }}`). Unknown placeholders abort with a load-time-style error so typos in pack prompts are caught before billing an LLM call.
3. Computes `prompt_template_hash` (SHA-256 of the template bytes) and `resolved_prompt_hash` (SHA-256 of the post-substitution string).
4. Calls the LLM provider with the pinned model + temperature + max tokens + JSON Schema for structured output.
5. Records the raw output byte-for-byte alongside the parsed verdict.

Returned `EvaluationResult` carries: `rule_id`, `verdict`, `severity`, `reasoning`, `raw_output`, `model_family`, `model_version`, `evaluator_runtime_version`, `prompt_template_hash`, `resolved_prompt_hash`, `latency_ms`. On provider exception the runtime returns an `errored: true` result with `verdict=uncertain` rather than throwing — evaluation is best-effort and downstream code expects a result, not an exception.

5.4 Pack-event compatibility (D7)

A pack declares the event schema version it was authored against. Today the runtime supports `event_schema_version: 1` and `pack_format_version: 1`; loading a pack with a different value is a hard reject at load time.

This is intentional. The trigger DSL references event field paths (`events[role=assistant].content`, `events[role=tool_call].metadata.fn`). If the event schema changes — fields renamed, removed, or restructured — existing triggers can silently change meaning or stop firing. Rather than guess at compatibility, we require packs to declare what they were authored against and refuse incompatible pairs.

When the event schema version increments (a hypothetical event-v2):

1. The runtime registers support for both v1 and v2 (a transition window — never an overlapping silent migration).
2. Existing packs declaring `event_schema_version: 1` continue to load and evaluate against v1-shaped data only.
3. Pack authors update prompts and triggers to v2, bump `event_schema_version` to 2, and bump the pack's `version` major.
4. Customers explicitly opt in to v2 packs after evidence-driven regression testing against their fixture suite.

The compatibility key is also part of `body_hash` input (via the pack metadata), so a pack that bumps from v1 to v2 produces a new `body_hash` even when no other content changes. Auditors can prove which event-schema era a given evaluation belongs to from the recorded hash alone.

This pin is intentionally stricter than range matching. Trigger DSL paths are an unbounded surface; we cannot statically prove a v1 pack is safe against v2 data without re-running the fixture suite. The hard reject forces that re-validation to happen explicitly, by a human, on a known pack revision.

5.5 Cross-validation

Per-rule cross-validation is opt-in (`cross_validation.enabled: true`). When enabled, the evaluator runs the same context through a validator model from a **different family** (same-family validators are rejected at pack load time per D3.3 — they don't add independence).

Agreement classification is strict (D6.1):

- `full_match` — same verdict and same severity → result resolves to the primary verdict.
- `verdict_match_severity_disagreement` — same verdict, different severity → result resolves to `uncertain`.
- `verdict_disagreement` — different verdict → result resolves to `uncertain`.
- `validator_error` — validator threw or errored → result resolves to `uncertain`.

No adjacent-severity tolerance in v1. If production data shows >30% uncertain rate across the validation suite, we revisit the strictness; healthy target is 5-10%. The disagreement record retains both primary and validator results in full (verdict, severity, reasoning, model family, model version, runtime version) so downstream review can audit the disagreement without replaying the LLM calls.

5.6 Re-evaluation

Every evaluation is an append-only row. A pack version bump or a hand re-run produces a NEW row, never an overwrite. The combination of `pack_id + body_hash + rule_id + content_hash + evaluator_runtime_version` uniquely identifies a result; the system can serve "which pack revision flagged this turn" by joining on `body_hash`.

5.7 Pack-validation suite binding

Pack rules and validation suite scenarios share canonical rule identifiers. When a scenario expects `hipaa.identity_verification_before_phi` to fire, it does so because the same identifier exists as a pack rule, evaluated by the same code path. Divergence between scenario expectations and pack rules is a load-time error, not a runtime question. This binding is what allows validation results to be reported alongside production claims with intellectual integrity.

The binding is mechanical: the pack test suite (`packages/policy/tests/ packs.test.ts`) walks `validation/scenarios/<regime>/*.yaml` and asserts that every active scenario's `expected.primary_flag.rule_id` is a real rule in one of the loaded packs. Orphaned scenarios (referencing rules no pack ships) and orphaned rules (rules no scenario validates) both fail review. The validation suite is the single source of truth for both correctness and regression; pack tests are not allowed to maintain a parallel fixture library.

When the trigger of a primary rule fires for a scenario where `expected.any_flag === true`, the trigger gate is satisfied. When `any_flag === false`, the trigger MAY fire — triggers are intentionally over-inclusive on temporal/contextual judgments that the trigger DSL forbids (D1.3). The LLM evaluator clears those at evaluation time. Asserting trigger no-fire on `any_flag === false` would force triggers into the LLM's job; instead, only triggers that gate on concrete metadata predicates (e.g., `consent_on_file`, `do_not_call_record`) are asserted to honor `forbidden_flags`.

5.8 Two-layer evaluation: triggers and evaluators

NuWyre evaluates events in two layers. The trigger DSL is a fast, deterministic, structural first pass that identifies candidate events for evaluation. The evaluator is a slower, model-based, reproducible second pass that produces the verdict. Triggers are intentionally over-inclusive in regimes where the substantive judgment requires session context the trigger cannot capture (e.g., temporal ordering, tone, contextual ambiguity). The evaluator handles those cases with appropriate prompt context and a structured-output schema.

This separation has two consequences for system claims. First, trigger-layer behavior is fully auditable: the predicate vocabulary is closed, the path grammar is bounded, and trigger evaluation is deterministic and idempotent. Second, evaluator-layer judgment quality is empirically measured against the Validation Suite, with current performance reported in §5.7. Compliance officers and forensic reviewers can audit the trigger layer mechanically and assess the evaluator layer through the Validation Suite's published methodology.

5.9 Limits of LLM-as-judge (documented honestly)

- LLM evaluators are non-deterministic by default. We pin temperature and model version, but identical inputs may still produce different outputs on rare occasions — record the raw output so the variance is visible rather than hidden.
- Evaluators see only the events you provide. If a rule needs context outside the session window (prior 90-day account history), the trigger must surface that context as an event the LLM can read.
- Evaluators don't fact-check claims. A pack that asks "did the assistant make a guarantee" is reliable; a pack that asks "was the guarantee factually correct" is not.
- Cross-validation is the strongest defense against a single model's systematic blind spot. It is not a defense against both models having the same blind spot. For high-stakes rules, fixture coverage and human spot-review remain mandatory.

6. Detection and Response

This section documents how NuWyre transforms ingested events into actionable findings — the layer that sits above the integrity chain (§2) and translates "what was recorded" into "what requires operator or customer attention." Detection is the bridge between the evidence ledger and operational compliance: the chain preserves events forever; detection determines which events warrant immediate investigation.

Detection in NuWyre operates at two distinct layers:

- **Integrity detection** (§6.1) — chain-level anomalies surfaced by the cryptographic substrate (sequence gaps, signature failures, content_hash divergence, missing daily root, anchor mismatches). These detect TAMPERING with the evidence itself.
- **Compliance detection** (§6.2-§6.5) — policy-pack rule evaluation producing verdicts on EVENT CONTENT. These detect compliance violations in the underlying AI agent's behavior.

Both layers feed into the response pipeline (§6.6) which dispatches notifications + escalates findings + records the operational audit trail.

6.1 Integrity detection

The cryptographic substrate detects tampering automatically via the verifier checks (per §4.4) and operationally via continuous monitoring at the operator side.

6.1.1 Chain-level anomalies

The ingestion pipeline + the periodic chain-state-verification cron detect:

- **Sequence gaps**: a missing sequence_number in the per-org chain. Detected at: (a) ingestion-time SECURITY DEFINER RPC's optimistic- concurrency-control loop, which assigns gap-free sequence_numbers; (b) periodic chain-state-verification cron that walks the chain + flags gaps. Operator-side alarm fires within minutes.
- **Signature failures**: an event whose `ingestion_signature` does not validate against the pinned issuer key for the event's `generated_at` window. Detected at: bundle export time (Check 1); per-event re-verification on bundle generation. Operator-side alarm fires immediately.
- **content_hash divergence**: an event whose stored `content_hash` does not equal the canonical-form recomputation. Detected at: bundle export time (Check 3); periodic chain-state-verification cron. Operator-side alarm fires within minutes.
- **prev_event_hash break**: an event whose `prev_event_hash` does not equal the prior event's `event_hash`. Detected at: bundle export time (Check 3); periodic chain walk. Operator-side alarm fires within minutes.
- **Anchor mismatch**: the daily root's anchored value (in OTS / RFC 3161 / GitHub) does not equal the local-computed daily root for the date. Detected at: bundle export time (Checks 5/6/7); daily anchor-pipeline cron post-anchor-submission. Operator-side alarm fires when discrepancy is observed.

6.1.2 Operational response to integrity anomalies

When integrity detection fires, the operational response is:

1. **Freeze**: the affected daily root is marked "investigation-in-progress" + new event ingestion for the affected org pauses (or is mirrored to a separate audit table) pending investigation.
2. **Preserve**: forensic copies of the affected DB rows + bundle exports + audit-log entries are captured to an offline forensic archive. PRESERVATION TAKES PRIORITY OVER NEW INGESTION.

3. **Investigate**: operator-led investigation of root cause:
    - Was this a hash chain implementation defect (verifier regression)?
    - Was this a deliberate tampering attempt (insider; external attacker)?
    - Was this an upstream data corruption (DB-layer issue; PostgreSQL block corruption; backup-restore mishap)?
4. **Notify**: customer notification within the operator manual's stated timeline (§6 — typically 24 hours for confirmed tampering; 72 hours for inconclusive findings). Customer notification includes:
    - The specific anomaly observed.
    - The detection mechanism that surfaced it.
    - The forensic-archive snapshot identifier.
    - The blast radius (which events, which dates, which orgs affected).
    - The next-step procedure.
5. **Resolve**: per the investigation outcome:
    - **Implementation defect**: code fix + verifier release + customer notification with re-verification path.
    - **Confirmed tampering**: forensic preservation + law enforcement escalation (if criminal jurisdiction applies) + customer-led legal action support.
    - **Upstream corruption**: backup-restore + integrity re-validation against pre-corruption anchored roots + audit-log entry.

The audit trail for the entire freeze → preserve → investigate → notify → resolve cycle is itself recorded in the `audit_log_events` chain (per spec §16), making the response procedure itself tamper-evident.

6.2 Compliance detection: policy packs

Policy-pack rule evaluation per §5 produces compliance findings on event content. The pack format pins:

- **Trigger**: object-tree DSL with closed predicates (§5.1).
- **Evaluator**: LLM with deterministic settings (§5.3).
- **Output schema**: JSON Schema constraining the LLM's response shape (§5.4).
- **Cross-validation**: optional cross-rule cross-check for HIGH/ CRITICAL severity findings (§6.4).

6.2.1 Trigger-as-deterministic-filter

Triggers fire on every event matching the rule's object-tree predicate. Triggers are side-effect-free: they do NOT produce evaluation output; they only select events for evaluator consideration.

The deterministic-trigger pattern serves three purposes:

1. **Performance**: most events don't match any rule's trigger; skipping evaluator invocation on non-matching events is the cost-control substrate (each evaluator invocation costs an LLM API call).
2. **Auditability**: a regulator can independently verify "did this event match this rule's trigger?" by re-running the trigger against the event content. The trigger DSL is closed-vocabulary
    - side-effect-free, so independent re-execution produces the same result.
3. **Reproducibility**: a re-run of evaluation on a stored event produces the same trigger-fire-or-not decision, supporting the "regenerate evaluations deterministically" property per §5.

6.2.2 Evaluator-as-judgment-layer

When a trigger fires, the rule's evaluator is invoked with:

- The event content (canonicalized).
- The rule's prompt template (per §5.4).
- The rule's output schema (per §5.4).
- The pinned model version (deterministic settings: temperature=0, pinned model_id).

The evaluator produces a structured response conforming to the output schema: verdict (clean | flagged | error) + severity (info | low | medium | high | critical) + reasoning (LLM-generated structured text). The response is recorded as an evaluation row in `evaluations.jsonl`.

Determinism guarantees:

- Same event + same rule + same model version + temperature=0 → same evaluator output (modulo extremely rare model non- determinism that NuWyre measures + reports per §5.7).
- The `pack_body_hash` (per §5.2) pins the exact rule revision used; a relying party can reconstruct which rule produced which verdict.

6.3 Severity taxonomy

Evaluations carry a closed-vocabulary severity:

- **info** — informational; pattern matched but no compliance concern. Surfaces in dashboards for operator visibility; no notification.
- **low** — minor pattern matched; review at convenience. Surfaces in daily digest emails; no immediate alert.
- **medium** — pattern matched warrants review within the same business day. Surfaces in dashboards + queued for next-day-digest; may trigger configurable per-customer notification.
- **high** — pattern matched warrants prompt operator attention; triggers notification per the customer's configured channel (Slack, Teams, SMS, email) within the V1 dispatch-cadence window (see §6.5.1).
- **critical** — pattern matched indicates a likely compliance violation requiring action; triggers notification + cross-validation cascade (§6.4) + optional agent suspension (§6.5). Same V1 dispatch-cadence window as HIGH.

Severity is determined by the LLM evaluator per the rule's prompt specification. The severity ladder is enforced at the pack-loader boundary: rules that omit severity OR specify out-of-vocabulary severity are rejected at load time.

6.4 Cross-validation for high-severity findings

When a rule fires at HIGH or CRITICAL severity, NuWyre's evaluation pipeline triggers cross-validation: a second evaluator (different rule from a different family; or the same rule under a different pinned model version) re-evaluates the same event content.

The cross-validation contract:

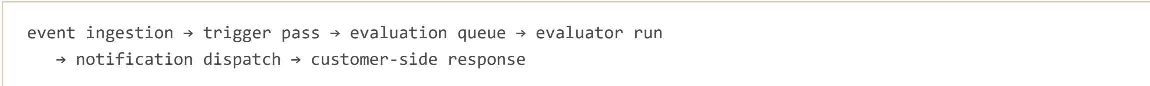
- **Agreement**: both evaluators reach the same verdict + severity. The finding is recorded with `cross_validation.agreement: "agree"`. Notification fires.
- **Disagreement**: evaluators reach different verdicts. The finding is recorded with `cross_validation.agreement: "disagree"` + the disagreement details (each evaluator's verdict + reasoning). Notification fires WITH the disagreement surfaced — the operator reviews the disagreement before deciding response.
- **Cross-validator error**: the cross-validator returns an error (LLM API failure, schema validation failure). The original finding fires notification without cross-validation; the `cross_validation.agreement: "n/a"` records the cross-validator unavailability.

Cross-validation is NOT a vote-counting mechanism that suppresses findings. A disagreement does NOT cancel the original finding — it surfaces the disagreement for human review. The principle: disagreements are MORE interesting than agreements (per §5.8); they indicate either an evaluator ambiguity OR a rule-coverage gap.

The cross-validation evaluator family selection prevents same-family cross-validators (which would correlate failures): a TCPA rule cannot cross-validate a TCPA rule; cross-validation must span rule families. The pack-loader enforces the family-separation constraint at load time.

6.5 Async pipeline mechanics

The detection pipeline operates asynchronously to avoid blocking the ingestion hot path:



```
event ingestion → trigger pass → evaluation queue → evaluator run
   → notification dispatch → customer-side response
```

6.5.1 Latency targets (V1 actual)

The V1 detection pipeline is implemented as a chain of three Vercel Cron jobs, each polling every minute. End-to-end latency is therefore bounded by cron-polling cadence, not real-time event dispatch:

- **Event ingestion → trigger pass**: in-process at the ingestion handler; <50ms typical. Triggers are deterministic side-effect- free predicate evaluations per §6.2.1.
- **Trigger fire → evaluation queued**: trigger match writes a row to the `evaluations` table with `status='pending'`; in-process, <10ms.
- **Evaluation queued → evaluator run**: the `evaluate` cron (every 1 min) claims pending evaluations + invokes the LLM evaluator + records verdict + severity + reasoning. Per-event latency: 60s worst-case waiting for cron tick + ~2-10s LLM call.
- **Evaluation recorded → firing produced**: the `produce-notification-firings` cron (every 1 min) polls evaluations matching enabled notification_rules + INSERTs `notification_rule_firings` rows. Per-firing latency: 60s worst-case waiting for cron tick.
- **Firing produced → channel dispatched**: the `dispatch-notification-firings` cron (every 1 min) polls firings without dispatch rows + invokes Slack / Teams / email / webhook helpers + records per-channel dispatch rows. Per-dispatch latency: 60s worst-case waiting for cron tick + ~1-5s for the external channel call (Slack/Teams webhook latency).

End-to-end V1 latency (event ingest → notification arrives):

- **Best case**: ~30-60 seconds (cron timings align favorably)
- **Typical**: ~90-120 seconds (median)
- **Worst case**: ~3 minutes (all three crons run just before the event lands)

This cadence is appropriate for HIGH/CRITICAL compliance findings, which require operator review + judgment — not sub-second blocking intervention. For real-time blocking (e.g., halt-call-mid-stream on a TCPA violation), customers should pair NuWyre with a guardrail tool that operates synchronously at the agent-runtime layer; NuWyre records what the guardrail does + provides the evidence trail.

V1.1+ scope — true sub-second dispatch (event-driven evaluator

- in-process notification dispatch on HIGH/CRITICAL evaluations) is a future enhancement, deferrable until customer signal indicates the 1-3 minute cadence is materially insufficient. Cron-polling cadence is the simpler operational substrate + has predictable cost characteristics.

Real-mode dispatch is operationally gated on:

- `NODE_ENV='production'`
- `SEND_REAL_NOTIFICATIONS='true'` environment variable
- Per-rule channel configuration (`notification_rules.channels` with validated Slack webhook URLs, Teams webhook URLs, email recipient lists, etc.)

Outside of real-mode (dev / staging / test), the dispatch helpers log the intended dispatch + return `{skipped: true}` + still write the dispatch row (status='skipped') so the operator audit trail records the would-have-fired-but-disabled state.

The pipeline does NOT block on Bitcoin OTS confirmation — notifications fire on evaluator output, not on anchor confirmation. The anchor confirmation is the integrity-evidence backstop, not the notification trigger. A customer receives a notification within the V1 dispatch-cadence window above; the same finding is anchored to Bitcoin within ~24 hours via the next-daily-root cycle.

6.5.2 Pipeline durability

The cron-polled pipeline is durable: pending evaluations + pending firings + pending dispatches survive Vercel function restarts, short-term LLM-API outages, and short-term external-channel outages. The pipeline's at-most-once semantics per cron tick + the idempotency patterns at each stage (one evaluation per `(event_id, rule_id, model_version)`; one firing per rule-trigger- window; one dispatch row per `(firing_id, channel)`) prevent duplicate work across overlapping cron ticks.

If the LLM API is completely unavailable for an extended period (>1 hour), the `evaluations` table accumulates `pending` rows. The operator-side observability dashboard surfaces pending-evaluation count + LLM API success rate; sustained backlog growth triggers operator investigation.

Failed dispatches stay in `status='failed'` and are NOT re-dispatched on subsequent cron ticks in V1. Operator-triggered retry is V1.1+ work. The dispatch failure is recorded with error details for the operator to investigate (typically a misconfigured channel URL, an external provider outage, or a recipient address that bounced).

6.6 Notification channels

Customer-configured notification channels for high-severity findings:

- **Slack**: webhook URL + per-rule channel mapping. Slack message contains the event_id + rule_id + severity + reasoning summary + link to the evaluation in the customer dashboard.
- **Microsoft Teams**: webhook URL per the Teams Incoming Webhook spec. Same payload as Slack.
- **Email**: per-rule recipient list. Email contains the same structured payload + a link to the evaluation.
- **SMS**: per-rule phone number list (Twilio Programmable SMS). SMS contains a one-line summary + a short-link to the evaluation.

Each channel has documented delivery semantics:

- **Slack/Teams**: at-most-once delivery; failure (HTTP error from the webhook endpoint) is recorded in the `notification_delivery_log` table for operator visibility + manual re-send.
- **Email/SMS**: at-least-once delivery via the upstream provider (Resend for email; Twilio for SMS). Provider-side delivery reports are captured in the delivery log.

Notification failure does NOT suppress the finding: the evaluation row remains in `evaluations.jsonl` with the recorded verdict + severity, regardless of notification delivery state. A notification-failed finding surfaces in the operator's notification-delivery dashboard for manual response.

6.7 Agent suspension

When a notification rule configures `suspend_agent: true` AND a CRITICAL severity finding fires, the response layer suspends the offending agent at the API gateway:

1. The agent's `api_key` is marked `suspended_at: <ISO 8601 UTC>` in the database.
2. Subsequent `/v1/events` requests for the suspended agent return `403 Forbidden` with a structured `agent_suspended` error code.
3. The suspension event is recorded in `audit_log_events` with:
    - The triggering rule_id + evaluation_id.
    - The suspended agent's identifier + organization_id.
    - The suspending principal (the notification rule itself, with attribution to the customer's configured rule).
4. The customer's notification includes the suspension state alongside the finding.

Suspension is **reversible by the customer's operator**: a designated customer-role user can lift suspension via the customer portal. The lift is itself recorded in `audit_log_events`. There is no NuWyre-side override of the customer's suspension decision; the customer controls their own agent operational state.

6.7.1 When NOT to use agent suspension

Agent suspension is a heavy-handed response. It should be reserved for findings where continuing operation poses serious harm:

- HIPAA PHI leak in real-time agent output.
- TCPA pre-call-consent violation at scale.
- Confirmed prompt-injection attack producing unsafe agent behavior.

For lower-severity findings, the appropriate response is: notification + human review + targeted intervention (prompt adjustment, agent retraining, customer-side process change). Agent suspension is a circuit-breaker, not a routine response. Customers configuring `suspend_agent: true` should pair it with a runbook for the agent-suspended state (typically: failover to human-staffed intake; customer notification with cause + ETA for resumption).

6.8 Post-detection forensic preservation

When a finding fires AND the customer determines the finding is material to a pending or potential regulatory matter (litigation hold; subpoena; regulatory investigation), the operational preservation procedure:

1. **Hold flag on event(s)**: the affected events receive a `legal_hold: true` flag at the compliance_metadata layer (per §7). The retention sweep cron honors the hold and does not tombstone or archive the events.
2. **Bundle export at hold-time**: a customer-export bundle is generated containing the held events + their evaluations + their anchor proofs. The bundle's `manifest.daily_roots[]` includes the per-day Merkle roots for the affected dates.
3. **Off-system preservation**: the customer's compliance/legal team preserves the bundle in their own controlled storage (the customer's discovery-archive systems; offsite immutable storage; counsel's evidence repository). Customer-side hash-recording at receipt time (per §9.6.1) is the substrate for chain-of-custody documentation.
4. **Audit trail entry**: the legal-hold application is recorded in `audit_log_events` with: who applied the hold, when, the triggering finding(s), the bundle_id exported.
5. **Subsequent operations**: any operation against the held events (re-export, additional finding, NuWyre-side action) is recorded in `audit_log_events`. The customer's discovery preservation remains valid as long as the customer's off-system bundle is intact.

The forensic preservation procedure intentionally hands control to the customer: NuWyre's role is to provide the cryptographically- preserved evidence + the operational hooks; the customer's counsel decides how preservation interacts with their specific legal posture.

6.9 Operator-side observability

NuWyre operators maintain visibility into the detection + response pipeline via:

- **Per-customer dashboards**: real-time view of finding rates + severity distribution + cross-validation agreement rates + notification delivery success.
- **Aggregate metrics**: pipeline latency percentiles + LLM API success rates + queue depth + per-rule precision/recall (against fixture sets per §5.7).
- **Alert thresholds**: operator-configured alarms on sustained pipeline-latency degradation; LLM-API failure-rate spikes; notification-delivery failure spikes.

The observability substrate is documented at the operator manual §9. Customers do NOT have access to operator-side observability metrics for other customers; cross-customer aggregate metrics are documented at the methodology PDF's appendix for trust transparency.

6.10 Boundaries

What detection + response does NOT do:

- **Does not make compliance decisions FOR the customer**. The evaluator surfaces findings; the customer's compliance officer decides response. NuWyre does not unilaterally pause an agent without the `suspend_agent: true` configuration; does not unilaterally notify regulators; does not unilaterally take legal action.
- **Does not replace human review on high-stakes findings**. Per §9.3, the LLM evaluator's judgment is bounded by model + prompt
  - fixture coverage. High-stakes findings (CRITICAL severity; regulator-relevant patterns) require human review by the customer's compliance team.
- **Does not guarantee zero false positives or false negatives**. Per §5.7, each rule has measured precision + recall on its fixture set. Customers deploying rules in production accept the measured error rates; ongoing monitoring of evaluator-vs-human- reviewer agreement is the customer-side discipline.

The detection + response layer is the actionable surface above the evidence ledger. The ledger preserves; the detection + response flags for attention. The customer + their counsel + their compliance team decide how to act.

7. Retention and Legal Hold

This section documents NuWyre's retention model: how long events are kept, under what classes, under what legal-hold semantics, and how retention interacts with the cryptographic integrity chain. The retention model is operationally significant — it determines whether a 2026 event is available to verify in 2034, and under what conditions it might no longer be.

A note on legal framing. The technical mechanics documented below are NuWyre-implementation-specific. The legal interpretation of retention obligations under specific regulatory frameworks (TCPA, HIPAA, GDPR, state recording laws, broker-dealer recordkeeping) is the customer's counsel's responsibility. NuWyre's role is to provide configurable retention enforcement + auditable retention operations; the customer's role is to determine which retention class applies under which regulatory framework for which class of event content.

7.1 Retention class taxonomy

Each event carries a `compliance_metadata.retention_class` field at ingestion time. The closed-vocabulary enum:

- **`default`** — standard retention; events deleted after the retention-period default for the customer's configured tier (typically 7 years for V1; configurable per customer-specific agreement).
- **`extended`** — extended retention; events retained beyond default for customer-controlled reasons (high-value engagement; customer-specific regulatory requirement; pending litigation preservation).
- **`litigation_hold`** — immutable until hold released; events cannot be tombstoned, deleted, or archived while this flag is set. Hold-release procedure documented at §7.5.
- **`pii_minimized`** — PII-bearing events that should be tombstoned per data-minimization principles when retention period expires. Per §2.9, tombstoning preserves chain integrity while removing retrievable content.
- **`phi_minimized`** — PHI-bearing events under HIPAA data-minimization; same tombstone semantics as pii_minimized but separate operational class for HIPAA-specific audit-log attribution.

The retention class is captured at ingestion time + persisted with the event in the database. Subsequent retention-class changes are audit-logged: the change history is itself a recorded event in `audit_log_events`, making retention reclassification tamper-evident.

7.1.1 Closed vocabulary at the storage layer

The retention_class enum is enforced via Postgres CHECK constraint at migration `20260526000000_phase_6_2_b_e_retention_sweep_substrate. sql`:

```
retention_class_at_archive text not null
  check (retention_class_at_archive in ('7y','10y','indefinite'))
```

The same closed vocabulary applies at the `events` table + `audit_log_events` table + `operator_archive_log` table. An attempt to write an out-of-vocabulary retention_class fails at the DB constraint layer — defense against application-layer-only enforcement gaps.

7.1.2 Retention class semantics by class

CLASS	DEFAULT RETENTION PERIOD	HOLD-APPLICABLE?	TOMBSTONE-ON-EXPIRY?	CUSTOMER-CONFIGURABLE?
`default`	7 years (V1)	Yes (lift to `litigation_hold` class)	No (hard-delete after grace period)	Per-customer config
`extended`	Customer-specified	Yes	No	Yes
`litigation_hold`	Indefinite until hold released	N/A (already held)	Never (until reclassification)	Yes (set + release)
`pii_minimized`	7 years (configurable)	Yes (delays tombstone)	Yes (preserves chain hash)	Per-customer config
`phi_minimized`	6 years (HIPAA-aligned default)	Yes (delays tombstone)	Yes (preserves chain hash)	Per-customer config

The retention period values above are V1 defaults; production deployments configure per-customer values per the customer's specific regulatory profile + executed agreement.

7.2 Per-organization, per-jurisdiction policy expression

Different customers operate under different regulatory frameworks + multiple frameworks may apply simultaneously to a single event stream. The retention policy expression:

- **Default class** at the customer level (e.g., "all events for this customer default to retention_class='default' with 7-year retention").
- **Override class** per agent (e.g., "events from the HIPAA-context agent default to retention_class='phi_minimized'").
- **Override class** per event-pattern (e.g., "events matching the voice-consent-capture trigger receive retention_class='extended' for 10 years").
- **Manual override** via the customer portal for specific events (e.g., "this specific event_id receives retention_class='litigation_ hold' pending the X v. Y matter").

The expression layering is evaluated at ingestion time: per-event explicit overrides win over per-pattern, which wins over per-agent default, which wins over per-customer default. The resolved retention_class is recorded on the event row + cannot be silently changed without audit-log attribution.

7.2.1 Cross-jurisdiction conflicts

When multiple jurisdictions apply (e.g., a call recording captured in a two-party-consent state with one party in a one-party-consent state; a HIPAA-covered entity operating in GDPR-applicable EU member state):

- **Longer retention wins for hold semantics**: if any applicable framework mandates retention, the longest-applicable period applies. A 10-year SEC broker-dealer record cannot be deleted at the 7-year GDPR data-minimization milestone.
- **Shorter retention wins for deletion semantics**: if any applicable framework mandates deletion (e.g., GDPR right-to-erasure for a specific data subject), the shortest-applicable period applies — with the exception of legal-hold overrides per §7.5.
- **Tombstone reconciles the conflict**: when retention mandates removal but other frameworks require ability to attest "this event existed," tombstone semantics (per §2.9) preserve the chain hash + cryptographic existence proof while removing retrievable content. A relying party post-tombstone can attest "an event with this content_hash existed at this position in the chain" without being able to retrieve the original content.

The cross-jurisdiction analysis is the customer's counsel's responsibility; NuWyre provides the operational mechanics + the audit-log surface. Counsel-led decisions are recorded in `audit_log_events` with the deciding-principal attribution.

7.3 Legal hold semantics

A legal hold flags one or more events as exempt from retention deletion + tombstoning. While held:

- The retention-sweep cron does NOT tombstone or delete the held event(s).
- The event(s) remain fully retrievable via bundle export.
- Subsequent attempts to apply a more-restrictive retention class to the held event(s) are deferred until hold release.

7.3.1 Hold application

Legal hold is applied via:

- **Customer portal** at a designated authorized-user role (typically "compliance officer" or "GC" customer-role). The application form captures: triggering matter (e.g., "X v. Y, S.D.N.Y. case no. 26-cv-12345"), applying principal, application timestamp, expected hold-release condition.
- **API**: programmatic hold application for systematic legal-hold scenarios (e.g., subpoena response automation). The API requires service-role-equivalent authorization + records the same audit- trail metadata.

The application is recorded in `audit_log_events` with the triggering-matter description + the applying-principal identity. The audit-log entry is itself chained + signed + anchored — hold-application is tamper-evident.

7.3.2 Hold precedence

Legal hold takes precedence over all other retention semantics:

- A `litigation_hold` event cannot be tombstoned even if a customer reclassifies the underlying retention class. Reclassification requires hold release first.
- A `litigation_hold` event cannot be deleted via GDPR right-to-erasure request. Counsel-led process required to reconcile: typically, the hold takes precedence (preserving for pending matter) until the matter is resolved, at which point the deletion request is honored.
- A `litigation_hold` event is exempted from the retention-sweep cron's deletion logic via explicit predicate at the cron's WHERE clause; verified at scripts/test-rls.ts Phase 7.D coverage.

7.3.3 Hold release

Hold release is the reverse procedure:

- **Customer portal**: authorized customer-role user releases the hold + captures the release reason (e.g., "matter resolved"; "preservation order vacated").
- **Audit-log entry**: the release is recorded with releasing- principal identity + release reason.
- **Subsequent retention semantics**: post-release, the event reverts to its pre-hold retention class. If the pre-hold class's retention period has expired during the hold, the next retention-sweep cron cycle processes the event per its pre-hold semantics.

7.3.4 Who can apply / release

The customer controls authorization for hold application + release. NuWyre does NOT unilaterally apply or release holds:

- **NuWyre operator** cannot apply a hold without explicit customer authorization (typically via documented operator manual procedure).

- **NuWyre operator** cannot release a customer-applied hold without explicit customer authorization.
- **Subpoena response**: if NuWyre receives a subpoena requiring preservation of customer events, NuWyre notifies the customer per the customer agreement's notification clause + the customer applies the legal hold via the customer-controlled procedure. NuWyre's own preservation obligation is satisfied by the customer-applied hold (which preserves the events) + by NuWyre's documented procedure of not silently deleting events in NuWyre-internal procedures.

The strict separation of customer-authorization-for-hold-actions prevents a NuWyre insider from silently releasing a hold to facilitate spoliation. The audit-log discipline + the customer-side controls + the operator manual's documented procedures form the substrate.

7.4 Tombstone vs deletion

Two distinct end-of-retention semantics:

7.4.1 Tombstone

Per §2.9: replace the event's `content` field with `[REDACTED:<reason>]` marker; preserve the original `content_hash` + chain integrity fields. The event row remains in the database; the chain remains verifiable; the original content is no longer retrievable.

Use when: retention period has expired for a class requiring data-minimization (`pii_minimized`, `phi_minimized`); customer-side preservation copies exist for cases where original content must be re-validatable against the preserved chain.

7.4.2 Hard deletion

The event row is removed from the database after a documented grace period; the chain's `prev_event_hash` linkage from subsequent events to the deleted event is preserved (the chain doesn't break) but the deleted event's content + chain-position-metadata is no longer in the database.

Use when: retention period expired for `default` class with no customer-side preservation requirement; customer has explicitly configured hard-deletion + the retention-sweep cron's grace period has elapsed.

Operational mechanics:

1. Retention-sweep cron identifies expired events.
2. Events are archived to `operator_archive_log` with archive metadata (event_id + archived_from_table + retention_class + archive_timestamp + archive_metadata) per migration `20260526000000`. The archive is append-only (operator_archive_log trigger rejects UPDATE + DELETE; service-role-bypass exists only for the Phase-7+ cold-storage-transfer path).
3. After grace period (configurable; V1 default 7 days), the original event row is hard-deleted via service-role bypass of the events table's append-only trigger (per migration `20260527000000` `auth.uid() IS NULL` bypass clause).
4. The deletion is recorded in `audit_log_events` with deletion metadata + the operator_archive_log archive_log_id reference.

7.4.3 Cryptographic implications

Tombstone: chain remains fully verifiable. A relying party verifying a bundle containing tombstoned events sees the tombstone marker + verifies the preserved content_hash matches the chain. If the relying party has a pre-tombstone copy of the content, they can independently validate the content matches the preserved hash.

Hard deletion: chain remains verifiable for non-deleted events (prev_event_hash linkage is preserved). For deleted events, the relying party can attest "an event with this event_hash existed at this chain position at deletion time" via the `operator_archive_log` entry + the daily-root anchor for the deletion date. The deleted event's content is no longer re-

trievable from NuWyre; customer-side preservation copies (if made before deletion) remain the only source of original content.

7.5 Retention-sweep enforcement

The retention-sweep cron runs daily + identifies events for retention-class-driven action:

```
For each event where archived_at IS NULL AND legal_hold = false:
  If retention_class = 'litigation_hold': skip (held)
  Else if event's age >= retention_period:
    If retention_class IN ('pii_minimized', 'phi_minimized'):
      tombstone (preserve hash; replace content)
    Else if retention_class = 'default':
      archive to operator_archive_log (records preservation)
      mark archived_at = now (waits for grace period)
  Else: skip (not yet expired)

For each archived event where archived_at + grace_period < now:
  hard-delete via service-role bypass
  record deletion in audit_log_events
```

The cron is implemented at `apps/api/src/app/api/cron/retention-sweep/ route.ts`. Each cron run:

- Is logged with start + end timestamps + operations performed.
- Includes a dry-run mode (`--dry-run` flag) for operator pre-execution validation.
- Records all archive + tombstone + deletion operations in `audit_log_events` for the affected events' organization.
- Surfaces errors to operator-side observability for investigation.

7.5.1 Audit trail per retention operation

Every retention-sweep action produces an audit-log entry:

```
{
  "event_id": "<UUID>",
  "event_type": "audit-log:retention:<operation>",
  "actor": { "type": "cron", "id": "retention-sweep" },
  "subject": {
    "type": "<events | audit-log-events>",
    "id": "<affected event ID>",
    "organization_id": "<org UUID>"
  },
  "content_hash": "<sha256 of canonical operation metadata>",
  "forensic": { /* chain integrity fields */ }
}
```

The audit-log chain is independently verifiable + anchored alongside the primary event chain (per dual-subtree composition §2.5.3). A relying party can reconstruct the complete retention-operation history for any event from the audit-log chain.

7.5.2 Dry-run mode

The retention-sweep cron's dry-run mode produces the same identification logic + the same audit-log preview but does NOT execute the operations. Used for:

- Operator pre-execution validation that the sweep will not affect legal-held or otherwise-protected events.
- Customer pre-execution review for customers wanting visibility into upcoming retention actions.

- Compliance audit of "what would happen if we ran the sweep tomorrow" without committing the actions.

7.6 Customer-managed encryption keys interaction

For customer-managed-encryption-key (CMK) deployments (custom-quote engagement per operator manual §4):

- **Tombstone**: works identically; the tombstone marker replaces the encrypted content blob, preserving the customer-key-encrypted-content_hash.
- **Hard deletion**: requires customer-side coordination if the customer's KMS retention policy differs from NuWyre's deletion schedule. Typically: NuWyre's retention-sweep deletes the encrypted blob; customer's KMS retention policy determines when the encryption key itself is destroyed; once both are destroyed, the content is cryptographically unrecoverable.

The CMK retention interaction is documented at the customer's specific custom-engagement contract. NuWyre's role is to provide the encryption-substrate hooks; the customer's role is to manage their key lifecycle in alignment with their retention obligations.

7.7 Operator-side retention actions

Beyond automated retention-sweep, operator-side actions:

- **Emergency deletion**: per operator manual §7, the documented procedure for emergency cleanup of append-only tables. Used for: PII-leak post-ingestion (a customer-side mistake that ingested PII not expected to be retained); operator-induced corruption requiring cleanup. Every emergency action is recorded in the operations log at `docs/operations-log.md` + audit-logged.
- **Customer-requested deletion**: GDPR right-to-erasure requests
  - analogous deletion rights under other frameworks. Customer submits the request; NuWyre's procedure includes legal-hold reconciliation per §7.3 + the deletion operation + audit-log attribution.
- **Backup restoration**: in the event of PostgreSQL corruption requiring backup restoration, the restoration process re-validates chain integrity against the pre-corruption anchored roots. A successful restoration produces a chain that verifies against Bitcoin OTS receipts unchanged from pre-corruption state.

All operator-side actions follow the audit-log discipline: who acted, when, what was affected, what was the justification. The operations log (`docs/operations-log.md`) is the canonical record of operator-side actions per the system-description document at `/legal/system-description.pdf`.

7.8 What retention does NOT do

The honest scope:

- **Does not enforce regulatory compliance automatically**. The retention classes are a configurable substrate; the customer's counsel decides which class maps to which regulatory framework. NuWyre's retention engine executes the classes as configured.
- **Does not guarantee enforceability against subpoena**. Subpoena response is the customer's counsel's process; NuWyre provides the preservation hooks + the chain-of-custody audit log.
- **Does not survive complete NuWyre infrastructure loss**. Per §9.9, already-anchored bundles remain verifiable using only the open-source verifier + Bitcoin + GitHub. But events still in NuWyre's database that have NOT been exported as bundles are lost if NuWyre's infrastructure is destroyed. Customers should maintain periodic bundle exports (daily; weekly; per their retention-strategy decision) to ensure independent preservation.

- **Does not provide legal advice**. The retention class chosen for a particular customer's event stream is a legal question; NuWyre's retention engine executes the configured choice. Customers must consult counsel for class-selection decisions.

7.9 Where to find the implementation

- **Retention substrate migration**: `packages/database/supabase/migrations/20260526000000_phase_6_2_b_e_retention_sweep_substrate.sql` (operator_archive_log table; RLS policies; append-only triggers).
- **Retention-sweep trigger-bypass migration**: `packages/database/supabase/migrations/20260527000000_phase_6_2_c_d_retention_sweep_trigger_bypass.sql` (`auth.uid() IS NULL` service-role bypass for the retention-sweep cron's UPDATE archived_at + DELETE).
- **Retention-sweep cron**: `apps/api/src/app/api/cron/retention-sweep/route.ts`.
- **RLS test coverage**: `packages/database/scripts/test-rls.ts` Phase 7.D section validates operator_archive_log RLS + retention-bypass semantics.
- **Operations log**: `docs/operations-log.md` records all operator-side retention actions.

A relying party auditing NuWyre's retention behavior can read the above files + reproduce the retention semantics from first principles. The implementation is the authoritative source; this methodology section is the prose-summary.

8. Audio Handling

This section documents how NuWyre handles audio attached to AI agent events — typically voice-platform-integration scenarios where the agent is conducting a real-time voice conversation (Twilio Voice; Bland; Retell; OpenAI Realtime). Audio is forensically distinct from transcription: for TCPA disputes, broker-dealer call-recording obligations, and many regulatory regimes, the AUDIO is the load-bearing evidence; transcripts alone are insufficient.

NuWyre's audio binding mechanism cryptographically ties audio files to the event chain, ensuring tampering with the audio (or substituting different audio) is detectable via the standard verification pipeline (per §4).

8.1 Audio binding mechanism

When a NuWyre event includes audio (typically system_event-type records for voice-platform-generated session events), the audio binding works as follows:

1. **Raw audio bytes**: the upstream voice platform provides the audio file (e.g., Twilio recording webhook delivers a recording URL; OpenAI Realtime audio stream is captured and serialized).
2. **SHA-256 of audio bytes**: NuWyre computes the SHA-256 of the raw audio file bytes.
3. **audio_ref embedded in event content**: the event's `content.audio_ref` field captures:

```
{
  "audio_ref": {
    "hash": "<sha256-hex of raw audio bytes>",
    "format": "<wav | mp3 | opus | ogg | etc>",
    "duration_seconds": <number; from upstream platform metadata>,
    "source": "<twilio-voice | bland | retell | openai-realtime | direct>",
    "source_recording_id": "<upstream platform's recording identifier>"
  }
}
```

4. **content_hash incorporates audio_ref**: the event's `content_hash` (per §2.1.1) is computed over the canonical form of `content`, INCLUDING `audio_ref`. The audio_ref hash is part of the chain.
5. **Chain integrity propagates**: the event's `event_hash` (per §2.1.2) incorporates the content_hash; subsequent events' `prev_event_hash` references this event_hash. Modifying the audio file changes its SHA-256, which changes content_hash, which changes event_hash, which breaks the chain at this point.

The audio file itself is stored separately from the event (see §8.2). The cryptographic binding is via the hash; the storage location can change without affecting the chain integrity.

8.1.1 Why hash-binding rather than embedded audio

The audio file is typically substantial (~1 MB per minute for compressed audio; 10 MB+ per minute for uncompressed). Embedding audio bytes directly in the event would:

- Bloat the event chain (an hour-long call would produce a 600 MB event record).
- Bloat the bundle export (a multi-day customer-export bundle could be tens of GB).
- Bloat the daily Merkle tree (Merkle leaves become enormous).

The hash-binding approach preserves cryptographic integrity (the hash IS in the chain) while keeping the events themselves compact. Audio storage + retrieval becomes a separate concern.

8.1.2 What the binding proves

A relying party verifying a bundle containing audio-bound events can attest:

- "The audio file at `audio/<sha256>.<ext>` has the SHA-256 recorded in the event's audio_ref.hash" (verified by re-hashing the audio file in the bundle's `audio/` directory + comparing against the event's audio_ref.hash).
- "The event's content_hash incorporates this audio_ref" (verified by re-canonicalizing the event's content + re-computing content_hash + comparing against the stored content_hash).
- "The event is in a valid chain extending back to genesis" (verified per Check 3).
- "The chain is anchored to the daily Merkle root in Bitcoin / RFC 3161 / GitHub" (verified per Checks 5/6/7).

Therefore: "this audio file is the audio file the customer's AI agent generated/received at the ingestion-recorded time, and has not been substituted or tampered with since."

8.2 Storage layout

Audio files are stored in a private Supabase Storage bucket (`evidence-audio`) under content-addressed paths:

```
evidence-audio/
└── <organization_id>/
    └── <sha256>.<extension>
```

The path components:

- **`<organization_id>`**: per-organization scoping; RLS prevents cross-organization access at the storage policy layer.
- **`<sha256>`**: the audio file's own SHA-256 hex (the same value recorded in `audio_ref.hash`). Content-addressed: the same audio bytes always produce the same path.
- **`<extension>`**: indicates the audio format. NOT part of the hash input; included for tooling convenience.

8.2.1 RLS policy

The `evidence-audio` bucket's RLS policy admits only:

- The owning organization's authenticated members (read).
- The NuWyre service-role context (read + write).

Cross-organization access attempts (a customer's authenticated user attempting to fetch another organization's audio file via guessed URL) are rejected at the storage policy layer. Verified at `packages/database/scripts/test-rls.ts` Phase 6 audio-storage coverage.

8.2.2 Bucket-level encryption at rest

Supabase Storage applies server-side encryption to the `evidence-audio` bucket at rest. The encryption is operator-managed (Supabase's managed encryption keys); customers can request CMK (customer-managed-key) encryption for the bucket under custom-quote engagement per §7.6 + operator manual §4.

V1 default encryption is NOT zero-knowledge — NuWyre operator infrastructure has the technical capability to decrypt audio files for legitimate operational purposes (retention sweep, customer- requested bundle export, subpoena response with customer-side notification). The zero-knowledge audio storage option is a separate engagement requiring different infrastructure (HSM-backed key management; customer-side decryption tooling).

8.3 Audio in bundle exports

When a customer exports a bundle containing audio-bound events, the bundle's `audio/` directory contains the audio files:

```
bundle.zip/
└── audio/
    └── <sha256>.<extension>
```

The bundle's `manifest.artifacts` includes per-audio-file entries with their SHA-256s + sizes. Check 2 (artifact integrity) verifies each audio file's bytes match the manifest-recorded SHA-256.

8.3.1 Audio-bundle size implications

Audio-bundle exports are substantial. A typical voice-platform customer with active call traffic generates ~MB/minute of audio; a daily bundle could be hundreds of MB; a multi-day bundle could be GB.

Operational implications:

- **Bundle generation**: takes longer (audio download from storage + re-packaging in the bundle ZIP).
- **Bundle distribution**: customers receiving bundles via download need bandwidth-adequate retrieval; very-large bundles may benefit from differential downloads (V1.1+).
- **Storage cost**: customers paying for audio retention should understand the storage-cost implications of long retention periods + extensive audio-bound event volumes.

The custom-quote engagement (operator manual §4) addresses audio-retention cost + duration on a per-customer basis.

8.4 Retention independence

Audio retention is independent from the event retention class. The `audio_records` table carries its own `retention_class` field (same closed vocabulary as event retention per §7.1) which can differ from the parent event's retention class.

8.4.1 When audio retention diverges from event retention

Common scenarios where they diverge:

- **State wiretapping consent laws**: a recording made in a two-party-consent state may have shorter retention obligations than the underlying business event. State laws may require deletion of audio after a specific period even if the business record (the EVENT) has longer regulatory retention.
- **HIPAA PHI provisions**: audio containing PHI may be subject to 6-year HIPAA retention; if the underlying event is a non-PHI business transaction, the event may have 7-year retention.
- **Biometric privacy laws (BIPA, similar)**: voice biometrics may be subject to data-minimization obligations; audio retention can be shorter than event retention to comply with biometric data minimization.
- **Jurisdiction-specific recording statutes**: certain jurisdictions prohibit retention of call recordings beyond specific periods; the event business record can persist while audio is age-out-deleted.

8.4.2 Sweep logic: shorter wins for deletion, longer wins for hold

The audio retention sweep follows the same principle as event retention (§7.2.1):

- **Deletion**: shortest applicable retention period triggers audio deletion (subject to legal-hold override).
- **Hold**: longest applicable hold period delays audio deletion.

A held audio file remains retrievable as long as ANY applicable hold is in effect; an unheld audio file is deleted when the shortest applicable retention period expires.

8.5 Audio redaction without chain break

When audio must be removed under a retention or compliance trigger (retention expiry; customer-requested deletion; subpoena requiring audio production with subsequent destruction):

1. The audio file in `evidence-audio` storage is deleted.
2. The `audio_records` row is updated to a tombstone state:
   - Original `audio_ref.hash` is preserved.
   - Original `audio_ref.format` + `duration_seconds` + `source` + `source_recording_id` are preserved.
   - A new field `tombstoned_at` is set + a `tombstone_reason` captures the deletion justification.
3. The bundle export's behavior:
   - Tombstoned audio_ref still appears in the event content (preserved chain).
   - The bundle's `audio/` directory does NOT contain the tombstoned audio file.
   - The `manifest.artifacts` does NOT include the tombstoned audio file (verifier won't expect to find it).
   - A `tombstone_notice.json` in the bundle enumerates which audio files were tombstoned + their preserved hashes + tombstone reasons.
4. The event's chain integrity remains intact — content_hash + event_hash + chain links unchanged.

8.5.1 What tombstoned audio attests to

A relying party encountering tombstoned audio in a bundle can attest:

- "An audio file with this SHA-256 existed at this event's ingestion time" (the audio_ref.hash is preserved in the chain).
- "The chain remains internally consistent + anchored" (the chain verification proceeds normally).
- "The audio bytes are no longer retrievable from NuWyre" (the tombstone explicitly indicates this).

If the customer (or a regulatory adversary) has a pre-tombstone copy of the audio file with the matching SHA-256, they can independently validate "this audio file is the audio file that was in NuWyre's chain at the recorded time." The preserved hash is the substrate for independent verification.

8.5.2 Customer-side audio preservation

For customers needing long-term audio access independent of NuWyre's retention class — common for legal-defense scenarios requiring permanent audio retention — the recommendation:

1. Configure audio retention to match the longest applicable retention obligation (extended retention class).
2. AT BUNDLE EXPORT TIME, preserve a copy of the bundle (including audio files) in the customer's controlled long-term storage.
3. Recompute audio file SHA-256s at customer-side preservation time + verify they match the bundle's manifest-recorded values.
4. Preserve the customer-side bundle indefinitely (or per the customer's specific retention strategy).

NuWyre's retention engine handles the operational mechanics; the customer's long-term-evidence strategy is the customer's responsibility (per §9.6).

8.6 Voice consent capture: the TCPA case

A specific high-stakes audio context: TCPA (Telephone Consumer Protection Act) compliance for customer-facing AI agent voice calls.

TCPA requires:

- Express written consent (for SMS) or prior express consent (for certain call types) before contacting consumers via automated systems.
- The consent must be documented + retrievable in the event of a dispute.
- Violations carry $500-$1,500 per call statutory damages.

For voice-platform AI agents conducting outbound calls, the **captured audio of the consent capture** is often the load-bearing evidence in TCPA disputes. Transcription alone is insufficient because:

- Transcription accuracy is bounded by the upstream platform.
- Transcripts can be challenged for ambiguity ("did the consumer actually say 'yes'?").
- The audio captures intonation, hesitation, and other contextual signals that may be material to the consent's legal validity.

NuWyre's audio binding produces:

- A cryptographically-anchored audio file with sub-second-resolution timestamp.
- A preserved SHA-256 hash linking the audio to the consent-capture event.
- A chain position anchored to Bitcoin proving the audio existed at the recorded time.

In a TCPA dispute, the customer's defense can produce: "Here is the audio file from the consent-capture event. Its SHA-256 matches the audio_ref.hash in the event. The event is anchored at chain position N in the Bitcoin block at height B at time T. T is BEFORE the disputed-call time."

The bound audio is 95% of the defense; the transcript alone is ~30% of the defense. NuWyre's role is to provide the cryptographically-preserved audio + the chain-of-custody proof; the customer's counsel uses the bundle in the legal proceeding.

8.7 Voice platform-specific audio handling

8.7.1 Twilio Voice

Twilio sends recording-available webhook events when a call recording completes. NuWyre's Twilio Voice adapter:

1. Receives the webhook (per §6 detection pipeline).
2. Fetches the audio from Twilio's recording URL (with appropriate auth + retry handling).
3. Computes SHA-256 of the fetched bytes.
4. Stores the audio in `evidence-audio/<org_id>/<sha256>.wav`.
5. Creates an event with `audio_ref.source: "twilio-voice"` + `audio_ref.source_recording_id: "<Twilio RecordingSid>"` for traceability back to the source platform.

If Twilio's recording is unavailable (Twilio-side outage; deleted recording; permission revocation), NuWyre's adapter records an event without audio_ref (audio_unavailable_reason captured in content). The chain remains intact; the audio binding is simply absent for that event.

8.7.2 OpenAI Realtime

OpenAI Realtime API provides real-time audio streams. V1 NuWyre adapter captures session-level metadata; per-utterance audio binding is V1.1+ scope (requires substantial audio-stream- processing infrastructure).

The V1 contract: OpenAI Realtime session events are recorded with content describing the session (model version, voice configuration, session duration); no per-utterance audio binding yet. Customers needing per-utterance audio for OpenAI Realtime should configure the upstream OpenAI session to capture session audio + provide NuWyre with the captured audio via the Twilio-Voice-style webhook pattern (operator manual §3 documents the integration pattern).

8.7.3 Bland and Retell

Bland and Retell provide recording URLs via their respective webhook patterns. NuWyre's adapters for both platforms:

- Receive the webhook.
- Fetch the audio per platform-specific auth + retry semantics.
- Compute SHA-256 + store + record event with audio_ref.

The adapter implementations are at `packages/integrations/src/adapters/bland-client.ts` + `packages/integrations/src/adapters/retell-client.ts`.

8.8 What audio handling does NOT do

The honest scope:

- **Does not validate transcription accuracy**. NuWyre's audio binding attests to the audio file as ingested. The transcription (typically produced by the upstream voice platform) is recorded in the event's content but NuWyre does not independently verify transcription accuracy. Customers needing transcription accuracy validation should pair NuWyre with a separate transcription QA process.
- **Does not provide real-time audio analytics**. Audio is captured
  - bound + retained; real-time analytics on audio content (sentiment detection, voice biometrics, real-time PII detection in audio) are V1.1+ scope. V1 detection layer operates on transcribed content (per §6); audio is the preserved evidence, not the real-time analysis substrate.
- **Does not interact with the upstream voice platform's storage**. NuWyre fetches audio from the platform's recording URL once + stores it in NuWyre-controlled storage. Subsequent changes to the upstream platform's recording (deletion; permission changes) do not affect the NuWyre-stored copy. Customer-side coordination with the upstream platform's retention policies is the customer's responsibility.
- **Does not provide audio in formats different from the upstream source**. NuWyre stores the audio file as received from the upstream platform (typically WAV for Twilio Voice; format varies by platform). Customers needing format conversion (e.g., MP3 for archive efficiency) should perform conversion at the customer-side preservation step.

8.9 Cryptographic implications of audio handling

A relying party verifying an audio-bound event:

1. **Audio file exists in bundle**: bundle's `audio/` directory contains the file at `<sha256>.<ext>` path.
2. **Audio file SHA-256 matches event audio_ref**: re-hash the audio file + compare against `event.content.audio_ref.hash`.
3. **Event content_hash incorporates audio_ref**: re-canonicalize event content (including audio_ref) + recompute content_hash + compare against `event.content_hash`.
4. **Event chain integrity holds**: per Check 3.

5. **Event is anchored**: per Checks 5/6/7.

A successful walk through all five steps establishes: "This audio file is THE audio file that was in NuWyre's chain at the recorded ingestion time. It has not been substituted or modified since. The chain position is anchored to Bitcoin / RFC 3161 / GitHub."

Tampering scenarios + their detection:

- **Substituted audio**: different audio bytes at the same path → file's SHA-256 differs from the manifest-recorded value → Check 2 (artifact integrity) fails.
- **Modified audio**: byte-level tampering of the audio file → same detection as substitution; Check 2 fails.
- **Removed audio**: audio file deleted from bundle → Check 2 fails (missing artifact).
- **Modified audio_ref.hash in event content**: tampered event has different content_hash → Check 3 (hash chain) fails.
- **Substituted audio file + matching audio_ref.hash update + matching content_hash recomputation + matching prev_event_hash recomputation + ... full chain rewrite**: requires also rewriting the daily Merkle root, the OTS receipt, the RFC 3161 receipts, the GitHub anchor commit. Defended by multi-leg anchoring per §1.

The chain-of-custody discipline + the multi-leg anchor + the customer-side bundle preservation form the cryptographic substrate for "this audio is the audio that was captured."

8.10 Where to find the implementation

- **Audio records substrate**: `packages/database/supabase/ migrations/<phase>_audio_records.sql` (audio_records table; RLS policies; per-org content-addressed paths).
- **Audio storage**: Supabase Storage bucket `evidence-audio` with RLS configuration in the database migration.
- **Audio-binding event schema**: `docs/spec/event-v1.schema.json` `content.audio_ref` field definition.
- **Adapter implementations**: `packages/integrations/src/adapters/twilio-client.ts` (Twilio Voice recording fetch + storage), `packages/integrations/src/adapters/bland-client.ts` (Bland recording fetch), `packages/integrations/src/adapters/retell-client.ts` (Retell recording fetch).
- **Bundle audio export**: `packages/evidence/src/generate-bundle.ts` includes audio files in the bundle ZIP per per-bundle audio_ref enumeration.
- **RLS test coverage**: `packages/database/scripts/test-rls.ts` audio-storage cross-org-rejection scenarios.

A relying party auditing NuWyre's audio handling can read the above files + reproduce the audio semantics from first principles. The implementation is the authoritative source; this methodology section is the prose-summary.

9. Limitations and Honest Disclosures

This section enumerates everything NuWyre does NOT guarantee. A hostile reviewer reads this section first to look for holes — that posture is invited. Honest enumeration of limitations is itself a trust-building artifact: a vendor who claims none of these limitations apply should be treated with skepticism.

The limitations below are organized by category. Each entry names the limitation, the reason it exists, and (where applicable) the mitigation NuWyre provides for the residual risk.

9.1 Cryptographic assumption dependencies

NuWyre's integrity claims rest on standard cryptographic assumptions that the academic community has validated to date but cannot guarantee for indefinite-future timelines.

9.1.1 SHA-256 collision resistance

NuWyre uses SHA-256 throughout: for content hashes, event hashes, Merkle tree internal nodes, anchor commitments. If SHA-256 collision resistance materially degrades during a bundle's retention period, an adversary with collision-finding capability could in principle produce a "tampered" event whose SHA-256 collides with the original — passing all hash-based verification despite modified bytes.

Current state: SHA-256 has no known practical collision attacks. The NIST and IETF treat it as cryptographically sound for 2026-era use. Quantum-cryptanalysis estimates suggest a post-quantum collision attack would still require infeasible resources for preimage attacks (Grover's algorithm halves effective security strength; SHA-256's post-quantum security is ~128 bits).

NuWyre's commitment: when cryptographic-community consensus shifts to a stronger primary hash, an `event-v2` schema will adopt it. Already-anchored events under v1 SHA-256 remain valid against the v1 contract; the migration path is "new keys + new schema for new events" not "rewrite history."

9.1.2 Ed25519 signature unforgeability

NuWyre uses Ed25519 for ingestion signatures + bundle manifest signatures. If the underlying twisted Edwards curve assumption breaks (highly improbable for Curve25519 specifically), an adversary could forge NuWyre signatures.

Current state: Ed25519 has no known practical attacks. Curve25519 was designed by Daniel J. Bernstein specifically to avoid the parameter-choice attacks that have undermined other elliptic curves.

NuWyre's commitment: post-quantum signature schemes (e.g., NIST-standardized lattice-based) will be evaluated for `event-v2` adoption when library + tooling maturity supports cross-language implementation parity.

9.1.3 Bitcoin proof-of-work cost

NuWyre's OTS anchor leg depends on Bitcoin's economic security: the cost of producing a deep reorganization of the Bitcoin chain at the relevant block-heights must remain prohibitive for the relying party's expected adversary.

Current state: Bitcoin's hash-rate represents tens of billions of dollars of capital expenditure + ongoing energy expenditure. A deep reorganization attack at the block-heights NuWyre's anchor commits sit at is widely considered economically irrational for any adversary short of a nation-state.

NuWyre's commitment: multi-leg anchoring (RFC 3161 + GitHub + Bitcoin) provides defense-in-depth. Compromise of Bitcoin's economic security alone does not collapse the integrity claim; the RFC 3161 TSAs + GitHub mirror provide independent attestation.

9.2 Third-party anchor dependencies

NuWyre's anchor pipeline depends on the continued operation + honesty of several third-party systems. Compromise of any single one does not collapse the chain — that is the design intent of multi-leg anchoring — but the multi-dependency surface should be acknowledged.

9.2.1 OpenTimestamps calendar operators

NuWyre submits daily roots to multiple OTS calendar servers. The calendars relay the submission to Bitcoin's mempool + return attestation receipts. Compromise of a single calendar would prevent that calendar from producing valid receipts; multi-calendar submission preserves the anchor.

Mitigation: multi-calendar OTS submission. Verifier accepts any successful Bitcoin attestation from any calendar.

Residual risk: simultaneous compromise of all OTS calendar operators — a fantasy threat model per §1.1. Multi-decade-operated calendar infrastructure operated by independent organizations.

9.2.2 RFC 3161 commercial TSAs

NuWyre uses three TSAs at v1: FreeTSA (non-profit), Sectigo (commercial), DigiCert (commercial). The ≥2-of-3 verification threshold means single-TSA compromise does not collapse the chain.

Mitigation: 2-of-3 threshold; cert chains pinned to trust roots embedded in the verifier; ongoing operator monitoring of TSA cert expiration + key-rotation events.

Residual risk: simultaneous compromise of ≥2-of-3 TSAs — would require independent compromise of three commercial timestamping authorities operated by different organizations under different jurisdictions. Detection: verifier surfaces partial-verification for the affected day; operator-led customer notification per operator manual §6.

9.2.3 GitHub anchor repository

NuWyre's anchor commits are mirrored to a public GitHub repository (`NuWyre/anchors`) + Codeberg mirror. GitHub compromise could in principle forge or delete commits; Codeberg mirror provides cross- anchor verification.

Mitigation: dual-mirror (GitHub + Codeberg); commits are signed with NuWyre's GitHub PGP key (separate from the ingestion signing key); Bitcoin OTS receipts provide independent attestation that does not depend on GitHub at all.

Residual risk: GitHub-account compromise that includes NuWyre's PGP signing key + Codeberg-account compromise + GitHub Actions override of the anchor-mirror workflow. Bitcoin OTS attestation remains valid in this scenario; the GitHub leg degrades to "not verifiable" rather than to "forged-and-passes-verification."

9.3 LLM evaluator judgment quality

Per §5, NuWyre's policy evaluation uses LLM evaluators with deterministic settings (temperature=0; model version pinned). The evaluator produces verdict + severity + reasoning per policy-pack rule.

Limitation: the evaluator's judgment is bounded by:

- The underlying model's capabilities + training distribution.
- The policy pack prompt's framing + edge-case coverage.
- The cross-validation pair coverage (fire / no-fire fixtures per §5.6 + §5.7).

Empirical measurement: per §5.7, each rule has a measured precision + recall on its fixture set. Production deployments report these metrics as part of bundle-level evaluation provenance.

Honest scope: the LLM evaluator is NOT a substitute for human review on high-stakes findings. A regulator-significant finding (a TCPA pre-call-consent violation; a HIPAA-aligned phi-leak; an off- script high-severity deviation) should

trigger operator-facing notification per §6 with human review of the underlying event + the evaluator's reasoning. The evaluation is the trigger for investigation; the human reviewer is the deciding authority.

Bias considerations: LLM evaluators inherit biases from training data. NuWyre cannot independently audit upstream model bias; the mitigation is per-rule precision/recall measurement on the customer's specific use-case fixtures + ongoing monitoring of evaluator-vs-human-reviewer agreement rates. Customers deploying high-stakes evaluation in protected-class contexts must perform bias auditing on the specific rule set deployed.

9.4 What evidence bundles do NOT prove

A successfully-verified bundle proves what NuWyre observed at ingestion + that it has not been tampered with since. It does NOT prove:

9.4.1 Upstream AI agent correctness

The bundle attests to ingested content. It does NOT attest to the AI agent's underlying decision-making correctness, the prompt configuration that produced an output, or the absence of model-side hallucination in any specific exchange. A customer using NuWyre bundles in a regulatory defense should pair the bundle with the agent's deployment-time configuration audit trail (prompt versions, model versions, system prompts, tool definitions).

9.4.2 Customer-side consent validity

The bundle records that consent capture occurred (if the event stream includes a consent-capture event) and what was said. It does NOT validate that consent was legally valid in the relevant jurisdiction. Consent validity is a function of the customer's consent-capture procedure (TCPA-compliant double-opt-in; HIPAA authorization form; GDPR explicit-consent mechanism) + the customer's documentation of that procedure.

9.4.3 Regulatory verdict

The bundle is evidence; it is not verdict. A regulator may rule favorably or unfavorably based on the bundle + the broader case record + the regulator's interpretation of the relevant statute. NuWyre cannot guarantee favorable rulings. NuWyre provides the evidence layer; the legal interpretation is the customer's counsel's responsibility.

9.4.4 Voice transcription accuracy

For voice-platform integrations (Twilio, Bland, Retell), the transcription that NuWyre ingests is produced by the upstream voice platform. NuWyre attests to "this is what the voice platform sent us." It does NOT attest to "this is what was actually said in the call." Transcription accuracy is the upstream platform's responsibility; NuWyre provides the audio_ref binding (per §8) for cases where the original audio is available + the bundle's forensic integrity for cases where it is not.

9.5 Adapter trust boundary

Ingestion adapters (the customer-side or platform-side code that sends events to NuWyre's `/v1/events` endpoint) are trusted ONLY for event content. The chain integrity fields (sequence_number, prev_event_hash, content_hash, event_hash, ingestion_signature) are computed by NuWyre's ingestion service — NOT by the adapter.

Why this matters: an adapter cannot inject fake hash chains because the adapter does not compute the hash chain. A compromised adapter can submit fabricated content that NuWyre will sign + chain (see Class B threat in §1.3), but the resulting chain remains internally consistent + tamper-evident. The customer-side investigation of "did our adapter get compromised?" is separate from the chain-integrity investigation; NuWyre's bundle provides the evidence trail for what was ingested, not the diagnosis of whether the ingestion was authentic.

9.6 Operational dependencies the customer must maintain

NuWyre's verification is mechanical + reproducible, but it depends on the customer maintaining several operational practices:

9.6.1 Bundle storage hygiene

Bundles are immutable cryptographic artifacts; customers should store them with appropriate integrity controls (write-once storage where available; SHA-256 hash recorded at receipt time; chain-of- custody documentation for who handled the bundle when).

NuWyre cannot enforce customer-side storage hygiene. Out-of-band hash-recording at bundle-receipt time is the customer-side discipline that defends against bundle-substitution attacks (T7 in §1.4).

9.6.2 Public key verification

The NuWyre Ed25519 issuer public key is pinned in the verifier binary. Customers should:

- Verify the verifier binary's SHA-256 against the published release-notes hash.
- Independently verify the pinned key fingerprint matches what is published in this methodology PDF + at `nuwyre.com/verify`.
- Be alert to verifier-substitution attacks where a malicious verifier ships with attacker-controlled pinned keys.

A relying party who cannot independently verify the verifier binary's authenticity should re-build from source (see §4.8).

9.6.3 Time-of-receipt documentation

Customers receiving bundles should document the time of receipt + the bundle hash + the source URL. This metadata is the customer- side substrate that distinguishes a bundle the customer received from NuWyre from a bundle an adversary substituted later. NuWyre's bundle export endpoint records the export time + bundle hash; the customer's responsibility is to record receipt-time metadata independently.

9.7 Standards-track maturity disclosures

Spec v1.x is the first publicly-ratified version of bundle-format-v1. The conformance fixture suite (14 fixtures + reference verifier) provides implementation-validated invariants. Standards-community review will surface edge cases in production.

Limitation: until additional independent verifier implementations exist + run against production bundles, the spec is single-implementation-validated. The standards-track posture (§5 "multiple independent implementations") is the goal; the achievement is gradient.

NuWyre's commitment: the spec + fixtures + KAT golden vectors are versioned + published. Spec evolution follows SPEC_GOVERNANCE.md procedure (semantic versioning; documented amendment process; backward-compatibility commitments).

Customer-side mitigation: customers deploying NuWyre as their sole evidence layer should maintain an alternative integrity surface (off-system backups; a second evidence vendor) during the v1.x maturity window. The "all-eggs-in-one-evidence-basket" posture is appropriate after the standards-track surface has matured through multiple independent implementations + standards-body review.

9.8 V1 vs V1.1+ scope boundaries

Several capabilities are V1.1+ scope, explicitly excluded from V1:

9.8.1 Real-time tamper detection

V1 anchors daily. A streaming-anchor mode (per-hour or per-event anchoring) is V1.1+ scope. Customers requiring sub-second tamper detection must not rely on V1 as the sole integrity layer.

9.8.2 Customer-managed encryption keys

V1 uses Supabase's at-rest encryption for stored events. Customer- managed-key encryption (with KMS-backed key control) lands in larger custom-quote engagements per operator manual §4. Customers with regulatory requirements for customer-controlled encryption keys should evaluate whether V1's Supabase-managed encryption meets their requirement OR contract for the custom engagement.

9.8.3 Audit-log-export production deployment

V1 ships the audit-log-export bundle spec + fixture suite + verifier support, but production audit-log-export crons remain in deploy- bootstrap state. Customers needing audit-log-export bundles in production should coordinate with NuWyre during the bootstrap window.

9.8.4 Standalone Go binary CLI distribution

V1 ships the verifier as source-buildable Go + as in-browser WASM. Pre-built binary distribution via GitHub releases is V1.1+ scope. Customers requiring pre-built binaries should build from source per §4.8 in the interim.

9.8.5 Quarterly methodology review log

This methodology document is versioned but lacks ongoing quarterly review entries (§10) until the first production review window completes. The §10 review log will accumulate entries post-first- customer + post-first-quarter-of-production-deployment.

9.9 What happens if NuWyre disappears

This is the question every compliance buyer asks first about a SaaS evidence vendor. The honest answer:

Already-issued bundles remain verifiable using only:

- The open-source verifier CLI (source-buildable from any clone of the NuWyre/cli repository).
- The public Bitcoin blockchain (any Bitcoin node can fetch the attested blocks).
- The pinned TSA trust roots embedded in the verifier (or fetched from the TSAs' published cert chains; the cert chains are industry-standard PKI artifacts independent of NuWyre).
- The public NuWyre/anchors GitHub repository (mirrored to Codeberg; both are independent of NuWyre's operational continuity).

No NuWyre infrastructure is in the verification path for already-issued bundles. A bundle issued in 2026 remains verifiable in 2046 even if NuWyre ceases to exist in 2027.

Continuity gates:

- Verifier source code is published under permissive licensing (per spec governance) so any party may continue maintaining it.
- Spec + fixtures + KAT vectors are published under permissive licensing so any party may produce additional verifier implementations.
- The anchor pipeline (OTS + RFC 3161 + GitHub) is built from industry-standard tools (`opentimestamps-client`, standard RFC 3161 client libraries, `git`) so any successor operator can continue producing new bundles using NuWyre's published methodology.

The standards-track posture (per spec governance §5) is the operational embodiment of this commitment: NuWyre's value is in the service + the institutional consistency, not in the code being secret. Customers concerned about long-term NuWyre continuity should:

1. Retain the verifier binary + source code at bundle-receipt time.
2. Retain a copy of the spec + fixture suite + KAT vectors at bundle-receipt time.
3. Verify periodically (annually; at retention-period milestones) that already-received bundles still verify under the then-current pinned verifier.

The standard outlives the company. That is the design intent of the standards-track posture.

9.10 Summary of what NuWyre guarantees vs does not

GUARANTEE	MECHANISM	LIMITATION
Events ingested are tamper-evident	Hash chain + Ed25519 + Merkle root + multi-leg anchor	Detection latency bounded by daily-root cadence; complete-infrastructure-takeover scenario excluded per §1.1
Bundles are verifiable without NuWyre	Open-source verifier + public Bitcoin + public GitHub mirror + pinned TSA trust roots	Verification depends on continued OTS + GitHub + TSA continuity; quantum-cryptanalysis timeline assumed favorable
Multi-leg anchoring	OTS Bitcoin + RFC 3161 ≥2-of-3 + GitHub commit	Anchor-pending state in V1 deploy-bootstrap window; ephemeral-session-attestation deferred to two-key topology
LLM evaluation determinism	temperature=0 + pinned model version + body_hash	Evaluator judgment bounded by model + prompt + fixture coverage; NOT a substitute for human review on high-stakes findings
Customer-managed encryption key option	Custom-quote engagement substrate	Not in V1 default deployment; requires explicit contracting
Real-time tamper detection	DEFERRED — V1.1+ streaming-anchor mode	V1 detection latency bounded by daily-root cadence (~24 hours)
Regulator-favorable verdict	NOT GUARANTEED	Evidence layer is necessary but not sufficient for compliance posture
Upstream AI agent correctness	NOT GUARANTEED	NuWyre attests to ingestion, not upstream behavior
Voice transcription accuracy	NOT GUARANTEED	Upstream voice platform's responsibility
Customer-side consent validity	NOT GUARANTEED	Customer's consent-capture procedure + documentation responsibility

This summary is what a compliance officer should be able to recite about NuWyre after reading the methodology. If anything in the left column is unclear, the relevant section of the methodology addresses it in detail. If anything in the right column is unfamiliar, customers should consult their counsel on whether NuWyre's scope fits their specific regulatory context.

10. Quarterly Review Log

> **Stub.** *First entry lands at Phase 5 deployment per build plan v3.1.5 "Notes on Using This Plan." The log is part of the methodology document, not a separate file; entries accumulate over the document's lifetime.*

Format

Each quarterly review produces a row with explicit fields:

```
Reviewer:    <name>
Affiliation: <org>
External:    <yes | no>
Scope:       <which sections reviewed; which case law surveyed>
Deferred:    <items raised but not yet addressed; with rationale>
Date:        <YYYY-MM-DD>
```

A row with `External: no` is acceptable as a bridge entry provided it is surrounded by entries with `External: yes`. A log composed entirely of `External: no` entries is not credible regardless of how the prose describes it. Honest field labels beat prose hedging.

Review cadence (operator's manual v1.1.3 §5)

- **Year 1 Q1-Q2** — pre-revenue. Law-school clinic engagements (Stanford CIS, Berkeley Tech Clinic, Harvard Cyberlaw Clinic). Free, slow, publicly verifiable.
- **Year 1 Q3-Q4** — early revenue. Flat-fee external spot-checks ($1-2K per quarter).
- **Year 2+** — post-$500K ARR. Full quarterly review on retainer ($2-5K per quarter, $8-20K per year).

Entries

> *No entries yet. First entry lands Phase 5 deployment.*

11. Worked Example

> **Forthcoming in methodology v1.2.** *Lands after Phase 3 produces real ingestion data. The worked example walks an actual flagged interaction end-to-end: event ingestion → hash chain → daily Merkle root → dual anchoring (OpenTimestamps + RFC 3161 + GitHub) → evaluation → flag → evidence bundle export → independent CLI verification. Real bytes, real cryptographic proofs, anonymized customer context.*

Why deferred

A worked example using synthesized data would be a methodological demonstration; a worked example using real ingestion data is a methodological proof. Build plan v3.1.5 explicitly defers this section to v1.2 to preserve the latter character.

What v1.0-rc readers should know in the meantime

The example evidence bundle published at `apps/marketing/public/examples/nuwyre_export_cypress-derm_2026-04-22.zip` is a working artifact today: 33 EventV1 records composed from 10 active validation scenarios, real OpenTimestamps anchor (in Bitcoin's mempool awaiting confirmation as of bundle generation), Ed25519-signed manifest, Puppeteer-rendered cover PDF. It demonstrates the bundle format and the cryptographic chain. v1.2's worked example walks the same artifact's content through the methodology in narrative form.