GTS-SPEC: Graph Transport Substrate Wire-Format Specification

Normative wire-format specification for append-only RDF 1.2 graph transport

Blackcat Informatics Inc. Patrick Audley

2026-06-18T00:00:00Z

Contents

1 GTS — Graph Transport Substrate — Specification **2**
1.1 Abstract . . . 3
1.2 Status of this document . . . 3
1.3 Document history . . . 3
1.4 Table of contents . . . 4
1.5 1. Overview and non-goals . . . 5
1.6 2. Terminology and conformance . . . 6
1.6.1 2.1 Conformance scopes . . . 6
1.6.2 2.2 Reader and writer conformance classes . . . 7
1.6.3 2.3 Baseline reader API shape . . . 7
1.6.4 2.4 Reader diagnostics . . . 7
1.7 3. File structure . . . 8
1.7.1 3.1 Multi-segment files (`cat`-append composition) . . . 8
1.7.2 3.2 Streaming and progressive enhancement . . . 9
1.7.3 3.3 Layout states: accretive and streamable . . . 10
1.8 4. CBOR conventions . . . 11
1.9 5. Header . . . 11
1.10 6. Frames . . . 12
1.10.1 6.1 Payload resolution . . . 12
1.10.2 6.2 Index frame (optional) . . . 13
1.11 7. Graph data model and fold . . . 14
1.11.1 7.1 Terms (`terms` frame) . . . 15
1.11.2 7.2 Term-id assignment (normative) . . . 16
1.11.3 7.3 Quoted triples and reifiers (`reifies` frame) . . . 16
1.11.4 7.4 Quads and annotations . . . 16
1.11.5 7.5 Fold algorithm (normative) . . . 17
1.11.6 7.6 Opaque nodes . . . 17
1.11.7 7.7 Streaming fold and bounded memory . . . 18
1.11.8 7.8 Duplicates and conflicts (normative) . . . 18
1.12 8. Transform catalog . . . 19
1.12.1 8.1 Classes . . . 19
1.12.2 8.2 Stacking . . . 19
1.12.3 8.3 Capability model and graceful degradation . . . 20
1.12.4 8.4 Mandatory core set and durability . . . 20
1.12.5 8.5 Canonical codec registry (v1) . . . 20
1.13 9. Integrity and confidentiality . . . 20
1.13.1 9.1 Per-frame self-hash and content-id chain (mandatory) . . . 20
1.13.2 9.2 Signatures (optional, algorithm-agile) . . . 21

1.13.3 9.3 Encryption (optional) . . . 21
1.13.4 9.4 The opacity invariant (normative) . . . 22
1.14 10. Compaction . . . 22
1.14.1 10.1 Streamable compaction (ordering-only) . . . 22
1.15 11. Suppression (additive “deletion”) . . . 23
1.16 12. Binary and content-addressing . . . 24
1.16.1 12.1 Nested GTS (recursive composition) . . . 24
1.17 13. Profiles . . . 24
1.17.1 13.1 Language-tag discipline (profile-level normative) . . . 26
1.17.2 13.2 The `files` profile (optional-standard) . . . 27
1.17.3 13.3 The `stream` vocabulary (optional-standard) . . . 28
1.17.4 13.4 Domain profile example: `music-package` (informative) . . . 29
1.18 14. Transforms out . . . 30
1.18.1 14.1 Composition tooling requirements (normative for conformant tools) . . . 31
1.18.2 14.2 Archive tooling (`files` profile) . . . 31
1.19 15. Worked examples . . . 32
1.19.1 15.1 Minimal distribution snapshot (`dist`) . . . 32
1.19.2 15.2 Evidence: image + signed accrual (`evidence`) . . . 33
1.19.3 15.3 Notary: partially-opaque frame (`opaque`) . . . 33
1.19.4 15.4 Graceful degradation (`image`, content negotiation) . . . 33
1.19.5 15.5 Matryoshka: a whole signed GTS sealed inside a frame (`bundle` / `opaque`) . . . 33
1.20 16. Media type and HTTP serving contract . . . 33
1.20.1 16.1 Media type and file extension (normative) . . . 34
1.20.2 16.2 File identification algorithm (normative) . . . 34
1.20.3 16.3 HTTP serving semantics (normative) . . . 34
1.20.4 16.4 Immutability-aware caching (normative) . . . 34
1.21 17. Versioning and durability guarantees . . . 35
1.22 18. Security considerations . . . 35
1.23 19. Conformance test vectors . . . 36
1.24 20. IANA considerations . . . 37
1.24.1 20.1 Media type registration: `application/vnd.blackcat.gts+cbor-seq` . . . 37
1.25 21. Complete CDDL appendix . . . 38
1.25.1 21.1 Sequence grammar . . . 38
1.25.2 21.2 Copyable CDDL . . . 38
1.26 22. Hash, signature, and extension-key preimages . . . 42
1.26.1 22.1 Preimage and subject table . . . 42
1.26.2 22.2 Unknown extension-key behavior . . . 43
1.27 23. References . . . 44
1.27.1 23.1 Normative references . . . 44
1.27.2 23.2 Informative references . . . 44

Check for updates

1 GTS — Graph Transport Substrate — Specification

Field	Value
Document version	0.9-draft
Wire-format major version	1
Date	2026-06-18
Editor	Patrick Audley, Blackcat Informatics® Inc.
This version	`bfbe9f34142b116c26e5577963f418e5cfc3d267`

Field	Value
DOI	https://doi.org/10.67342/6pta6imnmw/v1

1.1 Abstract

GTS (Graph Transport Substrate) is an ontology-independent binary container and transport format for RDF 1.2 datasets and content-addressed binary payloads. A GTS file is a CBOR Sequence of one or more append-only segments. Each segment consists of a deterministic CBOR header followed by deterministic CBOR frames linked by BLAKE3 content identifiers. The logical dataset is obtained by a deterministic fold over the segment sequence. GTS supports partial readability, opaque encrypted or unknown-codec frames, append-only suppression, optional signatures and encryption, and cross-language conformance through a shared vector corpus.

1.2 Status of this document

Field	Value
Status	Working draft
Document version	0.9-draft
Wire-format major version	1, encoded in the segment header `"v"` field
Date	2026-06-18
Document DOI	https://doi.org/10.67342/6pta6imnmw/v1
Stability	Wire-format changes remain possible until v1.0
Change control	Blackcat Informatics / GTS governance process
Conformance	Defined by this document and the versioned vector corpus (§19)
Implementation versions	Package versions are independent release artifacts
Corpus version	The corpus is versioned separately from package releases

This specification is maintained in the `gmeow-gts` repository, alongside four interoperable reference engines (Rust, Python, Go, TypeScript) that gate against the shared vector corpus. Report errata and propose changes there. Core semantic changes, registry additions, and optional-standard profile promotion follow the GTS governance process.

GTS is ontology-independent. GMEOW is a primary downstream consumer and distribution use case for GTS, but GTS readers and writers do not require GMEOW vocabulary, tooling, or semantics. Domain-specific profiles, including GMEOW and music-package profiles, are layered above the core format.

1.3 Document history

This section records changes to this specification document. Package releases, package version numbers, and per-engine release notes are separate artifacts and are not implied by the document version.

Changes in v0.9-draft (2026-06-18):

- Aligns publication metadata with the current v1.0-rc1 preparation state while keeping package versions independent from the spec document version.
- Clarifies conformance scopes, reader/writer classes, streaming-reader memory bounds, and canonical reader diagnostics.
- Formalises the graph fold, multi-segment value union, blank-node scoping, RDF 1.2 triple-term and `rdf:reifies` mapping, position constraints, and duplicate/conflict behavior.

- Adds streamable-layout rules, optional index/MMR proof preimages, proof verification, unknown extension-key behavior, media type and HTTP serving contracts, and durability/security considerations.
- Expands vector-corpus and manifest references so conformance claims name corpus revisions, subsets, tiers, modes, and release-stamped manifest artifacts.
- Pins the RDF 1.2 substrate to the 07 April 2026 W3C Candidate Recommendation Snapshot and states which RDF semantics GTS imports.

Earlier v0.3 document notes:

- Multi-segment files (`cat`-append composition, §3.1); segment-scoped term-ids (§7.2); per-segment fold and value-union semantics (§7.5); cross-segment suppression (§11); profile union and per-section language-tag discipline (§13); composition-tool requirements (§14.1); conformance vectors 15–21 (§19).
- Layout states and the streamable claim (§3.3, §5); streamable compaction with detached frame signatures (§10.1); the `stream` vocabulary (§13.3); the `compact` verb (§14.1); conformance vectors 24–26 (§19).

1.4 Table of contents

- 1. Overview and non-goals
- 2. Terminology and conformance
  - 2.1 Conformance scopes
  - 2.2 Reader and writer conformance classes
  - 2.3 Baseline reader API shape
  - 2.4 Reader diagnostics
- 3. File structure
  - 3.1 Multi-segment files (`cat`-append composition)
  - 3.2 Streaming and progressive enhancement
  - 3.3 Layout states: accretive and streamable
- 4. CBOR conventions
- 5. Header
- 6. Frames
  - 6.1 Payload resolution
  - 6.2 Index frame (optional)
- 7. Graph data model and fold
  - 7.1 Terms (`terms` frame)
  - 7.2 Term-id assignment (normative)
  - 7.3 Quoted triples and reifiers (`reifies` frame)
  - 7.4 Quads and annotations
  - 7.5 Fold algorithm (normative)
  - 7.6 Opaque nodes
  - 7.7 Streaming fold and bounded memory
  - 7.8 Duplicates and conflicts (normative)
- 8. Transform catalog
  - 8.1 Classes
  - 8.2 Stacking
  - 8.3 Capability model and graceful degradation
  - 8.4 Mandatory core set and durability
  - 8.5 Canonical codec registry (v1)
- 9. Integrity and confidentiality
  - 9.1 Per-frame self-hash and content-id chain (mandatory)
  - 9.2 Signatures (optional, algorithm-agile)
  - 9.3 Encryption (optional)
  - 9.4 The opacity invariant (normative)
- 10. Compaction

  - 10.1 Streamable compaction (ordering-only)
- 11. Suppression (additive "deletion")
- 12. Binary and content-addressing
  - 12.1 Nested GTS (recursive composition)
- 13. Profiles
  - 13.1 Language-tag discipline (profile-level normative)
  - 13.2 The `files` profile (optional-standard)
  - 13.3 The `stream` vocabulary (optional-standard)
  - 13.4 Domain profile example: `music-package` (informative)
- 14. Transforms out
  - 14.1 Composition tooling requirements (normative for conformant tools)
  - 14.2 Archive tooling (`files` profile)
- 15. Worked examples
  - 15.1 Minimal distribution snapshot (`dist`)
  - 15.2 Evidence: image + signed accrual (`evidence`)
  - 15.3 Notary: partially-opaque frame (`opaque`)
  - 15.4 Graceful degradation (`image`, content negotiation)
  - 15.5 Matryoshka: a whole signed GTS sealed inside a frame (`bundle` / `opaque`)
- 16. Media type and HTTP serving contract
  - 16.1 Media type and file extension (normative)
  - 16.2 File identification algorithm (normative)
  - 16.3 HTTP serving semantics (normative)
  - 16.4 Immutability-aware caching (normative)
- 17. Versioning and durability guarantees
- 18. Security considerations
- 19. Conformance test vectors
- 20. IANA considerations
- 21. Complete CDDL appendix
  - 21.1 Sequence grammar
  - 21.2 Copyable CDDL
- 22. Hash, signature, and extension-key preimages
  - 22.1 Preimage and subject table
  - 22.2 Unknown extension-key behavior
- 23. References

1.5 1. Overview and non-goals

GTS encodes a graph as an **append-only log of CBOR frames**. The logical graph is the *fold* (replay) of the log. Growth is an append; "deletion" is **suppression**, never a physical removal; optimisation is a separate, explicitly **lossy** compaction that rewrites the log into a snapshot.

Four properties define the format:

1. **CBOR all the way down** (RFC 8949). One ubiquitous, IETF-standardised binary encoding with native byte strings (no base64 tax), deterministic encoding (clean content hashes), and CBOR Sequences — concatenated data items with no enclosing length, so append is cheap. A reader needs only a CBOR library.
2. **A durable transform catalog.** Each frame's payload carries a *stackable* chain of codecs drawn from an open, long-lived catalog (`identity`, `base64`, `base85`, `gzip`, `zstd`, `lzma2`, `cose-encrypt`, …). The catalog separates *structure durability* (CBOR + this spec, forever) from *density and confidentiality* (swappable codecs).
3. **Integrity by construction.** Every frame carries an independent **BLAKE3 self-hash** (a content-id) and names its predecessor's id — a git-style content-addressed chain. Verification is **parallel**, a damaged frame is **independently detectable** (and the survivors recoverable given an intact index, §9.1), and the head id transitively commits to all history. Cryptographic signatures and encryption

(COSE, RFC 9052) are optional, layered, and algorithm-agile.

4. **Recursive composition (matryoshka).** A payload, after its transforms are reversed, is just bytes — and a GTS file is just bytes. So a payload MAY itself be a complete GTS, wrapped in any transform (compressed *or* encrypted). A whole signed graph can ride inside an encrypted field, with its own independent header, chain, and signatures (§12.1).

Non-goals. GTS does not define a query language, an index format mandatory for reading, a reasoner, or a mutation protocol. Random-access query, deep traversal, and SPARQL are the job of a transform target, not of GTS.

Informative motivation. GTS keeps the baseline reader surface small: a reader needs CBOR, BLAKE3, the mandatory codecs, and the fold rules rather than an RDF text parser. Tools that need richer query, indexing, or analytics project the folded data to an operating substrate such as N-Quads, SQLite, DuckDB, or Parquet.

1.6 2. Terminology and conformance

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHOULD**, **MAY**, and **OPTIONAL** are to be interpreted as described in BCP 14 (RFC 2119, RFC 8174).

- **Log** — the ordered sequence of frames in a GTS file.
- **Frame** — one CBOR data item in the log (§6).
- **Fold** — the deterministic replay of the log into a graph state (§7.5).
- **Term** — an RDF term (IRI, literal, blank node, or quoted triple) with a stable integer id.
- **Reifier** — a term that denotes a quoted triple, carrying statement-level metadata (RDF 1.2).
- **Capability** — what a reader must hold to decode a payload: a *codec library* or a *key*.
- **Opaque node** — the graph representation of a frame the reader could not decode (§7.6).

1.6.1 2.1 Conformance scopes

This specification separates the following conformance scopes:

- **Wire-format conformance** covers the byte-level CBOR Sequence structure, deterministic CBOR encoding, header and frame grammar, content-id preimages, and segment boundaries.
- **Reader conformance** covers parsing, chain verification, payload resolution, fold behavior, diagnostics, opaque-node handling, and resource-bound behavior.
- **Writer conformance** covers deterministic output, valid headers and frames, correct content identifiers, codec declarations, and signature/hash preimages.
- **Tool conformance** covers command-line or library policy that is stricter than local file validity, such as validating composition, extraction, publication, or archive operations.
- **Profile conformance** covers profile-specific vocabulary, validation, capability, and trust rules layered above the core format.
- **Deployment conformance** covers serving and distribution behavior such as media type, caching, range requests, and byte-preservation across HTTP or artifact hosting.

The conformance classes below define reader and writer behavior. Tool, profile, and deployment requirements are scoped explicitly in the sections that define them.

Baseline reader/writer conformance is independent of profile validation, CLI verbs, transform targets, and HTTP deployment behavior. A locally valid GTS file remains locally valid when it declares an unsupported profile; a reader records the profile declaration and folds the bytes according to its reader class, while a profile-aware tool MAY apply additional checks in the profile conformance scope.

Profiles, tools, and deployments MUST NOT change header or frame grammar, segment-boundary detection, content-id or signature/hash preimages, transform-catalog resolution, or the core fold semantics in §7. A stricter profile MAY reject an otherwise valid artifact only as a profile-level validation failure, not by redefining core GTS validity.

1.6.2 2.2 Reader and writer conformance classes

- A **Baseline Reader** MUST: parse the CBOR sequence; verify the id/prev chain (§9.1); fold `terms`, `quads`, `reifies`, `annot`, `blob`, `suppress`, `meta`, and `snapshot` frames; support the `identity`, `gzip`, and `zstd` codecs; and surface any frame it cannot decode as an opaque node (§7.6). It MAY ignore signatures and encryption.
- A **Streaming Reader** is a Baseline Reader that processes frames one at a time and emits to a sink **without materialising the whole graph**: it maintains only the term dictionary (and a running chain check), plus the maximum decoded frame size and validation sidecar state, giving O(distinct terms + maximum decoded frame size + validation sidecar state) retained memory rather than O(triples + blobs) (§7.7). The `gts → duckdb/sqlite` transforms (§14) are Streaming Reader-shaped when implemented through a non-materializing sink.
- A **Full Reader** additionally verifies COSE signatures, decrypts COSE-encrypted frames for which it holds keys, MAY recurse into nested GTS blobs (§12.1), and MAY use the optional index frame (§6.2) for parallel verification and random access.
- A **Writer** MUST emit deterministic CBOR (§4) for any bytes that are hashed or signed, and MUST compute each frame's `"id"` self-hash and set `"prev"` to the previous item's `"id"`.

1.6.3 2.3 Baseline reader API shape

A Baseline Reader SHOULD expose at least:

```
open(bytes|path)            -> Graph          # parse + verify chain + fold
Graph.quads()               -> iterator[(s,p,o,g)]   # term ids resolved to terms
Graph.term(id)              -> Term
Graph.annotations(reifier)  -> iterator[(prop, value)]
Graph.blob(digest)          -> bytes | OpaqueRef
Graph.opaque()              -> iterator[OpaqueNode]
Graph.to_nquads(out)        # §14
```

This API shape is intentionally small: it exposes the folded tables, diagnostics, and common projection path without requiring an RDF text parser, prefix resolver, query engine, or reasoner. The cross-language API and CLI parity contract is maintained in GTS-API-CLI-PARITY.md.

1.6.4 2.4 Reader diagnostics

Readers surface machine-observable diagnostics with these canonical classes (an implementation MAY map them to error returns or structured warnings):

class	meaning
`EmptyFile`	empty byte stream or no segment header; return an empty result with a fatal diagnostic rather than aborting (§3)
`TornAppendError`	trailing incomplete CBOR item at EOF (§3)
`DamagedFrame`	self-`"id"` mismatch / invalid frame hash (content corruption); opaque `reason:"damaged"` (§7.6)
`BrokenChain`	valid frame hash, but `"prev"` ≠ the previous item's `"id"` (insertion / reorder / splice) (§9.1)
`TruncatedLog`	a head commitment is present but the observed head differs (§9, §18)
`UnknownCodec`	a transform names a codec the reader lacks; opaque `reason:"unknown-codec"`
`MissingKey`	an `encrypt` codec the reader cannot decrypt; opaque `reason:"missing-key"`

class	meaning
`KeyWrapFailed`	a deferred multi-recipient key unwrap failed; opaque `reason:"missing-key"`
`ConflictingReifier`	a reifier rebound to a different triple (§7.8)
`PositionConstraint`	a term appears in an illegal subject, predicate, object, or graph-name position; reject/diagnose the offending row (§7.4)
`ForwardReference`	a term-id reference names a term not introduced by an earlier frame in the same segment (§7.2, §7.5)
`SegmentBoundary`	a compatibility reader reaches a later segment header where file-global term ids would misfold; stop with a fatal diagnostic (§3.1, §19)
`RecursionLimit`	nested-GTS depth or decoded-size budget exceeded (§12.1, §18)
`StreamableLayoutError`	a segment claims `"layout": "streamable"` but its covered region violates delivery ordering, or its index footer is missing or contradicts the frames it covers (§3.3)
`IndexMmrError`	an optional `index.mmr` root is present but does not match the covered frame ids (§6.2)
`UnknownFrameType`	a frame type is not understood by the reader/profile; preserve chain verification and either ignore it or surface it as opaque until a profile handles it (§7.8)

1.7 3. File structure

A GTS file is a **CBOR Sequence** (RFC 8742): zero framing bytes between items, each item a well-formed CBOR data item. Each segment MAY begin with the CBOR self-describe tag `55799` (`0xd9 0xd9 0xf7`) as a magic number for that segment. If present, tag `55799` MUST tag the segment **Header** data item; it is not a separate log item, has no `"id"`, and does not participate in the id/prev chain. A GTS file MUST NOT wrap the whole sequence in an outer CBOR item.

```
GTS-file = segment *segment
segment  = [self-describe-tag] header *frame
```

- The **first** data item of a segment MUST be a **Header** (§5).
- Every subsequent data item of a segment is a **Frame** (§6), in log order, until the next Header (which begins a new segment) or end of input.
- **Append** = concatenate one more frame (extending the last segment), or concatenate a whole further segment (§3.1). No length prefix or count is stored, so a writer never rewrites earlier bytes.

1.7.1 3.1 Multi-segment files (cat-append composition)

A GTS file is one or more **segments**, each a complete, self-contained `header *frame` log. The defining property: **byte-concatenation of valid GTS files is a valid GTS file** —

```
cat music.gts >> core.gts        # core.gts is now a valid two-segment GTS
```

- **Boundary detection (normative).** A reader that has consumed at least one frame and encounters a data item that is a map containing the key `"gts"` and lacking the key `"t"` MUST treat it as the Header of a **new segment** (the optional self-describe tag `55799` MAY tag that header; writers SHOULD emit the tag on every segment header to make boundaries human-recognizable). The tag is attached to the segment header, so byte-concatenation of independently valid tagged segments remains byte-concatenation of CBOR Sequence items, not a nested whole-file wrapper. Any other non-frame item remains malformed input (§17).

- **Independent integrity.** Each segment has its own genesis (its header `"id"`), its own id/prev chain, its own signatures, and its own optional `index` (an index covers ONLY its segment). The file's composite identity is the **ordered list of segment head ids**. A third-party segment carries its own signer; concatenation rewrites nothing (a `cat` cannot rewrite an earlier segment's header without breaking its self-hash — by design).
- **Identity across segments.** Term-ids are **segment-scoped** (§7.2); the ONLY cross-segment identity is the term **value** (IRI, literal, quoted-triple structure). Blank-node labels are segment-local and MUST NOT be merged across segments (the §12.1 nested-GTS rule, applied at the top level).
- **Profiles union.** The file's effective profile/requirement set is the union of the segment headers' `"prof"` values (and any profile requirements carried in segment metadata). A reader lacking the capabilities a segment requires degrades that segment's frames to opaque nodes (§7.6) — "this data needs the gmeow-music profile" is a header read, not an error.
- **Relationship to nesting.** Nested GTS (§12.1) composes by *containment* (a sealed, independently-shippable subgraph); segments compose by *concatenation* (open, tool-free aggregation). Both yield a union fold; choose nesting when the part must travel or seal independently, segments when plain `cat` must work.

1.7.2 3.2 Streaming and progressive enhancement

The append-only log makes streaming a **property of the format**, not a feature of a tool. Three facts compose, and conformant implementations MUST preserve all three:

- **Prefix-fold validity (normative).** Every byte prefix of a valid GTS file that ends on a data-item boundary is itself a valid GTS file, and a reader MUST fold it to exactly the state it would reach folding those same items inside the complete file. A live stream in flight is therefore *indistinguishable* from a file with a torn append (§3): the partial trailing item means "not yet arrived", and a consumer MAY keep reading as bytes land (`tail -f` semantics) — every intermediate fold is a real, usable graph state, never a half-parsed error state.
- **Monotone refinement.** Appended frames only ever *add* knowledge: quads accumulate (§7.8 set semantics), a reifier binding is first-wins so an established rendering never changes under it, and suppression is an additive display overlay (§11) — arrival of a `suppress` frame refines presentation without invalidating any prior fold. The chain check is likewise incremental: O(1) state (the expected `"prev"`) verifies each frame as it arrives.
- **Chunk-safe framing.** CBOR Sequence items are self-delimiting, so item boundaries are safe re-chunking points for relays and proxies, and resumption is content-addressed: a receiver that states the last frame `"id"` it verified can be resumed from the next byte with no negotiation beyond that hash.

Progressive enhancement. Producers SHOULD order content most-significant-first so an early prefix is maximally useful: within a segment, `terms`/`quads` (the graph) before bulky `blob` frames, and small or preview manifestations before large ones; across a file, segments ARE the enhancement layers — a base segment (core graph + thumbnails) followed by enhancement segments (full-resolution blobs, computed projections) gives a receiver a complete, verifiable package at every segment boundary, §3.1's composition rules applied as a delivery schedule. **Checkpoint `index` frames** (§6.2) emitted periodically give a streaming consumer intermediate truncation anchors (`"head"`), random-access offsets for ranged re-fetch, and a manifest of what has arrived; the index remains an accelerator, never a dependency (§3, §6.2).

The manifest is the graph. GTS needs no table-of-contents structure, because the frames that *describe* content can precede the frames that *carry* it: a producer SHOULD emit the quads naming each upcoming manifestation — its content digest, media type, size, role — before the `blob` frames whose bytes they promise. The fold of an early prefix then contains the delivery schedule as ordinary knowledge: every digest the graph names but the stream has not yet delivered is a content-addressed IOU, so "stop here", "skip ahead" and "range-fetch only the RAW file" are *informed* consumer decisions, taken against a verifiable catalog rather than a guess. (A blob that never arrives in this file is simply an external blob, §12 — the reference degrades gracefully to "bytes live elsewhere".)

Worked delivery schedule — a photograph as a progressive stream; a consumer may stop at any item boundary

with a complete, verified package of everything above its stopping point:

```
header                          profile, codec catalog
terms/quads                     the catalog: Work + every manifestation below,
                                each with digest, mt, size, role (the IOUs)
blob  image/webp        ~20 KB  thumbnail - first paint
blob  image/jxl         ~8 MB   full-resolution render
terms/quads                     scene description (what is IN the image)
blob  image/x-raw       ~80 MB  RAW sensor dump
meta/quads                      full camera metadata
terms/quads/annot               AI analysis as RDF, statement-level provenance
terms/quads/annot               opinions - standpoint-qualified claims
terms/quads                     processing-pipeline provenance
index                           footer: offsets, head anchor, MMR (§6.2)
```

A casual viewer stops after the thumbnail; an archivist takes everything; an editor range-fetches the RAW by digest after reading only the catalog. Same bytes, same chain, three consumers.

A reader streams items until end of input. Trailing partial bytes (a torn append) MUST be detected and ignored with a diagnostic: a reader attempts to decode each successive CBOR item, and if the decoder signals an incomplete item or unexpected EOF at end-of-file, it MUST treat the trailing bytes as a torn append, ignore that incomplete item, and surface a machine-observable diagnostic (e.g. a `TornAppendError` warning). In particular, if a crash occurred while writing an `index` frame (§6.2) the trailing index is torn: a reader MUST ignore it and fall back to an earlier intact `index` or to a plain **sequential scan**, so every surviving frame remains recoverable. The optional index is an accelerator, never a dependency.

Every property above holds for any frame order; what a producer *chooses* as the order is a separate, named concern: a segment is in one of two **layout states** — **accretive** (append-ordered) or **streamable** (delivery-ordered) — defined next (§3.3).

1.7.3 3.3 Layout states: accretive and streamable

A GTS segment is always valid and always prefix-foldable (§3.2), but it lives in one of two layout states:

- **Accretive** — append-optimized. Frames land in arrival order (live capture, agent memory accrual, evidence accrual). Writes are cheap forever and the stream is consumable as it lands, but significance is not front-loaded and the catalog may trail the bytes it describes. This is the default state; it is never declared.
- **Streamable** — delivery-ordered. The catalog *presages* the payload: a **leading streaming index** (ordinary `terms/quads` frames in the `stream` vocabulary, §13.3 — one `stream:Manifestation` per promised blob, carrying digest, media type, size, role, and intended order) precedes every `blob` frame, blobs follow most-significant-first, and a trailing offset `index` (§6.2) closes the covered region as the random-access footer.

Append-friendly and stream-optimal are different *layouts of the same content* (precedent: mp4 `faststart`, zip central-directory rewrites, LSM compaction). A one-pass writer cannot produce the second state, so conversion is an explicit rewrite — **streamable compaction** (§10.1), exposed as `gts compact --streamable` (§14.1).

The claim (normative). A segment declares the streamable state with the optional header key `"layout": "streamable"` (§5). The claim is per-segment (each segment has its own header, §3.1) and tamper-evident (the header self-hash covers it). Streamability is a **declared-vs-computed claim** in the sense of §14.1 — refuse-don't-trust:

- The **covered region** of a claimed segment is the prefix delimited by the segment's **last intact `index` frame**: `"count"` frames, ending at the frame whose `"id"` equals the index's `"head"`. The footer MUST immediately follow the frames it covers (`"count"` = the index's own frame position − 1) — otherwise frames could sit between the covered prefix and the footer, counted neither as covered nor as accretive

tail. A claimed segment with no intact `index` frame, whose last index is not immediately adjacent to its covered prefix, or whose `"head"` does not equal the id of frame `"count"`, is in violation.
- Within the covered region, every inline `blob` frame MUST be preceded by a `quads` frame that describes its digest via `stream:digest` (§13.3) — catalog-before-payload. A covered blob delivered before its description is in violation.
- A reader encountering a violation MUST surface a **`StreamableLayoutError`** diagnostic (§2.3); a verifying tool treats it as an error (§14.1). The claim can never rot against the bytes.

Appends after compaction are legal and foldable. Frames after the last `index` are simply *unpresaged*: they are the segment's **accretive tail**, carry no ordering obligation, and trigger no diagnostic. The segment is then "streamable through frame *N*, accretive after" — tooling SHOULD report the boundary (§14.1). Re-compact to re-streamline. Likewise, a segment appended by `cat` makes no claim unless its own header claims.

In-flight prefixes. A prefix of a streamable segment cut before the trailing `index` has, by construction, a claim and no footer yet; a streaming consumer MUST NOT treat the missing footer as a lie while input may still be arriving — the missing-footer violation applies to a *complete* file. The catalog-before-payload rule, by contrast, is prefix-stable: a violation observed in any prefix is a violation of the whole file.

1.8 4. CBOR conventions

- Maps use **short text-string keys** (e.g. `"t"`, `"d"`) for self-description and eyeball debuggability; compactness is the transform layer's job, not the schema's.
- Any bytes that are **hashed or signed** MUST use **Deterministic Encoding** (RFC 8949 §4.2): shortest-form integers, definite-length items, and map keys sorted **bytewise on their encoded form** — explicitly the RFC 8949 rule, NOT RFC 7049's length-first canonical ordering. (For the short text keys GTS itself uses the two coincide, because a CBOR text string's initial byte embeds its length; the rules diverge on mixed-type keys, so implementations MUST NOT rely on a CBOR library's legacy "canonical" mode without checking which ordering it implements.)
- Unsigned integers are used for all ids. BLAKE3 digests are 32-byte (256-bit) byte strings.
- Short grammar fragments are given in **CDDL** (RFC 8610). The complete copyable CDDL appendix is §21, and the canonical preimage rules are §22.

```
term-id      = uint            ; append-order, frozen (§7.2)
digest       = bstr .size 32   ; BLAKE3-256
content-id   = digest          ; a frame's self-hash (§9.1)
digest-ref   = digest / tstr    ; raw digest or "blake3:<hex>" text (§21.2)
codec-id     = uint            ; index into the header codec catalog (§8)
```

1.9 5. Header

The Header is the first data item and the chain genesis; it is not a frame (it has no `"prev"`).

```
header = {
  "gts"  : "GTS1",                    ; magic / format id
  "v"    : uint,                      ; spec major version (1)
  "prof" : tstr,                      ; profile (§13); "generic" if unspecified
  "cat"  : { * codec-id => codec },   ; the transform catalog (§8)
  ? "layout": tstr,                   ; layout-state claim (§3.3); absent = accretive
  ? "dct": { * tstr => bstr },        ; named, UNCOMPRESSED dictionaries for dict-codecs
  ? "meta": any,                      ; free-form, non-normative metadata
  "id"   : content-id,                ; self-hash of the header content (the chain genesis)
}

codec = {
  "name" : tstr,                      ; "identity" | "gzip" | "zstd" | "lzma2" | "cose-encrypt" | ...
```

```
  "cls"  : "encode" / "compress" / "encrypt",
  ? "dct": tstr,                      ; references header "dct" key (dict codecs)
  ? "p"  : any,                       ; codec parameters (e.g. lzma2 level)
}
```

The catalog is **closed within a file** (a frame may only reference codec-ids the header declares) but **open across the ecosystem** (new codecs may be registered by name). The Header carries its own `"id"` (self-hash of its content) and no `"prev"` — it is the genesis, and the first frame's `"prev"` is the Header's `"id"`. The Header `"id"` MUST equal the BLAKE3-256 of the deterministic CBOR of the Header map **excluding the `"id"` key**; all other keys (including `"meta"` and unknown extension keys) participate. The preimage table in §22 is the single source of truth for hashed and signed bytes. The optional `"layout"` key claims a layout state (§3.3): the only value defined by this revision is `"streamable"`, which a verifying reader MUST check against the segment's actual layout; readers MUST ignore unknown `"layout"` values (forward compatibility — an unknown state imposes no check). Dictionaries are stored **uncompressed and in-band** — there is no external-dictionary dependency. A codec's `"dct"` value MUST match a key in the header `"dct"` map, and the codec MUST use the corresponding byte string as its compression/encoding dictionary.

1.10 6. Frames

All frames share one envelope:

```
frame = {
  "t"   : frame-type,         ; discriminator
  ? "x" : [+ codec-id],       ; transform chain, applied in order on encode; default [identity]
  ? "pub": any,               ; CLEARTEXT public envelope (always readable; §9.4)
  ? "to": [+ recipient],      ; recipients, for encrypt-class chains
  ? "d" : bstr / any,         ; payload: bstr when "x" transforms it; structured CBOR otherwise
  "prev": content-id,         ; the PREVIOUS data item's "id" (chain link; §9.1)
  "id"  : content-id,         ; BLAKE3-256 self-hash of this frame's CONTENT (all keys but "id"/"sig")
    ? "sig": bstr,            ; COSE_Sign1 over "id" (§9.2)
}

frame-type = "terms" / "quads" / "reifies" / "annot" / "blob" / "suppress"
/ "snapshot" / "meta" / "index" / "opaque"

recipient = { "kid": tstr, ? "alg": tstr, * tstr => any }   ; key identifier; never the key
```

Each frame's `"id"` MUST equal the BLAKE3-256 of the deterministic CBOR of its content (every key except `"id"` and `"sig"`; unknown extension keys participate). Each frame's `"prev"` MUST equal the previous data item's `"id"`; the **first** frame's `"prev"` is the Header's `"id"`. Because `"prev"` is inside the hashed content, each `"id"` transitively commits to all prior frames (§9.1). §22 centralizes the complete preimage and subject rules.

1.10.1 6.1 Payload resolution

To obtain a frame's logical payload:

1. If `"x"` is absent, the payload is `"d"` directly (structured CBOR) — equivalent to a single `identity` transform; a chain resolving to only `identity` likewise leaves `"d"` unchanged.
2. If `"x"` is present, `"d"` MUST be a byte string and every codec-id MUST resolve through the header `"cat"`; apply the **reverse** of each codec, last to first. Each step requires a **capability** (§8.3). On any missing capability (unknown codec or missing key), stop and treat the frame as **opaque** (§7.6).
3. The fully-decoded bytes are a CBOR item; decode them to the type-specific structure (§7).

1.10.2 6.2 Index frame (optional)

A writer MAY append an `index` frame — a footer that accelerates large files without raising the simple-reader floor (a Baseline Reader ignores it). Because the log is append-only, a fresh `index` MAY be appended after more frames; the **last** `index` wins.

```
index-payload = {
  "count"  : uint,                          ; frames covered
  "head"   : content-id,                    ; "id" of the last covered frame (truncation anchor)
  ? "off"  : [+ uint],                      ; byte offset of each frame (random access; parallel verify)
  ? "ti"   : { * frame-type => [+ uint] },; frame indices by type
  ? "dict" : [+ uint],                      ; indices of "terms" frames (dictionary locator; §7.7)
  ? "mmr"  : content-id,                    ; Merkle-Mountain-Range root over frame ids (§9.1)
}
```

Given `"off"`, a Full Reader dispatches frame-hash verification across threads and seeks to any frame; given `"dict"`, a Streaming Reader loads only the dictionary (§7.7); given `"head"`/ `"mmr"`, it detects truncation and produces O(log n) inclusion proofs. A **checkpoint** index is simply an `index` emitted periodically rather than only as a footer; an earlier `index` MAY still serve as a recovery anchor even though the last intact `index` is preferred for acceleration. Current package support and deferrals for `off`/`ti`, `dict`, `mmr`, proof verbs, range fetch, and replication workflows are tracked in GTS-ADVANCED-PRIMITIVES.md.

The `mmr` root uses a Merkle-Mountain-Range over the ordered frame ids covered by the index. The index frame itself is not covered by its own `mmr`; a later index may cover an earlier index frame like any other earlier frame. Peaks are built left-to-right by appending one leaf per frame id and repeatedly merging the two rightmost peaks while their heights match. The root for `count = 0` is the root preimage with an empty peak list.

All MMR preimages use deterministic CBOR (§4) and BLAKE3-256:

```
leaf(index, frame_id) =
  BLAKE3-256(deterministic-CBOR(["gts-mmr-leaf-v1", index, frame_id]))

parent(parent_height, left_hash, right_hash) =
  BLAKE3-256(deterministic-CBOR(["gts-mmr-parent-v1",
                                 parent_height, left_hash, right_hash]))

root(count, peaks) =
  BLAKE3-256(deterministic-CBOR(["gts-mmr-root-v1",
                                 count,
                                 [[peak_height, peak_hash], ...]]))
```

A detached proof JSON object has this stable shape:

```
{
  "schema": "gts-mmr-proof-v1",
  "hash": "blake3-256",
  "preimage": "gts-mmr-v1",
  "count": 4,
  "leaf_index": 2,
  "frame_id": "<32-byte frame id hex>",
  "root": "<32-byte mmr root hex>",
  "peak_index": 0,
  "peaks": [{"height": 2, "hash": "<32-byte peak hex>"}],
  "path": [
    {"side": "right", "parent_height": 1, "hash": "<32-byte sibling hex>"}
  ]
}
```

`verify-proof` MUST reject a proof unless the peak heights match `count`, `leaf_index` belongs to `peak_index`, every hash field is 32 bytes, the path reconstructs the selected peak, and the peaks reconstruct `root`. Proof verification does not require the original `.gts` file.

1.11 7. Graph data model and fold

A fold is the deterministic projection of an ordered frame log into a value-indexed graph state. Term ids are local compression artifacts used while reading a segment; the exposed file graph is defined by the value-union rules below.

Folded state model (normative). A reader folds each segment into this logical state:

- `terms`: an ordered vector of segment-local term values.
- `quads`: a set of asserted RDF quads over term values.
- `reifiers`: a partial map from a reifier term value to exactly one quoted triple value.
- `annotations`: an ordered multiset of `(reifier, predicate, value)` rows over term values.
- `blobs`: a content-addressed map from BLAKE3 digest to inline bytes or an external content reference.
- `blob_meta`: a shallow metadata map per blob digest, built from blob `"pub"` maps.
- `meta`: a shallow segment metadata map, plus a file-level shallow merge (§7.5).
- `suppressions`: an ordered list of display/precedence directives (§11).
- `opaque`: an ordered list of frames intentionally or necessarily carried without decoded payload semantics (§7.6).
- `signatures` and `diagnostics`: ordered reader observations about frames. They are part of the folded GTS state but not part of the RDF dataset.
- `segment_heads`, `segment_profiles`, `segment_meta`, and segment layout state: the ordered segment ledger needed to preserve cat-append identity, profile requirements, per-segment metadata, and streamable-layout claims.

File fold (normative). A file fold is the ordered value-union of its segment folds: segments are processed in file order; every term id is first resolved within its own segment; then quads, reifier bindings, annotations, blob declarations, metadata, suppressions, opaque nodes, signatures, diagnostics, and segment ledgers are merged by the rules in §7.5 and §7.8. The union MUST NOT compare raw term ids across segments. Cross-segment identity is always value identity.

RDF substrate (normative). GTS imports the RDF 1.2 Concepts and Abstract Data Model Candidate Recommendation Snapshot dated 07 April 2026 for IRIs, blank nodes, literals, RDF datasets, triple terms, version label `"1.2"`, and `rdf:reifies` (§23.1). GTS freezes that substrate for major version 1 unless a later GTS major version updates this reference. A Baseline Reader need not implement an RDF parser, query language, entailment regime, canonicalization algorithm, or RDF 1.2 concrete syntax; it only needs the term, quad, reifier, annotation, and dataset mapping defined here. RDF Semantics entailment regimes are not part of the core GTS fold unless a profile or projection explicitly applies them above the transport layer.

Value equality (normative). The fold compares values as follows:

value kind	equality rule
IRI	Exact Unicode string equality after CBOR decoding. No percent, case, Unicode, base-IRI, or prefix normalization is applied by core GTS.
Literal	Same lexical string, same datatype IRI value after defaulting (§7.1), and same language-tag string when present. Datatype lexical canonicalization is not applied; `"01"^^xsd:int` and `"1"^^xsd:int` are distinct transport values.

value kind	equality rule
Language tag	Exact string equality in core GTS. Profiles and projections MAY apply language-range matching, preferred BCP 47 casing, or public/private tag translation (§13.1), but that is not term identity.
Datatype	Equality of the datatype IRI value, not the local term id that names it.
Blank node	Equality is scoped to the blank-node scope plus the non-empty label. Blank nodes from different segments or nested GTS files are never equal. A blank node with absent or empty `"v"` is anonymous: each term entry is a distinct blank node within its scope.
Quoted triple term	Equality of the resolved subject, predicate, and object term values of the quoted triple. Quotation alone does not assert the triple (§7.3).
Graph name	Equality of the graph-name term value. An absent graph slot is the default graph and is not equal to any named graph.
Blob	Equality of the normalized BLAKE3 digest (`blake3:<hex>` or raw digest bytes normalized to that form); inline byte equality is proved by the digest.
Opaque node	Equality of an opaque occurrence is its segment identity plus frame content id. Exact duplicate presentations MAY be collapsed by a display layer, but the fold preserves occurrence order.
Metadata	Map keys compare by exact string; values compare by deterministic CBOR data-model equality. The file-level view is a shallow later-wins merge, while per-segment originals remain addressable (§7.5).
Suppression	A suppression target first resolves in its own segment and then applies value-wise to the file union (§11). Repeated directives have idempotent display effect but are retained as ordered fold state.

1.11.1 7.1 Terms (`terms` frame)

Payload: an **ordered array** of terms. Ids are assigned by append order within the current segment dictionary (or within a `snapshot` dictionary before it is shifted into the enclosing segment).

```
terms-payload = [+ term]
term = {
  "k"   : 0 / 1 / 2 / 3,   ; 0=IRI 1=literal 2=bnode 3=quoted-triple
  ? "v" : tstr,            ; IRI string | literal lexical form | bnode label
  ? "dt": term-id,         ; literal datatype IRI (a term)
  ? "l" : tstr,            ; literal language tag (BCP 47)
  ? "rf": term-id,         ; quoted-triple: the reifier (§7.3) whose triple this term denotes
}
```

Literal datatype defaulting (normative). For a `k:1` (literal) term: if `"l"` (language tag) is present and `"dt"` is absent, the datatype is `rdf:langString`; if both `"l"` and `"dt"` are absent, the datatype is `xsd:string`.

Blank-node labels (normative). A `k:2` (blank node) term's non-empty `"v"` label is local to the current

blank-node scope: the segment for ordinary frames, the snapshot dictionary for a `snapshot`, or the nested GTS file for recursive composition (§12.1). It MUST NOT be treated as a globally stable identifier and MUST NOT be merged with the same label in another segment or nested GTS. If `"v"` is absent or the empty string, the term is anonymous and denotes a fresh blank node for that term entry within the scope. Transforms MAY relabel blank nodes while preserving blank-node isomorphism and scope separation.

1.11.2 7.2 Term-id assignment (normative)

Term ids are unsigned integers assigned **in append order, per segment**, starting at `0` at each segment's header, and are **frozen within their segment**: a term minted while folding frame *N* keeps its id for the rest of that segment. A `quads`, `annot`, or `reifies` frame at position *N* MUST only reference term-ids introduced at positions `0..N-1` **of the same segment** (such frames introduce no terms of their own). This makes writing pure-append, reading single-pass, and concatenation sound: term-ids are **compression artifacts, never identity** — cross-segment identity is the term *value* only (§3.1), exactly as a `snapshot`'s dictionary already restarts at `0` (§10). An implementation that applied file-global ids to a multi-segment file would misfold silently; the boundary rule (§3.1) and vector 17 (§19) exist to make that failure loud instead.

1.11.3 7.3 Quoted triples and reifiers (`reifies` frame)

RDF 1.2 lets a triple be the subject or object of another. GTS keeps quoted triples in the id domain: a **reifier** is an ordinary IRI/bnode term; a `reifies` frame binds it to the triple it quotes.

```
reifies-payload = { * term-id => [term-id, term-id, term-id] }  ; reifier => (s, p, o)
```

A quoted triple used as a node is a term with `"k": 3` and `"rf"` pointing at its reifier.

RDF dataset mapping (normative). A folded GTS graph maps to an RDF 1.2 dataset as follows: each `quads` row `(S,P,O,G?)` asserts the RDF triple `(S,P,O)` in the default graph when `G` is absent, or in the named graph `G` when `G` is present. A `reifies` binding `R => (S,P,O)` asserts the triple `R rdf:reifies <<( S P O )>>` in the default graph. A `k:3` term denotes that triple term, reached through its reifier `R`. Each `annot` row `(R, P', V')` asserts the triple `R P' V'` in the default graph. Profiles MAY define additional graph-placement conventions for projection, but the core mapping above is the interoperable baseline.

Quotation does not imply assertion (normative). Referencing a triple term, either via a reifier or a `k:3` term, does NOT assert the base triple `(S P O)`. The base triple is asserted iff it also appears in a `quads` frame.

RDF 1.1 degradation (informative). RDF 1.1 has no quoted triple term. A lossy RDF 1.1 projection MAY replace a quoted triple term with its reifier resource and emit ordinary reification-style triples such as `R rdf:subject S`, `R rdf:predicate P`, and `R rdf:object O`, or carry `R rdf:reifies` as an extension predicate understood by the consumer. Such a projection MUST NOT assert `(S P O)` merely because the GTS file quoted it, and tooling SHOULD label the projection as lossy whenever a triple term was present.

1.11.4 7.4 Quads and annotations

```
quads-payload = [+ [term-id, term-id, term-id, ? term-id]]  ; s, p, o, (g; default graph if absent)
annot-payload = [+ [term-id, term-id, term-id]]             ; reifier, predicate, value
```

Statement-level metadata (confidence, validity interval, standpoint/vantage, modality, …) is expressed as `annot` rows on a reifier. **Contested claims coexist**: several `annot` rows on one reifier, or several reifiers over one (s,p,o), are all retained — none is privileged. Annotations are an ordered multiset in the folded GTS state: readers MUST preserve row order within each segment and concatenate segment annotation rows in file order. Exact duplicate annotation rows are retained in the GTS fold; an RDF dataset projection MAY collapse identical emitted RDF triples because RDF datasets are set-valued.

Position constraints (normative). In a `quads` row the predicate `p` MUST be an IRI (`k:0`); the subject `s` MUST be an IRI, blank node, or quoted triple (`k:0|2|3`); the object `o` MAY be any term; and the graph name `g`, when present, MUST be an IRI or blank node (`k:0|2`) — never a literal or quoted triple. A `reifies`

triple `(S,P,O)` obeys the same subject/predicate/object constraints. In an `annot` row the predicate MUST be an IRI.

1.11.5 7.5 Fold algorithm (normative)

```
result := empty file state
          (terms, quads, reifiers, annotations, blobs, blob_meta, meta,
           suppressions, opaque, signatures, diagnostics, segment ledger)
for segment in file order:                        # §3.1; single-segment files: one iteration
  verify each frame's id (self-hash) and prev-link within the segment;
  record sig status if "sig" present
  terms := []   graph := {}   reif := {}   annot := []
  blobs := {}   blob_meta := {}   meta := {}   suppressed := []   opaque := []
  diagnostics := []
  for frame in segment log order:
    P := resolve payload (§6.1); if undecodable -> add opaque node (§7.6); continue
    switch frame.t:
      "terms"    : append each term (assign next id); each "dt"/"rf" MUST name an
                   already-introduced term-id (no forward references)
      "quads"    : add each (s,p,o,g) value tuple to graph
      "reifies"  : bind reifier to (s,p,o), keeping the first non-conflicting binding (§7.8)
      "annot"    : append (reifier, predicate, value)
      "blob"     : if "d" present -> blobs[BLAKE3(decoded "d")] := bytes (inline);
                   else -> register external blob by "pub".digest;
                   shallow-merge "pub" into blob_meta[digest]
      "suppress" : append directive to `suppressed` (display contract; §11)
      "snapshot" : load a self-contained fold wholesale (§10)
      "meta"     : shallow-merge map into segment meta (later keys overwrite earlier)
      "opaque"   : add explicit opaque node
  union segment fold into result BY TERM VALUE     # ids resolve locally, never cross segments;
                                                   # bnodes keep their scope (§3.1, §12.1)
result
```

The fold is deterministic: the same intact log yields the same value state in every conformant reader. Within a segment, `meta` accumulates as a shallow union over one map — a later frame's keys replace earlier ones; values are not concatenated. **Across segments**, each segment's folded `meta` is exposed per segment (keyed by segment head id) AND shallow-merged in file order into a file-level view — a later segment's keys win, but the per-segment originals remain addressable (a third-party segment's metadata is never silently absorbed).

1.11.6 7.6 Opaque nodes

When a frame's payload cannot be decoded — an unknown codec, or a `cose-encrypt` codec for which the reader holds no key — the reader MUST NOT drop it. It MUST add an **opaque node** to the graph carrying everything still in cleartext:

```
opaque-node = {
  "id"      : content-id,      ; the frame's self-hash
  "type"    : frame-type,      ; declared "t"
  ? "pub"   : any,             ; the cleartext public envelope, if any
  ? "to"    : [+ recipient],   ; declared recipients
  "sigstat" : "none" / "valid" / "invalid" / "unverified",
  "reason"  : "unknown-codec" / "missing-key" / "damaged",
}
```

Most opaque nodes are produced by a reader at decode time; a writer MAY also emit an explicit `opaque` frame (e.g. a redaction placeholder) whose payload is the structure above, in which case `"sigstat"` is omitted

(a reader determines it). A `damaged` frame (failed self-hash, or absent) is isolated and folded as an opaque node too (§9.1): a reader MAY surface its cleartext fields as **untrusted** diagnostic metadata, but MUST set `"sigstat"` to `invalid/unverified` and `"reason": "damaged"` — the bytes are not trustworthy. The frame still participates in the id/prev chain, so it cannot be silently stripped.

1.11.7 7.7 Streaming fold and bounded memory

A graph need not be materialised to be *transformed*. A **Streaming Reader** (§2.1) processes frames in order and emits to a sink, holding only the term dictionary, the current decoded frame or blob, and the running id/prev and validation state:

- `gts → duckdb/sqlite` (§14) keep the **integer-id** model: stream `terms` deltas into a `terms` table and `quads/reifies/annot` deltas into id-valued tables, bulk-inserting as frames arrive. **No term resolution and no graph materialisation occur** — memory is bounded by the dictionary, the largest decoded frame, and validation sidecar state rather than by triples or blobs. The relational join that resolves ids is the engine's job, later.
- `gts → ttl/nq` must resolve ids to emit text. If the dictionary exceeds memory, the reader uses the index `"dict"` locator (§6.2) to load (or memory-map, or spill to an on-disk kv) only the `terms` frames first, then streams the quads.

Even O(distinct terms + maximum decoded frame size + validation sidecar state) can exceed memory for pathologically irregular graphs (e.g. a crawl dumping millions of unique UUID IRIs, or a single very large inline blob). A Streaming Reader therefore MAY **flush its in-memory dictionary to a temporary on-disk key-value store** when a memory limit is reached, trading RAM for a local spill file; correctness is unaffected because term-ids are append-order and frozen (§7.2). The `gts → duckdb/sqlite` transforms get this for free — the target table *is* the spill.

The package-level streaming-sink claim boundary and memory benchmark helper are maintained in GTS-ADVANCED-PRIMITIVES.md.

A multi-gigabyte log thus transforms to an operating substrate in bounded memory — the resolve-and-materialise OOM failure mode is avoided by construction.

1.11.8 7.8 Duplicates and conflicts (normative)

All duplicate and conflict behavior is defined here so frame handlers do not invent local policy:

item	fold behavior
Duplicate terms	A writer SHOULD intern repeated terms, but each term entry still receives its own segment-local id. Non-blank values that compare equal by §7 are the same value in the file union. Anonymous blank nodes (`"v"` absent or empty) are fresh per term entry.
Duplicate quads	The folded graph is a set: identical `(s,p,o,g)` value rows collapse to one without diagnostic.
Reifier bindings	A reifier SHOULD have exactly one `reifies` binding. Repeated identical bindings are harmless. A conflicting binding is a data-quality error: the reader surfaces `ConflictingReifier`, keeps the first binding in file order, and ignores the conflicting binding for the reifier map.
Annotations	Annotation rows are an ordered multiset (§7.4). Multiple rows on one reifier coexist; exact duplicate rows are retained in the GTS fold. RDF dataset projections may collapse identical emitted triples.

item	fold behavior
Blob bytes	Blobs are addressed by digest. Repeating the same digest/bytes is idempotent; a content-addressed view stores one byte value per digest. Validating extraction re-hashes inline bytes against the requested digest (§14.1).
Blob metadata	`blob_meta[digest]` is a shallow map built in file order. Later metadata keys for the same digest replace earlier keys in the file-level view; earlier declarations remain in the original frames.
Suppressions	Suppression directives are additive. Repeating an equivalent directive has idempotent display effect but remains present in the ordered suppression list. There is no unsuppress frame (§11).
Metadata keys	Segment `meta` is shallow later-wins; file `meta` is the shallow later-wins merge of segment metadata in file order. Per-segment metadata remains addressable (§7.5).
Malformed frames	A frame whose payload cannot be decoded or whose handler cannot safely fold it becomes an opaque node with a diagnostic when recovery is possible (§7.6, §9.1). Surviving later frames are still foldable when item boundaries are known.
Unknown structural frame types	A Baseline Reader does not assign graph semantics to an unknown frame type. It MUST preserve chain verification and MAY surface an opaque node or diagnostic; a profile-aware Full Reader MAY interpret the frame.
Profile conflicts	Profile declarations and profile requirements union across segments (§3.1, §13). Unsupported profile requirements degrade the affected segment's unsupported payloads to opaque nodes or profile diagnostics; they do not invalidate core wire-format folding by themselves.

1.12 8. Transform catalog

1.12.1 8.1 Classes

Every catalog entry declares a **class**:

class	examples	capability needed to reverse
`encode`	`identity`, `base64`, `base85`	none (pure function)
`compress`	`gzip`, `zstd`, `lzma2`	a codec library
`encrypt`	`cose-encrypt0`, `cose-encrypt`	a **key** (per recipient)

1.12.2 8.2 Stacking

`"x"` is applied in array order on encode and reversed on decode. Example: `[zstd, cose-encrypt]` means *compress, then encrypt*; a reader decrypts (if keyed) then decompresses.

1.12.3 8.3 Capability model and graceful degradation

Decoding a chain requires **every** capability it names. A missing capability is uniform whether it is a library (`unknown-codec`) or a key (`missing-key`): the frame becomes an opaque node (§7.6). This single mechanism yields **in-file content negotiation** — a logical object MAY appear as several frames in different codecs/formats (e.g. a high-fidelity representation a reader can't decode *and* a widely-supported fallback it can), and the reader uses the best frame for which it holds the capabilities.

1.12.4 8.4 Mandatory core set and durability

A Baseline Reader MUST implement `identity`, `gzip`, and `zstd` — so a conformant reader's full dependency set is **CBOR + BLAKE3 + gzip + zstd**. Writers targeting maximum longevity SHOULD restrict to the core set. Density-oriented writers MAY use `lzma2` with an in-band dictionary. All core codecs are stable, widely deployed primitives.

Rsyncable codecs. A `compress`-class codec MAY be *rsyncable*: it periodically synchronizes (resets) its compression state so that a local change in the uncompressed input only affects a bounded neighborhood of the compressed output. This improves delta-transfer tools (e.g. `rsync`) and version-control delta compression (e.g. Git packfiles) at the cost of a small compression-ratio overhead. The only rsyncable codec defined in this revision is `zstd-rsyncable` (§8.5).

1.12.5 8.5 Canonical codec registry (v1)

Catalog entries are referenced by integer id within a file (§5), but each entry's `"name"` MUST be a canonical identifier from this registry so writers interoperate:

name	cls	baseline?	parameters
`identity`	`encode`	yes	none
`gzip`	`compress`	yes	`level`?
`zstd`	`compress`	yes	`level`?, `window`?, `dct`?
`zstd-rsyncable`	`compress`	no	`block_size`: uint (default 65536)
`lzma2`	`compress`	no	`level`?, `dct`?
`base64url`	`encode`	no	none (unpadded)
`base85`	`encode`	no	none
`cose-encrypt0`	`encrypt`	no	`COSE_Encrypt0` (1 recipient)
`cose-encrypt`	`encrypt`	no	`COSE_Encrypt` (n recipients)

A reader MUST match codecs by canonical `"name"`, not by catalog id (ids are file-local). Later spec versions register new codecs by canonical name; an unknown name degrades to an opaque node (§8.3).

1.13 9. Integrity and confidentiality

GTS keeps four integrity concerns distinct:

1. **Frame integrity** — the per-frame BLAKE3 self-hash `"id"` (§9.1).
2. **History integrity** — the `"prev"` content-id chain (§9.1).
3. **Origin / authorship** — optional COSE signatures (§9.2).
4. **Freshness / non-truncation** — a head commitment: a signature over the head `"id"`, or an index `"mmr"`/`"head"` root (§9.1, §13).

The first two are mandatory and key-free; the last two are optional and profile-driven.

1.13.1 9.1 Per-frame self-hash and content-id chain (mandatory)

Each frame's `"id"` is the BLAKE3-256 of its own content (every key except `"id"` and `"sig"`), so a frame is **content-addressed and independently verifiable**. Each frame's `"prev"` names the previous frame's

"id"; because "prev" is part of the hashed content, the chain is a git-style content-addressed list in which the **head id transitively commits to all history**.

- **Parallel verification.** Every "id" is a hash of a self-contained byte range; with the index "off" table (§6.2) all frame hashes are recomputed concurrently, followed by a trivial O(n) "prev"-equality pass. No accumulating dependency forces single-threaded reading. (The only inherently sequential step is discovering frame boundaries in a bare CBOR sequence — a cheap length-scan the index removes.)
- **Damage isolation and recovery.** A corrupt frame fails *its own* "id", so damage is **independently detectable**. Recovery of *subsequent* frames, however, is guaranteed only when their byte offsets are known — from an intact index "off" table, a checkpoint frame, external framing, or the storage layer. In a bare CBOR Sequence (no per-frame length) arbitrary byte corruption can desynchronise the decoder: a reader **with** offsets skips the bad frame and folds the survivors (reason: "damaged"), while a reader **without** offsets MAY be unable to resynchronise past the damage. evidence writers SHOULD emit periodic checkpoint indexes (§13) so recovery is robust.
- **Tamper-evidence.** Any insertion, reordering, or mutation breaks a "prev" link or a self- hash. **Truncation** (dropping trailing frames) is detected only against a head commitment — a signature over the head "id", the index "head"/"mmr" root (§6.2), or an out-of-band anchor. Opaque frames are part of the chain, so confidential frames cannot be stripped undetectably.

A **Merkle-Mountain-Range** (MMR) root over the frame ids (optional, carried in the index) is a single whole-file commitment that is itself parallel to compute and supports O(log n) inclusion proofs — proving a frame is in the log without shipping the log.

1.13.2 9.2 Signatures (optional, algorithm-agile)

A frame MAY carry "sig", a COSE_Sign1 (RFC 9052) over the frame's "id". Because "id" is the self-hash of the whole content — "pub", "d" (the ciphertext, if encrypted), and "prev" (the chain position) — one signature over "id" **binds** the public claims to the sealed payload and to the chain position, and signing the head "id" thereby anchors all prior history (§9.1). The signing algorithm is declared in the COSE header (e.g. EdDSA/Ed25519, ES256); readers MUST honour the declared algorithm. The evidence and opaque profiles (§13) REQUIRE signatures. Key discovery and trust anchoring (which keys are authentic, which signers are authorised) are **profile/deployment policy**, not core GTS: sigstat: "valid" means a signature is cryptographically valid under a *resolved* key, not that the key is trusted.

1.13.3 9.3 Encryption (optional)

An encrypt-class codec wraps the payload as COSE_Encrypt/COSE_Encrypt0. Recipients are listed in cleartext "to" by **key identifier only** — never the key material. Multiple recipients MAY share one sealed payload (each unwraps the content-encryption key with its own key). Key escrow, rotation, and revocation are the **issuer's** responsibility and are out of scope; a payload encrypted to a retired key MAY become permanently opaque.

This draft's v1 conformance surface implements and tests COSE_Encrypt0 for one direct recipient. Multi-recipient COSE_Encrypt envelopes and ECDH key-wrap are deferred outside v1 until dedicated vectors, key-management policy, and cross-engine interop tests exist; see GTS-SECURITY-POLICY.md.

The deferred cose-encrypt contract is pinned as a future Full Reader capability, not as a v1 implementation claim. A future implementation that claims it MUST consume a CBOR-tagged COSE_Encrypt envelope whose content encryption uses A256GCM and whose recipient array contains one entry per declared recipient in the frame's cleartext "to" list. The initially defined key-management mode is ECDH-ES+A256KW: each recipient entry carries the information needed to derive a key-encryption key and unwrap the same 256-bit content-encryption key with A256KW. A conforming future reader tries only recipients for which it holds the corresponding private key material.

Deferred failure modes are:

- no held key matches any recipient kid: emit MissingKey and preserve the frame as an opaque node with reason:"missing-key";

- ECDH recipient metadata is malformed, the derived key-encryption key cannot unwrap the content key, or AES-KW authentication fails: emit `KeyWrapFailed` and preserve the frame as opaque with `reason:"missing-key"`;
- content-authentication failure after a successful unwrap is a damaged encrypted payload and MUST NOT expose plaintext.

The descriptor vectors in `vectors/crypto-deferred/*.json` pin the two-recipient shape and the wrong-key, missing-key, and failed-unwrap opacity cases until byte-level `COSE_Encrypt` vectors replace the placeholders.

1.13.4 9.4 The opacity invariant (normative)

> Opacity hides **content** — never **existence**, **provenance**, or **position**.

For every frame, `{"id", "prev", "t", "x", "to", "pub", "sig"}` MUST remain in cleartext (the transform chain `"x"` is cleartext so a reader knows which codecs to reverse). A reader without the relevant key therefore still learns *that* the frame exists, *what kind* it is, *who* it is sealed for, *who* signed it, and *where* it sits in the chain. This is what makes selective disclosure safe: a holder can carry — and a verifier can authenticate the position of — data neither can read.

1.14 10. Compaction

Compaction folds a log and re-emits it as a single self-contained `snapshot` frame (re-interned dictionary, deduplicated quads, dropped self-loops, optionally a materialised entailment closure). A `snapshot`'s payload is a self-contained graph fold — terms, quads, reifiers, annotations, inline blobs, and meta:

```
snapshot-payload = {
  "terms"    : terms-payload,
  ? "quads"  : quads-payload,
  ? "reifies": reifies-payload,
  ? "annot"  : annot-payload,
  ? "blobs"  : { * digest => bstr },   ; inline content-addressed blobs
  ? "meta"   : any,
}
```

A reader folds a `snapshot` exactly as it would fold the equivalent sequence of `terms`/`quads`/`reifies`/`annot`/`blob` frames; term-ids restart at `0` within the snapshot's own dictionary. Compaction is **lossy by definition**: it discards the original per-frame signatures and the temporal stacking of the log. A compactor:

- MUST record the provenance of the fold (source log digest, time, agent) as quads in the snapshot, and
- SHOULD emit a fresh signature over the snapshot.

Two artifact classes follow: an **evidentiary log** (append-only, signed, never compacted) and a **distribution snapshot** (compacted, dense, lossy — ideal for shipping). A reader can tell which it holds from the profile and the presence of a `snapshot` frame.

1.14.1 10.1 Streamable compaction (ordering-only)

Streamable compaction converts an accretive segment (or multi-segment file) into one delivery-ordered segment in the streamable layout state (§3.3). Unlike snapshot compaction above, it is **a re-authoring of the ORDERING, and only the ordering**: the folded graph, the inline blobs, and every content-addressed fact are preserved. Three signature subjects behave differently under the rewrite, and a compactor MUST honour all three:

- **Content signatures** (subject = a content digest: a blob's BLAKE3, a statement or claim hash — "this is true, signed by Bob") are ordinary quads/annotations about digests. They are **compaction-invariant** and survive fully intact: nothing they attest to has changed.

- **Frame signatures** (a COSE_Sign1 over a frame `"id"`, which commits to `"prev"`, §9.2) become **detached, not broken**: they verify against the original frame id forever. A compactor MUST carry every source frame signature in **compaction provenance** — one `stream:DetachedSignature` node per signature, recording the original frame id (`stream:sourceFrame`) and the original COSE bytes (`stream:cose`), plus one `stream:sourceHead` per source segment head (§13.3) — so each remains a *checkable claim about the original log.*
- **Ordering commitments** (a signed head, an index `"mmr"` root) are the only layout-bound attestations. They cannot survive a reordering; the compactor re-issues the ordering commitment (the new trailing `index` with its `"head"`, §6.2) and thereby becomes the **sole attester of the new ordering**. A compactor MAY additionally COSE-sign the new head.

A compactor MUST record the rewrite itself as provenance quads in the output — a `stream:Compaction` node carrying the acting tool (`stream:agent`), the time (`stream:timestamp`), and the source segment heads (`stream:sourceHead`) — the §10 provenance MUST, given concrete vocabulary by §13.3.

Profiles demanding pristine third-party chain attestation. For an `evidence` segment the original signed chain *is* the artifact; a compactor MUST refuse it — unless it **seals the original log verbatim** as a nested GTS blob (§12.1) inside the streamable rewrite (role `"source"`, referenced from the provenance node via `stream:sealedSource`). The original bytes, chain, and signatures stay byte-intact and independently verifiable inside; the outer layout is delivery-ordered; one content digest binds them.

Refusals for publication tools (§14.1). A compactor MUST refuse: input that does not verify cleanly (any diagnostic); and input whose fold carries a frame-addressed suppression (`kind: "frame"`, §11) — the rewrite assigns new frame ids, so a frame-digest target would silently dangle. Digest-addressed `blob` suppressions are carried forward verbatim (content-addressing is layout-independent); id-addressed suppressions are carried forward value-wise (§11).

1.15 11. Suppression (additive "deletion")

GTS never physically deletes. To retract or hide prior content, a writer appends a `suppress` frame referencing the superseded subgraph or frame digest. The suppressed bytes remain present and hash-linked; suppression is a **display/precedence contract**, interpreted by the consumer, not an erasure. This preserves a complete, tamper-evident history.

```
suppress-payload = { "targets": [+ suppress-target], ? "reason": tstr, ? "by": term-id }
suppress-target =
    { "kind": "frame",   "id": digest } /                                  ; a frame, by its "id"
    { "kind": "blob",    "digest": digest } /                              ; a content-addressed blob
    { "kind": "term",    "id": term-id } /                                 ; a term + quads it appears in
    { "kind": "quad",    "q": [term-id, term-id, term-id, ? term-id] } /    ; one specific quad
    { "kind": "reifier", "id": term-id }                                   ; a reifier + its annotations
```

Suppression is **monotonic and additive**: a matched target is hidden from default resolution (a `term` target also hides every quad in which the term appears); the bytes remain present and hash-linked, and a consumer MAY surface suppressed content explicitly. There is no un-suppress in v1 — later frames may add further suppressions, and a later identical assertion does not revive a suppressed target.

Cross-segment suppression (normative, §3.1). Digest-addressed targets (`frame`, `blob`) are file-global: a content-id names the same bytes wherever they sit, so a later segment MAY suppress an earlier segment's frame or blob by digest. Id-addressed targets (`term`, `quad`, `reifier`) carry term-ids, which are segment-local — they are first **resolved to term values within the suppress frame's own segment**, and the suppression then applies **value-wise to the whole union fold**: a `quad` target hides every matching `(s,p,o,g)` value tuple in any segment, and a `term` target hides the term value (and the quads it appears in) file-wide. This is what lets an appended belief-revision segment suppress a statement made by an earlier segment without rewriting a byte of it — the earlier segment's record stays present, signed, and hash-linked (content-addressed at the wire level).

1.16 12. Binary and content-addressing

```
; a `blob` frame carries raw bytes in "d" (subject to "x"); its metadata lives in cleartext "pub":
blob-pub = { ? "mt": tstr, ? "rep": tstr, ? "digest": digest-ref }
; INLINE blob  -> "d" present; digest = BLAKE3(decoded "d").
; EXTERNAL blob -> "d" absent;  "pub".digest names bytes held elsewhere.
```

- A `blob` frame's bytes are addressed by their **BLAKE3-256 digest** — for an inline blob the `BLAKE3` of the decoded `"d"`, for an external blob `"pub".digest`; the graph references the blob by that digest. Identical bytes appearing twice are stored once by convention.
- A blob MAY be **inline** (bytes present, a self-contained package) or **external** (only the digest appears in the graph; bytes live elsewhere).
- A logical object MAY have **multiple representations** (`"rep"`/`"mt"` distinguishing, e.g., a master and a widely-supported fallback) — see content negotiation, §8.3.
- Transforming to a text format (§14) externalises inline blobs to a sidecar directory.

1.16.1 12.1 Nested GTS (recursive composition)

A blob whose media type is `application/vnd.blackcat.gts+cbor-seq` is itself a complete GTS file. Because a payload after transform reversal is opaque bytes, **any** frame payload MAY carry a nested GTS, wrapped in any transform chain — `[zstd]`, `[cose-encrypt]`, or both. The normative carrier is a `blob` whose `"pub".mt` is `application/vnd.blackcat.gts+cbor-seq`.

- **Fold semantics.** A Full Reader MAY recurse: decode the blob (subject to §6.1 capability rules), then fold the inner bytes as an independent GTS, exposing its result as a **subgraph** the parent graph references by the blob's digest. A Baseline Reader MAY treat a nested GTS as an ordinary blob (no recursion).
- **Blank-node scope.** The inner GTS has an independent blank-node scope. If a Full Reader exposes the inner fold beside the parent fold, it MUST relabel or scope inner blank nodes so labels cannot collide with the parent or with sibling nested GTS files.
- **Independent integrity.** The inner GTS has its own header, id/prev chain, and signatures. The **outer** chain proves the nested blob is present and intact at its position; the **inner** chain proves the nested log is intact. The two guarantees compose but do not depend on each other.
- **Composed opacity.** If the nested GTS is reached through an `encrypt`-class transform and the reader lacks the key, the *entire subgraph* — including its inner header — is an opaque node (§7.6): the holder can carry and prove the position of a whole sealed graph it cannot read. This is the matryoshka case ("a whole GTS inside an encrypted field").
- **Bounded recursion.** Readers MUST enforce a maximum nesting depth and total decoded-size budget (§18).

This composition needs no new frame type: nesting is "a blob that happens to be a GTS." The v1 Full Reader helper and negative security vectors for recursion limits are tracked in GTS-SECURITY-POLICY.md.

1.17 13. Profiles

A profile is a named set of conventions over the one format, declared in the segment header `"prof"` field. Profiles can define vocabulary expectations, validation rules, trust policy, capability requirements, and publication workflows, but they sit above the core wire format.

The status values used by this specification are:

- **core-required**: part of baseline wire-format, reader, or writer conformance.
- **optional-standard**: specified here as interoperable infrastructure, but not required for a Baseline Reader or Writer.
- **experimental**: described for early interoperability; details can change without changing core GTS.
- **domain-specific**: owned by an application or downstream community, not by core GTS.

profile or family	status	profile-level meaning	core impact
`generic`	core-required default	Any conformant log with no extra profile validation.	None; this is the absence of profile-specific requirements.
`dist`	optional-standard	A compacted distribution `snapshot`: vocabulary, definitions, and materialised closure.	None.
`evidence`	optional-standard	Append-only custody chain; profile validators require signatures and a head commitment.	None; signatures remain optional in core GTS.
`opaque`	optional-standard	Selective-disclosure convention over `encrypt`-class frames, signatures, and pseudonymous `kid`s.	None; encryption remains optional in core GTS.
`bundle`	optional-standard	A GTS whose `blob`s are themselves GTS files (`mt: application/vnd.blackcat.gts+cbor-seq`), using §12.1.	None.
`files`	optional-standard	A portable file-tree archive profile defined in §13.2 and §14.2.	None; baseline readers fold its graph normally.
`stream`	optional-standard	Streaming vocabulary and publication layout support used by §3.3 and §10.1.	None; layout checks are reader/tool diagnostics, not new frame grammar.
`image`	experimental	Blob representations plus descriptive metadata and analysis frames.	None.
`ai-package`	experimental	A concept plus logic, observations, opinions, refuted claims, embeddings, and data.	None.
`music-package`	domain-specific	GMEOW music transport conventions; informative here, specified by the downstream profile.	None.
GMEOW distribution profiles	domain-specific	Downstream GMEOW package conventions layered on GTS distribution artifacts.	None.
`agent-memory`	domain-specific	Application conventions for memory, belief revision, suppression, and provenance.	None.

Profiles constrain conventions, not the wire format; a `generic` reader reads all profile declarations it can parse. A reader that does not implement a named profile still parses, verifies, and folds the file under its

reader class, then reports the unsupported profile in diagnostics or metadata. In a multi-segment file each segment declares its own profile; the file's effective requirement set is the union (§3.1).

The `evidence` profile requires a head commitment (§9, item 4) at profile level, and writers SHOULD emit a checkpoint `index` at least every 1024 frames or 64 MiB, whichever comes first, so a damaged log recovers robustly (§9.1). That requirement does not make signatures, indexes, or evidence-profile support mandatory for baseline GTS.

Profile-policy configuration. A profile-aware verifier MAY accept a deployment trust policy beside the GTS bytes. The v1 policy document is JSON or YAML with these fields:

```
trusted_signers:
  - did:example:issuer
require_trusted_signer: true
pseudonymous_kid_pattern: "^anon:[0-9a-fA-F]{32,}$"
```

`trusted_signers` lists signer `kid` values authorized by the deployment. `require_trusted_signer` makes profiles that require signatures fail unless at least one valid signature is from a trusted signer. `pseudonymous_kid_pattern` controls the high-privacy recipient-id shape for the `opaque` profile. These settings are profile/deployment policy only: they do not change core GTS parsing, folding, frame ids, signature preimages, or baseline reader validity.

Third-party profile registration template. A third-party profile definition SHOULD publish:

- Stable profile name used in the header `"prof"` field.
- Owner, change-control process, contact URI, and specification URI.
- Status (`experimental`, `optional-standard`, or `domain-specific`) and intended compatibility policy.
- Vocabulary namespace IRIs, term shapes, and any profile-specific validation rules.
- Required codecs, keys, signature algorithms, trust anchors, or deployment assumptions.
- Failure taxonomy: which violations are errors, warnings, or informative diagnostics for a profile-aware tool.
- Interaction with segments, `cat` composition, suppression, compaction, and nested GTS blobs.
- Conformance vectors, including unsupported-profile behavior for baseline readers.
- Security and privacy considerations.

A profile definition MUST state that it does not change header/frame grammar, segment-boundary detection, content-id or signature/hash preimages, transform-catalog resolution, or the core fold semantics in §7. New profile behavior must be expressed as graph vocabulary, existing frame types, transform capabilities, metadata, or profile-aware validation rules.

The registry change policy, reserved namespaces, and optional-standard profile promotion process are maintained in GTS-GOVERNANCE.md.

1.17.1 13.1 Language-tag discipline (profile-level normative)

This subsection defines a profile/projection-writer rule, not a Baseline Reader requirement.

A producer's graph payload MAY carry **internal private-use language tags** (e.g. GMEOW's `x-gmeow-*`): the payload of a `dist` or `ai-package` segment *is* the canonical form, and canonical forms keep their internal tags. Every **projection section** — docs blobs, derived views, down-projected representations, anything generated *for an external consumer* — MUST carry **public BCP 47 tags only**; a producer that leaks private-use tags into a projection section MUST fail at write time, not warn (vector 20). The boundary is per *role*, not per file: one package legitimately carries a canonical payload with internal tags beside public-tagged docs sections. (This mirrors the GMEOW generator framework's internal-tag leak gate; the reference producer reuses its `retag` machinery at the section boundary.)

1.17.2 13.2 The `files` profile (optional-standard)

The `files` profile is an optional-standard, content-addressed archive of a file tree. It is the GTS answer to tar's `c/x/d`: pack a directory into a single-segment GTS, unpack it later, and `diff` it against a directory without byte comparison. The rules below are profile-level conformance requirements for `files` writers and validators; a Baseline Reader folds the graph without implementing archive tooling.

Namespace. The profile owns a small, spec-defined vocabulary at `https://w3id.org/gts/files#` (prefix `files`). GTS independence means an unpacker MUST NOT require GMEOW, schema.org, or any other ontology to read the archive; the vocabulary is authored in the spec and carried as literal IRIs in the graph.

term	IRI	shape
`FileEntry`	`https://w3id.org/gts/files#FileEntry`	One archived file.
`path`	`https://w3id.org/gts/files#path`	Relative path string, `/` separators, no leading `/`, no `..` components.
`digest`	`https://w3id.org/gts/files#digest`	`blake3:<hex>` content digest of the file bytes.
`size`	`https://w3id.org/gts/files#size`	Byte size as `xsd:integer`.
`mode`	`https://w3id.org/gts/files#mode`	POSIX permission bits as a decimal `xsd:integer` (for example, `420` for `0o644`). File-type bits are not recorded.
`modified`	`https://w3id.org/gts/files#modified`	Modification time as `xsd:dateTime` in UTC.
`mediaType`	`https://w3id.org/gts/files#mediaType`	Optional IANA media type string.

Quad shape. Each file in the archive is described by one blank-node `FileEntry`:

```
_:entry a files:FileEntry ;
    files:path "relative/path.txt" ;
    files:digest "blake3:<hex>" ;
    files:size 1234 ;
    files:mode 33204 ;
    files:modified "2026-06-10T20:00:00Z"^^xsd:dateTime ;
    files:mediaType "text/plain" .
```

Determinism. A `files` archive MUST be byte-reproducible for the same input tree:

- Paths are sorted lexicographically by their UTF-8 byte sequence before emission.
- Stored paths use `/` separators and MUST be non-empty relative paths. Writers, unpackers, and diff tools MUST refuse absolute paths, Windows drive-relative paths, `..` components, `.` components, empty components, and backslash separators before touching file bytes.
- Modification times are normalised to UTC and serialised as `xsd:dateTime` with second precision. Fractional seconds MUST be truncated before emission.
- Only POSIX mode and mtime are recorded; ownership, uid/gid, xattrs, and ACLs are deliberately excluded — they are tar's portability tarpit.
- Symlinks are deliberately excluded. `pack` and `diff` MUST refuse symlink entries rather than following them; `unpack` MUST also refuse any destination escape through an existing symlink below the output directory.

Inline and external blobs. A file's bytes MAY be carried as an inline `blob` frame (`"d"` present, digest = BLAKE3(decoded `"d"`)) or as an external blob (`"d"` absent, `pub.digest` names bytes held elsewhere, §12). Identical bytes appearing under multiple paths are stored once by convention. The standard `gts pack` command emits inline blobs only. Implementations MAY add an explicit external-blob mode, but it

MUST be opt-in and documented. `gts unpack` MUST refuse an unsuppressed `FileEntry` whose inline blob is absent; `gts diff` MAY compare by `files:digest` without fetching external bytes.

Suppression. A blob-targeted suppression (§11) hides matching file bytes from default extraction. `gts unpack` and `gts extract` skip/refuse suppressed blobs by default and expose an explicit `--include-suppressed` override when the operator intentionally wants retained history.

Relationship to other vocabularies. The profile is deliberately self-contained, but the terms align by reference to common surface vocabularies: `files:size` schema.org `contentSize`, `files:mediaType` schema.org `encodingFormat`, `files:modified` NFO `fileLastModified`, `files:path` NFO `fileName`. These alignments live in GMEOW's mapping DSL; the files profile itself does not depend on them.

1.17.3 13.3 The `stream` vocabulary (optional-standard)

The streamable layout state (§3.3) and streamable compaction (§10.1) use a small, optional-standard vocabulary at `https://w3id.org/gts/stream#` (prefix `stream`) — the same independence decision as the `files` profile (§13.2): no GMEOW or external ontology is required to stream a photo archive; the terms are authored here and carried as literal IRIs in the graph. The vocabulary is deliberately distinct from `files#` (the two compose: a `files` archive that is also streamable describes each file once as a `files:FileEntry` and once as a `stream:Manifestation` — the profile check (§14.1) and the layout check (§3.3) stay independent).

Streaming-index terms — one `stream:Manifestation` per promised blob, emitted in the leading streaming index before any `blob` frame (§3.3):

term	IRI	shape
`Manifestation`	`https://w3id.org/gts/stream#Manifestation`	Class. One blob this segment promises to deliver.
`digest`	`https://w3id.org/gts/stream#digest`	`blake3:<hex>` content digest — the IOU the blob redeems.
`mediaType`	`https://w3id.org/gts/stream#mediaType`	Declared IANA media type (mirrors the blob's `pub.mt`).
`size`	`https://w3id.org/gts/stream#size`	Byte size of the decoded blob as `xsd:integer`.
`role`	`https://w3id.org/gts/stream#role`	Delivery role string: `"preview"` / `"primary"` / `"source"`; open set.
`order`	`https://w3id.org/gts/stream#order`	Intended delivery position among the segment's blobs, `xsd:integer`, 0-based.

Compaction-provenance terms — the concrete vocabulary for §10/§10.1's provenance MUST:

term	IRI	shape
`Compaction`	`https://w3id.org/gts/stream#Compaction`	Class. One rewrite event (a blank node).
`agent`	`https://w3id.org/gts/stream#agent`	The acting tool, a string (e.g. `"gts-compact"`).
`timestamp`	`https://w3id.org/gts/stream#timestamp`	Rewrite time as `xsd:dateTime` in UTC.
`sourceHead`	`https://w3id.org/gts/stream#sourceHead`	`blake3:<hex>` head id of one source segment; repeated per segment.
`sealedSource`	`https://w3id.org/gts/stream#sealedSource`	`blake3:<hex>` digest of the nested-GTS blob holding the verbatim original (§10.1).

term	IRI	shape
`DetachedSignature`	`https://w3id.org/gts/stream#DetachedSignature`	Class. One carried-over frame signature (a blank node).
`sourceFrame`	`https://w3id.org/gts/stream#sourceFrame`	`blake3:<hex>` original frame `"id"` the COSE signature verifies against, forever.
`cose`	`https://w3id.org/gts/stream#cose`	The original COSE_Sign1 bytes, base64url (unpadded) literal.

Quad shape (a compacted segment's streaming index, then provenance):

```
_:m0 a stream:Manifestation ;
    stream:digest "blake3:<hex>" ;
    stream:mediaType "image/webp" ;
    stream:size 20480 ;
    stream:role "primary" ;
    stream:order 0 .
_:c a stream:Compaction ;
    stream:agent "gts-compact" ;
    stream:timestamp "2026-01-01T00:00:00Z"^^xsd:dateTime ;
    stream:sourceHead "blake3:<hex>" .
_:s0 a stream:DetachedSignature ;
    stream:sourceFrame "blake3:<hex>" ;
    stream:cose "<base64url>" .
```

Claim coupling (normative). Use of `stream#` terms in a segment that does NOT claim `"layout": "streamable"` is a **warning**, not an error (§14.1): provenance quads legitimately survive `gts → nq → gts` round trips and re-accretion after appends. The error class is reserved for the opposite rot — a claimed layout the bytes contradict (§3.3).

1.17.4 13.4 Domain profile example: `music-package` (informative)

This subsection is an informative example of a domain-specific profile. A Baseline Reader, Writer, or verifier is not required to implement GMEOW vocabulary, music-domain rules, notation projection rules, or the `music-package` validator to be conformant with core GTS.

The `music-package` profile can be defined as a single-segment GTS that carries frame-relative musical content: a `MusicalWork`/`MusicalExpression`, its `Voice`s and `MusicalSegment`s, `TuningSystem` and `MusicalTimeFrame` reference frames, atomic `ToneEvent`s, `DegreeOfFreedom` declarations, and standpoint-indexed analysis claims. It is the canonical transport form for the GMEOW music slice and the input to notation projections.

Namespace. The profile reuses the GMEOW music vocabulary (`https://blackcatinformatics.ca/gmeow/`). A `music-package` is not required to be a `dist` profile: it may carry only the musical content graph plus any projection blobs, and it MAY rely on an external `dist` snapshot for vocabulary definitions.

Header. A `music-package` segment declares `"prof": "music-package"`. The profile can be append-only for new claims; existing triples are not deleted, only superseded by statement-layer provenance (§7.3).

Example quad shape. A minimal package can contain:

```
@prefix gmeow: <https://blackcatinformatics.ca/gmeow/> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .

:piece a gmeow:MusicalExpression ;
    gmeow:hasVoice :voice1 .
```

```
:voice1 a gmeow:Voice ;
    gmeow:voiceTuningFrame :tuning12EDO ;
    gmeow:voiceTimeFrame :timeGrid .

:tuning12EDO a gmeow:TuningSystem .
:timeGrid a gmeow:MusicalTimeFrame .

:event1 a gmeow:ToneEvent ;
    gmeow:segmentOf :voice1 ;
    gmeow:toneEventPitchValue :pitchC4 ;
    gmeow:segmentSpan :span1 .

:span1 a gmeow:MusicalTimeSpan ;
    gmeow:hasMusicalTimeFrame :timeGrid ;
    gmeow:timeStartNumerator 0 ;
    gmeow:timeStartDenominator 1 ;
    gmeow:timeDurationNumerator 1 ;
    gmeow:timeDurationDenominator 4 .
```

Time and pitch are **frame-relative**: `toneEventPitchValue` points to a `PitchValue` interpreted under the event's voice tuning frame, and offsets/durations are rational values interpreted under the voice time frame.

Projections. A `music-package` can contain `blob` frames whose bytes are down-projected representations (MusicXML, MEI, ABC, LilyPond, Humdrum **kern, MIDI, Scala `.scl`, tablature, mensural, graphic notation). A music-package profile validator can require each projection to be accompanied by a declared-loss manifest that lists the `NotationProjectionProfile` used, the `MusicalParameter`s it can represent, and the `ProjectionLoss`es it incurs. The manifest can be a Turtle sidecar or an embedded header/comment and is considered part of the projection, not the canonical graph.

Bundle profile coupling. A `bundle` profile (§12.1) whose blobs are `music-package` segments provides the multi-movement / multi-version transport case. Each nested segment keeps its own profile declaration; the outer bundle does not impose additional conventions.

Verification. A music-package-aware verifier can check that every `NotationSystem` referenced by a projection blob has a corresponding `NotationProjectionProfile`, and that the profile accounts for every `MusicalParameter` declared in the music slice (no silent omissions). Baseline `gts verify` is not required to implement this profile; it can report the unsupported profile without failing wire-format validity.

1.18 14. Transforms out

Transforms convert GTS to operating substrates. Each is a thin shim over the folded tables — no RDF text parser is involved.

- `gts → nquads` / `gts → turtle` — serialise `quads` + `reifies`/`annot` (the latter as RDF 1.2 reification). Inline blobs are **externalised** to `./blobs/<blake3>.bin`, and the graph's digest references resolve to those paths. Opaque frames serialise as their opaque-node descriptions.
- `gts → duckdb` / `gts → sqlite` — bulk-load the four tables (`terms`, `quads`, `reifies`, `annot`) plus a `blobs` table; create the indexes appropriate to the engine. This is a near-mechanical load because the GTS tables already match the relational shape.

Each transform SHOULD be verifiable by **round-trip equivalence**: for **fully-decodable** frames, `gts → nq → gts` MUST yield the same folded graph (modulo blank-node labelling and deterministic CBOR re-encoding). Opaque nodes are excluded — they serialise as opaque-node descriptions and re-import as ordinary quads, not as opaque frames.

1.18.1 14.1 Composition tooling requirements (normative for conformant tools)

This section defines tool conformance only. A Baseline Reader or Writer need not ship these CLI verbs, transform targets, archive commands, or publication policies to be core-conformant. Profile-aware tools enforce only the profile validators they claim to support; unsupported profiles are surfaced as diagnostics or metadata unless the user explicitly requested that profile's validation.

Raw `cat` always works (§3.1); a conformant **validating composer** (`gts cat`) and verifier (`gts verify`) add the refuse-don't-trust posture:

- **`gts cat` MUST refuse degenerate inputs**: an input that is not a valid GTS, a segment whose fold yields zero quads and zero blobs (almost always a wiring bug, never a real package), or an output in which a suppress-only segment would hide every prior frame. Publish-class tools never trust a pathological state to be intentional.
- **`gts verify` MUST check declared-vs-computed requirements for supported profiles**: a segment whose graph uses a supported profile's vocabulary without declaring the profile is an **error**; a declared-but-unused supported profile is a warning. Declarations a tool reads (the CLI dependency report, §13) must not be able to rot against the content they describe.
- **`gts verify` SHOULD report per-segment**: head id, signer set, profile, term/quad counts, opaque-node count with reasons — the composition ledger of the file.
- **`gts verify` MUST check the layout claim** (§3.3): a segment claiming `"layout": "streamable"` whose covered region violates delivery ordering, or whose index footer is missing or contradicts the frames it covers, is an **error** (`StreamableLayoutError`, §2.3); `stream#` vocabulary in an unclaimed segment is a **warning** (§13.3). `gts info` and `gts verify` SHOULD report the streamable boundary of a claimed segment — "streamable through frame N, accretive tail of M frame(s)".
- **`gts compact --streamable <in> -o <out>` is the layout rewrite** (§10.1). It MUST refuse input that does not verify cleanly, input carrying frame-addressed suppressions, and `evidence` input without the seal-the-original option (`--seal-original`, §10.1); it MUST emit a single claimed segment in the normative streamable shape (§3.3) with compaction provenance and detached signatures (§13.3), and its output MUST be byte-deterministic for the same input and parameters (blobs ordered by ascending decoded size, ties broken by ascending digest; the rewrite timestamp is a parameter, not ambient time).
- **Deterministic graph authoring mode** is the reproducible-build writer surface for a folded graph. It emits one ordinary segment and MUST remap local term ids before writing: terms are sorted by semantic value (IRI string; literal lexical form plus effective datatype IRI plus language tag; blank-node label, with anonymous blank nodes using their input occurrence as a tiebreaker; quoted triple resolved to its subject/predicate/object value). It then emits authorable frames in this fixed order: `terms`, `quads`, `reifies`, `annot`, `blob`, `meta`, `suppress`. Quads, reifier bindings, annotations, blobs, metadata keys, and suppression frames are sorted by the remapped deterministic-CBOR representation. The mode does not replay reader observations (`opaque`, signatures, diagnostics, or segment ledgers); publication tools that need to preserve those observations must use a profile-specific rewrite such as streamable compaction or seal the original bytes as evidence.
- **Blob extraction is verification, never conversion** (`gts ls`, `gts extract`): blobs are addressed by content digest (frame indices are physical accidents that shift under `cat`); extraction re-hashes the bytes against the requested digest; a blob suppressed by digest (§11) is refused by default (suppression is a display contract and extraction is display) with an explicit override; a media-type flag is an **assertion** against the blob's declared `pub.mt` — a validating publication tool refuses a mismatch rather than transcoding.

1.18.2 14.2 Archive tooling (`files` profile)

The `files` profile adds three validating publication commands. They share the refuse-don't-trust posture of §14.1: raw byte operations are always valid GTS, but a tool refuses pathological states rather than trusting them to be intentional.

- **`gts pack <dir|file>... -o out.gts`** Produce a single-segment GTS whose header declares `"prof": "files"`. Each argument is archived: a file is added under its basename; a directory is added recursively.

The resulting archive contains, in order, the `terms` and `quads` describing every `files:FileEntry`, followed by the inline `blob` frames for the file contents. The command MUST refuse:
  - inputs that contain unsafe stored paths: absolute paths, drive-relative paths, `..`, `.`, empty components, or backslash separators;
  - symlinks;
  - inputs that are not readable or that disappear during the walk.
- **`gts unpack <archive> [-C dir]`** Write every `files:FileEntry` in the archive to the destination directory (default current working directory). The command MUST:
  - refuse to write outside the destination directory (`..`, absolute paths, or symlinks that escape it);
  - re-hash each written file and verify it matches `files:digest`;
  - restore the file's declared modification time and permissions (subject to the host OS);
  - skip entries whose digest is suppressed (§11) by default, with an explicit `--include-suppressed` override.
- **`gts diff <archive> <dir>`** Compare the archive's `files:FileEntry` set to the current state of `<dir>` by content digest. Report added, removed, and modified paths. Exit `0` if the directory matches the archive exactly; exit `1` if any path differs or if the input is refused. No byte comparison is needed: content addressing makes the operation O(read) on the directory.

Archive workflow comparison.

workflow	usual table of contents	GTS `files` profile behavior
`tar`	Header records are interleaved with file bytes; path and metadata interpretation is tool policy.	The manifest is RDF quads, so path, digest, size, mode, mtime, and media type are queryable before or beside blobs.
`zip`	Central directory enables random access but is a rewrite-oriented footer.	GTS stays append-only; optional indexes accelerate access without making the footer the archive identity.
BagIt-style package	Payload files plus sidecar manifests/checksums.	The graph-native manifest and content bytes travel in one verifiable CBOR Sequence; external blobs remain content-addressed when used.

The value proposition is not compression ratio. Use a compression transform or an outer transport when size dominates. The `files` profile is for graph-native manifests, digest-addressed deduplication, append composition, and consistent safety policy across engines.

1.19 15. Worked examples

CBOR is shown in **diagnostic notation** (RFC 8949 §8). Hashes/signatures are elided as `h'…'`.

1.19.1 15.1 Minimal distribution snapshot (`dist`)

```
55799(                                  / self-describe magic /
  { "gts": "GTS1", "v": 1, "prof": "dist",
    "cat": { 0: {"name":"identity","cls":"encode"},
             4: {"name":"zstd","cls":"compress"} },
    "id": h'…header.id…' }
)
{ "t": "terms", "prev": h'…header.id…', "id": h'…terms.id…',
  "d": [ {"k":0,"v":"https://example.org/Cat"},          / id 0 /
         {"k":0,"v":"http://www.w3.org/2000/01/rdf-schema#label"},  / id 1 /
```

```
        {"k":1,"v":"Cat","l":"en"} ] }                    / id 2 /
{ "t": "quads", "prev": h'…terms.id…', "id": h'…', "x": [4],
  "d": h'…zstd([[0,1,2]])…' }                             / Cat rdfs:label "Cat"@en /
```

Term 2 is a literal with a language tag and no `"dt"`, so its datatype is `rdf:langString` (§7.1).

1.19.2 15.2 Evidence: image + signed accrual (`evidence`)

```
{ "t": "blob", "prev": h'…header.id…', "id": h'…',
  "pub": {"mt":"image/jp2"}, "d": h'…image bytes…',      / digest = blake3(d) /
  "sig": h'COSE_Sign1 by did:photographer' }
{ "t": "annot", "prev": h'…blob.id…', "id": h'…',
  "d": [[10,11,12]],                                     / reifier 10: capturedAt … /
  "sig": h'COSE_Sign1 by did:photographer' }
{ "t": "annot", "prev": h'…prev.id…', "id": h'…',        / later custody transfer, separate signer /
  "pub": {"event":"custody-transfer"},
  "d": [[13,11,14]], "sig": h'COSE_Sign1 by did:evidence-clerk' }
```

Nothing is rewritten; every accrual is hash-linked and independently signed.

1.19.3 15.3 Notary: partially-opaque frame (`opaque`)

```
{ "t": "annot", "prev": h'…prev.id…', "id": h'…',
  "pub": { "claim": "I hereby notarized this document.",
           "notary": "did:notary:jane", "ts": "2026-06-09T12:00:00Z" },
  "x": [4, 7],                                            / 7 = cose-encrypt /
  "to": [ {"kid":"anon:7f3a…","alg":"ECDH-ES+A256KW"} ],  / pseudonymous kid (opaque profile, §18) /
  "d": h'COSE_Encrypt(verified ID record + provenance)',
  "sig": h'COSE_Sign1 by did:notary:jane' }
```

Anyone verifies the public notarization and its signature; only the court key decrypts the sealed record; the signature binds the two (§9.2). A reader without the court key folds this to an opaque node with `reason:"missing-key"`, `pub` intact, `sigstat:"valid"`.

1.19.4 15.4 Graceful degradation (`image`, content negotiation)

```
{ "t": "blob", "prev": h'…', "id": h'…', "pub": {"mt":"image/vnd.djvu","rep":"master"}, "x":[9], "d": h
{ "t": "blob", "prev": h'…', "id": h'…', "pub": {"mt":"image/jpeg","rep":"fallback"}, "d": h'…' }
```

A reader lacking codec 9 (djvu) folds the master to an opaque node and uses the JPEG fallback — both are present, both are hash-linked.

1.19.5 15.5 Matryoshka: a whole signed GTS sealed inside a frame (`bundle` / `opaque`)

```
{ "t": "blob", "prev": h'…', "id": h'…',
  "pub": { "rep": "sealed-evidence-graph", "mt": "application/vnd.blackcat.gts+cbor-seq" },
  "x": [4, 7],                                            / zstd then cose-encrypt /
  "to": [ {"kid":"did:court:registry"} ],
  "d": h'COSE_Encrypt( zstd( <a complete, independently-signed GTS file> ) )' }
```

Without the court key this folds to one opaque node — a whole subgraph the holder carries but cannot read, yet whose presence and position the outer chain proves. With the key, a Full Reader recurses (§12.1) and folds the inner GTS — header, chain, signatures and all — into a verifiable subgraph.

1.20 16. Media type and HTTP serving contract

GTS files are published artifacts. This section defines deployment conformance: a locally stored GTS file can be wire-format valid, reader-conformant, and writer-conformant even when it is never served over HTTP.

A conformant deployment MUST advertise the media type, support range requests, and set cache headers that respect the format's immutability.

1.20.1 16.1 Media type and file extension (normative)

- **Media type:** `application/vnd.blackcat.gts+cbor-seq` (registration template in §20.1). GTS uses the `+cbor-seq` structured-syntax suffix because a GTS file is a CBOR Sequence ([RFC 8742]) of segment headers and frames, not a single CBOR data item. The earlier provisional `application/vnd.blackcat.gts+cbor` spelling is obsolete; deployments MUST emit `application/vnd.blackcat.gts+cbor-seq`. Readers MAY accept the obsolete spelling as a legacy alias, but MUST NOT emit it in newly written metadata.
- **File extension:** `.gts`.
- **Magic bytes:** the CBOR self-describe tag `55799` (`0xd9 0xd9 0xf7`) at the start of the first segment's Header when the first segment is tagged. A reader MAY use these three bytes as one signal while identifying a candidate GTS file, but MUST confirm the Header shape before treating the bytes as GTS.

Servers that do not recognise `application/vnd.blackcat.gts+cbor-seq` SHOULD fall back to `application/octet-stream` rather than a wrong text type; clients SHOULD inspect the first CBOR data item when the media type is missing or generic.

1.20.2 16.2 File identification algorithm (normative)

Media type metadata is authoritative when it is available. When a reader must identify bytes without trusted metadata, it MUST use this algorithm:

1. Treat `.gts` and `application/octet-stream` as hints only; neither proves nor disproves GTS.
2. If the first three bytes are `0xd9 0xd9 0xf7`, parse the first CBOR item as a tagged item and unwrap tag `55799`. Otherwise parse the first CBOR item from byte offset `0`.
3. The unwrapped first item MUST be a Header map containing `"gts": "GTS1"` and lacking frame key `"t"`. A mismatch is not a GTS file.
4. A positive identification is still only an identification result. Complete validity requires parsing the whole observed byte stream as a CBOR Sequence (§3), applying segment-boundary rules (§3.1), and validating ids, chains, profiles, and capabilities as required by the selected conformance class.
5. Implementations MUST NOT require a whole-file CBOR wrapper, a total item count, or a length prefix. Independently valid tagged segments may be concatenated, so later `55799` tags identify later segment headers, not nested whole-file objects.

1.20.3 16.3 HTTP serving semantics (normative)

A GTS package is served like any other immutable binary release, with three extra requirements:

1. `Accept-Ranges: bytes` MUST be sent for every `.gts` response. The format is designed for partial, streaming consumption (§3.2): a consumer can fold the header and a prefix of frames without downloading the whole file. Clients choose byte ranges from discovered CBOR item offsets, indexes, or other trusted manifests; HTTP range support does not by itself validate or repair local file bytes.
2. **No transforms at the edge.** Because the bytes are a content-addressed chain, proxies and servers MUST NOT apply compression, minification, or any byte-altering transform. The frames are already compressed by the writer's chosen codec; re-compressing at the transport layer breaks content hashes and signatures.
3. **CORS.** A public vocabulary/dataset package is expected to be cross-origin readable. Responses SHOULD include `Access-Control-Allow-Origin: *` for the served `.gts` origin.

1.20.4 16.4 Immutability-aware caching (normative)

Published GTS releases are immutable; a GTS package URL names one exact byte sequence.

- **Versioned URLs** (…/gmeow/1.2.3/gmeow.gts, …/packages/music/2026-06-18/music.gts, or any URL that contains a version/date/head identifier) MUST be served with:

  ```
  Cache-Control: public, max-age=31536000, immutable
  ETag: "<last-segment-head>"
  ```

  The natural ETag is the hex of the file's last segment head id (§3.1), because it transitively commits to every byte of the file. The `immutable` directive tells caches they need not revalidate for the one-year lifetime.

- **`latest` / conneg aliases** (URLs that resolve to the current release and may change) MUST NOT be cached as a single variant:

  ```
  Cache-Control: private, no-store
  Vary: Accept
  ```

  The `Vary: Accept` prevents conneg-cache poisoning when the same path negotiates to HTML, Turtle, or the GTS package. This is the same cache-poisoning class addressed for slice IRIs by the Apache generator.

Profile selection remains URL-shaped in v0.2: one URL per package. RFC 6906 / `Accept-Profile` is noted as a possible future extension, not required for v0.2 conformance.

1.21 17. Versioning and durability guarantees

- The header `"v"` is the spec major version. A reader MUST refuse a major version it does not implement, but MUST still verify the id/prev chain and enumerate frame types/ids.
- **Segment semantics and older readers.** A reader implementing this revision MUST support segment boundaries (§3.1). A reader that does NOT (a pre-§3.1 implementation) encounters a second Header as a non-frame data item: such input is **malformed for that reader**, and it MUST surface a fatal diagnostic for the remainder of the file rather than skip the item — *silently misfolding (applying file-global term-ids across a boundary) is the one forbidden outcome* (vector 17). Because `cat` cannot rewrite the first segment's header (the self-hash seals it), multi-segment files cannot advertise themselves in the first header; boundary detection is therefore structural, and the hard-fail rule is what protects the ecosystem's oldest readers.
- **Structure durability:** a GTS file plus this specification is decodable forever with no engine and no external dictionary — CBOR is an IETF standard and dictionaries are in-band.
- **Density durability:** governed by the codec catalog; the mandatory core set (`identity`/`gzip`/`zstd`) guarantees a baseline that any era can decode.

1.22 18. Security considerations

- The id/prev chain provides integrity, **not** confidentiality; use `encrypt`-class codecs for confidentiality.
- **Truncation** (dropping trailing frames) is undetectable from the chain alone; an `evidence` artifact MUST anchor the head — a signature over the head `"id"`, or the index `"head"`/`"mmr"` root (§6.2) — so a verifier can detect a shortened log.
- **Recovery** of frames *after* a damaged one is guaranteed only with known offsets (an intact index, a checkpoint frame, or external framing); a bare CBOR Sequence can desynchronise on arbitrary corruption (§9.1). GTS defines no parity/erasure coding — durability against bulk loss is the storage layer's concern.
- `"to"`/`kid` values can leak relationship metadata (who a frame is sealed for). The `opaque` profile therefore REQUIRES pseudonymous `kid`s; other high-privacy profiles SHOULD use them. Use a per-document or pairwise identifier — e.g. `"kid": "anon:<BLAKE3(true-kid  head-id)>"` — or key blinding, so the same recipient is unlinkable across files.
- A valid signature attests the signer over the frame's bytes; it does **not** assert the truth of the claims (consistent with attestation semantics — vouching correctness).
- Opaque frames are unreadable but **not** invisible; do not place secrets in `"pub"`, `"to"`, or `"meta"`.

- Snapshot compaction (§10) destroys original signatures; an `evidence` artifact MUST NOT be snapshot-compacted. Streamable compaction (§10.1) detaches frame signatures rather than destroying them, but the re-ordered chain is attested only by the compactor; an `evidence` artifact MUST NOT be streamable-compacted except by sealing the original verbatim (§10.1), and a consumer's trust in the *ordering* of a compacted file is trust in the compactor.
- Decompression of attacker-supplied frames MUST be bounded (zip-bomb resistance); readers SHOULD cap decoded sizes.
- Nested GTS (§12.1) MUST be bounded: readers MUST enforce a maximum recursion depth and a total decoded-size budget across all nesting levels (matryoshka-bomb resistance).
- **Segments are independently authentic, not mutually vouched.** Concatenation implies no endorsement: segment A's signer attests nothing about segment B. A verifier MUST report signer sets per segment (§14.1), and a consumer deciding trust MUST NOT treat the file-level union as carrying the strongest segment's authority. Value-wise cross-segment suppression (§11) means an untrusted appended segment can HIDE earlier content from default resolution — readers SHOULD surface which segment suppressed what, and high-assurance consumers MAY resolve suppression only from segments whose signers they trust.
- A torn append at a segment boundary looks like a torn header: the §3 torn-append rule applies; the prior segments fold intact.

1.23 19. Conformance test vectors

A conformant implementation MUST pass a shared corpus. v1 requires at least these vectors (shipped with the reference implementation), each as the GTS bytes plus the expected folded graph (N-Quads) and the expected diagnostics:

The companion GTS-CONFORMANCE.md defines the tiered conformance claims, named vector subsets, expected JSON fields, vector manifest schema, diagnostics registry, and read/verify modes used to turn this corpus into comparable implementation claims.

1. Minimal valid file (header + one `terms` + one `quads`).
2. A `zstd`-transformed `quads` frame.
3. An unknown-codec frame → opaque `reason:"unknown-codec"`.
4. A frame with a wrong self-`"id"` → `DamagedFrame` opaque.
5. A torn append at EOF → `TornAppendError`, survivors intact.
6. Header self-hash verification (positive and tampered).
7. RDF 1.2 reifier + `annot` round-trip (`gts → nq → gts`), including quotation-without-assertion.
8. A nested GTS blob (`mt: application/vnd.blackcat.gts+cbor-seq`), recursed and folded.
9. Suppression over a term-id and over a frame digest.
10. Truncation detection against a signed head / index `"mmr"` root.
11. Literal datatype defaulting (§7.1): a literal with `"l"` and no `"dt"` → `rdf:langString`; with neither → `xsd:string`.
12. A reifier rebound to a different triple → `ConflictingReifier`, first binding kept (§7.8).
13. A position-constraint violation, e.g. a literal in predicate position → rejected/diagnosed (§7.4).
14. Blank-node label locality (§7.1, §12.1): identical bnode labels in an outer and a nested GTS remain isolated (not merged).
15. **Two-segment union (§3.1)**: `cat` of two single-segment files folds to the value-union of both graphs; term-ids resolve segment-locally (a shared IRI unifies; identical ids naming different values do NOT collide); identical bnode labels across segments stay isolated. *15b*: label-less blank nodes (absent **or empty** `"v"`) are distinct terms within a segment and across segments, and the union's serialized labels MUST keep them distinct — relabeling that merges what the graph separates is the forbidden outcome.
16. **Composed round-trip (§3.1, §14)**: a `cat`-composed file survives `gts → nq → gts` with the same union fold.
17. **Pre-segment reader hard-fail (§17, negative)**: an implementation in pre-§3.1 mode fed a two-segment file MUST surface a fatal diagnostic at the second header — folding frames past the boundary with file-global term-ids is the forbidden outcome this vector exists to catch.

18. **Cross-segment suppression (§11)**: a second segment suppresses (a) an earlier segment's frame by digest and (b) a quad by value; default resolution hides both; the suppressed segment's bytes verify intact; the verifier reports which segment suppressed what (§18).
19. **Profile union + graceful segment opacity (§3.1)**: a two-segment file whose second segment requires an undeclared-to-the-reader capability folds segment one fully and segment two as opaque nodes with the profile named in the diagnostics.
20. **Language-tag discipline (§13.1, negative)**: a producer emitting a private-use language tag into a projection/docs section MUST fail at write time; the same tag in a canonical `dist` payload section is accepted.
21. **Degenerate composition refused (§14.1, negative)**: `gts cat` refuses an empty-fold segment and a suppress-everything composition; raw byte `cat` of the same inputs still yields a structurally valid file (the tool is stricter than the format, by design).
22. **Inline blob (§12, §14.1)**: an inline blob folds to its `blake3:<hex>` digest with declared metadata (`pub.mt`) retained; extraction by digest re-verifies the bytes; a digest-suppressed blob is refused by default.
23. **Prefix-fold streaming property (§3.2, derived)**: not a vector but a property test over EVERY vector in this corpus — each item-boundary prefix folds without error, and across growing prefixes the folded tables only ever extend (terms/quads are list-prefixes while the segment count is unchanged; ground (blank-node-free) N-Quads lines are monotone across the single→multi-segment representation switch).
24. **Streamable compaction (§3.3, §10.1, §13.3)**: an accretive source (blobs interleaved before their catalog, one COSE-signed frame, no claim) and its compacted rewrite — the rewrite claims `"layout": "streamable"`, leads with the streaming index, orders blobs most-significant-first, closes with the offset `index` footer, and carries compaction provenance including the detached source signature; both files fold to the same content graph; the compacted bytes are **frozen** and double as the cross-engine determinism oracle (same input + same timestamp parameter byte-identical output in every engine).
25. **Streamable claim that lies (§3.3, negative)**: a segment claiming `"layout": "streamable"` that delivers a covered blob before the quads describing its digest → `StreamableLayoutError`; a verifying tool MUST refuse (exit non-zero).
26. **Appended-after-compaction boundary (§3.3)**: a compacted segment with frames appended after its `index` footer folds cleanly with no diagnostic, and tooling reports "streamable through frame *N*, accretive tail" — the unpresaged tail is legal.
27. **Total-reader hostility regressions (§2.4, negative)**: empty input, a first CBOR item that is not a Header, an unsupported header major version, an unknown structural frame type, a forward term reference, and a malformed transform payload all return structured diagnostics/opaque nodes where applicable. These vectors pin the "never panic on input bytes" invariant and make cross-engine diagnostic drift visible in CI.
28. **Deterministic graph writer (§14.1)**: two equivalent folded graph states with different local term ids and row order produce byte-identical GTS via deterministic graph authoring. The frozen vector pins term remapping, row sorting, blob metadata retention, metadata output, and suppression target remapping across Python and Rust producers.

1.24 20. IANA considerations

This section registers one media type. It follows the registration procedures of [RFC 6838] and the structured-syntax-suffix procedures of [RFC 9277]. Pending formal registration, the type lives in the vendor (`vnd.`) tree and is used provisionally.

1.24.1 20.1 Media type registration: `application/vnd.blackcat.gts+cbor-seq`

- **Type name:** `application`
- **Subtype name:** `vnd.blackcat.gts+cbor-seq`
- **Required parameters:** none
- **Optional parameters:** none

- **Encoding considerations:** binary. A GTS file is a CBOR Sequence ([RFC 8742]) and is not restricted to 7-bit or 8-bit text; transports that are not 8-bit clean MUST apply a content-transfer-encoding (e.g. base64).
- **Security considerations:** see §18 of this specification. In summary: the content-id chain provides integrity but not confidentiality; truncation is undetectable without a head commitment; decompression and nested-GTS recursion MUST be bounded; and signatures attest a signer over bytes, not the truth of claims.
- **Interoperability considerations:** the `+cbor-seq` structured-syntax suffix ([RFC 8742]) signals that the payload is a CBOR Sequence, so generic sequence tooling can inspect the ordered data items before applying GTS-specific rules. The self-describe tag `55799` ([RFC 8949] §3.4.6) MAY tag each segment header as a magic number. Conformance is defined by the shared test-vector corpus (§19).
- **Published specification:** this document (GTS — Graph Transport Substrate — Specification).
- **Applications that use this media type:** content-addressed RDF 1.2 graph transport and archival; signed agent-memory and provenance artifacts; package distribution where the payload bundles a graph and the binaries it references.
- **Fragment identifier considerations:** none.
- **Additional information:**
  - **Magic number(s):** `0xd9 0xd9 0xf7` (the CBOR self-describe tag `55799`) when present at the start of the file (§16.1). This prefix is OPTIONAL because the first segment header MAY be untagged.
  - **File extension(s):** `.gts`
  - **Macintosh file type code(s):** none
- **Person & email address to contact for further information:** Patrick Audley paudley@blackcat informatics.ca
- **Intended usage:** COMMON
- **Restrictions on usage:** none
- **Author / Change controller:** Blackcat Informatics® Inc.

1.25 21. Complete CDDL appendix

This appendix is the copyable schema surface for implementers. Inline CDDL fragments earlier in this document explain local context; this appendix collects the wire-level map shapes in one place.

1.25.1 21.1 Sequence grammar

A GTS file is a **CBOR Sequence**, not one enclosing CBOR item. CDDL describes the individual items in that sequence; the sequence grammar is defined in English and ABNF-like notation:

```
gts-file = 1*segment
segment  = [ self-describe-tag ] header *frame
```

`self-describe-tag` is CBOR tag 55799 applied to the header item only. It is a wire-level magic hint, not a member of the Header map, and it is not part of the Header `"id"` preimage (§22). Each segment begins with a Header and then zero or more frame items until the next Header or EOF (§3.1).

1.25.2 21.2 Copyable CDDL

```
; GTS v1 item grammar. The top-level file is a CBOR Sequence (§21.1).

gts-item = header-item / frame
header-item = header / self-described-header
self-described-header = #6.55799(header)

term-id = uint
frame-index = uint
codec-id = uint
```

```
digest = bstr .size 32
content-id = digest
blake3-uri = tstr                   ; "blake3:" + 64 lowercase hex characters
digest-ref = digest / blake3-uri
profile-name = tstr
layout-state = "streamable" / tstr
extension-key = tstr                ; any text key not defined by that map shape

header = {
  "gts": "GTS1",
  "v": 1,
  "prof": profile-name,
  "cat": { * codec-id => codec },
  ? "layout": layout-state,
  ? "dct": { * tstr => bstr },
  ? "meta": any,
  "id": content-id,
  * extension-key => any,
}

codec = {
  "name": tstr,
  "cls": "encode" / "compress" / "encrypt",
  ? "dct": tstr,
  ? "p": any,
  * extension-key => any,
}

frame = {
  "t": frame-type,
  ? "x": [+ codec-id],
  ? "pub": any,
  ? "to": [+ recipient],
  ? "d": frame-payload / bstr,
  "prev": content-id,
  "id": content-id,
  ? "sig": cose-sign1,
  * extension-key => any,
}

frame-type = "terms" / "quads" / "reifies" / "annot" / "blob" / "suppress"
/ "snapshot" / "meta" / "index" / "opaque"

recipient = {
  "kid": tstr,
  ? "alg": tstr,
  * extension-key => any,
}

cose-sign1 = bstr                   ; serialized COSE_Sign1, detached payload = frame "id"

frame-payload = terms-payload / quads-payload / reifies-payload / annot-payload
/ blob-payload / suppress-payload / snapshot-payload / meta-payload
/ index-payload / opaque-node
```

```
terms-payload = [+ term]
term = {
  "k": 0 / 1 / 2 / 3,              ; 0=IRI, 1=literal, 2=bnode, 3=quoted triple
  ? "v": tstr,
  ? "dt": term-id,
  ? "l": tstr,
  ? "rf": term-id,
  * extension-key => any,
}

triple-row = [term-id, term-id, term-id]
quad-row = [term-id, term-id, term-id] / [term-id, term-id, term-id, term-id]

quads-payload = [+ quad-row]
reifies-payload = { * term-id => triple-row }
annot-payload = [+ triple-row]

blob-payload = bstr
blob-pub = {
  ? "mt": tstr,
  ? "rep": tstr,
  ? "digest": digest-ref,
  * extension-key => any,
}

suppress-payload = {
  "targets": [+ suppress-target],
  ? "reason": tstr,
  ? "by": term-id,
  * extension-key => any,
}

suppress-target = suppress-frame / suppress-blob / suppress-term
/ suppress-quad / suppress-reifier
suppress-frame = { "kind": "frame", "id": digest-ref, * extension-key => any }
suppress-blob = { "kind": "blob", "digest": digest-ref, * extension-key => any }
suppress-term = { "kind": "term", "id": term-id, * extension-key => any }
suppress-quad = { "kind": "quad", "q": quad-row, * extension-key => any }
suppress-reifier = { "kind": "reifier", "id": term-id, * extension-key => any }

snapshot-payload = {
  "terms": terms-payload,
  ? "quads": quads-payload,
  ? "reifies": reifies-payload,
  ? "annot": annot-payload,
  ? "blobs": { * digest-ref => bstr },
  ? "meta": any,
  * extension-key => any,
}

meta-payload = any

index-payload = {
```

```
  "count": uint,
  "head": content-id,
  ? "off": [+ uint],
  ? "ti": { * frame-type => [+ frame-index] },
  ? "dict": [+ frame-index],
  ? "mmr": content-id,
  * extension-key => any,
}

opaque-node = {
  "id": content-id,
  "type": frame-type,
  ? "pub": any,
  ? "to": [+ recipient],
  ? "sigstat": sig-status,
  "reason": opaque-reason,
  * extension-key => any,
}

sig-status = "none" / "valid" / "invalid" / "unverified"
opaque-reason = "unknown-codec" / "missing-key" / "damaged"
/ "unknown-frame-type"

diagnostic = {
  "code": diagnostic-code,
  "detail": tstr,
  ? "frame_index": frame-index,
  * extension-key => any,
}

diagnostic-code = "EmptyFile"
/ "TornAppendError" / "DamagedFrame" / "BrokenChain"
/ "TruncatedLog" / "UnknownCodec" / "MissingKey"
/ "KeyWrapFailed" / "ConflictingReifier" / "RecursionLimit"
/ "StreamableLayoutError" / "PositionConstraint"
/ "ForwardReference" / "SegmentBoundary" / "IndexMmrError"
/ "UnknownFrameType" / tstr

profile-status = "core-required" / "optional-standard" / "experimental"
/ "domain-specific"
profile-registration = {
  "name": profile-name,
  "status": profile-status,
  ? "owner": tstr,
  ? "spec": tstr,
  ? "namespace": [+ tstr],
  ? "requires": any,
  ? "validation": any,
  ? "security": any,
  * extension-key => any,
}
```

When `"x"` is present and non-empty, the frame `"d"` value is a byte string carrying the encoded/compressed/encrypted payload. After reversing the transform chain (§6.1), those bytes decode to the frame-type-specific pay-

load above, except `blob`, whose decoded payload is raw bytes. When `"x"` is absent, `"d"` carries the frame-type-specific payload directly.

`blob-pub` is the conventional shape of a blob frame's `"pub"` map; the frame envelope keeps `"pub"` typed as `any` so profiles can layer additional public metadata without changing the core frame grammar. `digest-ref` accepts both the raw 32-byte digest and the `blake3:<hex>` text form used by the reference engines.

1.26 22. Hash, signature, and extension-key preimages

All preimages in this section use the deterministic CBOR rules in §4: definite lengths, shortest-form integers, and map keys sorted bytewise by their encoded CBOR form. Unless a row explicitly excludes a field, every key/value pair in the map participates, including unknown extension keys.

1.26.1 22.1 Preimage and subject table

subject	bytes hashed or signed	excluded fields	included extension fields	verifier behavior
Header `"id"`	`BLAKE3-256(deterministic-CBOR(header-map without "id"))`	`"id"`; CBOR tag 55799	All unknown Header keys participate.	Recompute before accepting the segment Header; mismatch is header tampering.
Frame `"id"`	`BLAKE3-256(deterministic-CBOR(frame-map without "id" and "sig"))`	`"id"` and `"sig"` only.	All unknown frame keys participate.	Recompute for every frame; mismatch is `DamagedFrame`.
Frame `"prev"` link	The `"prev"` value is included in the frame `"id"` preimage.	None beyond the frame `"id"` exclusions.	Unknown frame keys do not change `"prev"` semantics but are still in the frame `"id"` preimage.	Compare to the previous item's `"id"` within the same segment; mismatch is `BrokenChain`.
COSE frame signature	Detached COSE_Sign1 over the frame `"id"` bytes. The COSE Sig_structure is `["Signature1", protected, h'', frame-id]`; the COSE payload field is `null`/detached.	The signature is not part of the frame `"id"` preimage because `"sig"` is excluded there.	Extension keys affect the signature indirectly by changing the frame `"id"`.	Verify with the key resolved by `kid`; report `valid`, `invalid`, or `unverified`.
Inline blob digest	`BLAKE3-256(decoded blob bytes)`, after reversing transforms and decryption when available.	Frame envelope fields are not part of the blob digest.	Blob-public extension keys do not affect the blob digest, but they do affect the containing frame `"id"`.	Compare with `pub.digest` when present and with graph references that name the blob.

subject	bytes hashed or signed	excluded fields	included extension fields	verifier behavior
External blob digest	`pub.digest` names bytes stored elsewhere; the digest subject is those external bytes.	The external bytes are absent from the GTS frame, so only the digest claim participates in the frame `"id"` through `"pub"`.	Unknown public metadata participates in the frame `"id"`, not in the external blob digest.	A verifier can check only when it obtains the external bytes.
Index `"head"`	The content-id of the last covered frame, where `"count"` is the number of frames covered by the index payload.	Not applicable.	Unknown index-payload keys participate in the index frame `"id"`, not in the `"head"` subject.	Compare `"head"` to the covered frame id; mismatch invalidates the index/layout claim.
Index `"mmr"`	Merkle-Mountain-Range root over the ordered frame ids covered by the index, using the leaf/parent/root preimages in §6.2.	The index frame itself is not covered unless a later index covers it.	Unknown index-payload keys participate in the index frame `"id"`, not in the MMR root.	Use as an optional whole-covered-region commitment and proof root; mismatch is `IndexMmrError`.
Detached signature provenance	`stream:sourceFrame` names the original frame `"id"`; `stream:cose` carries the original COSE_Sign1 bytes. The signature still verifies over the original frame id.	The rewritten frame's new `"id"` is not the old signature subject.	Provenance graph extension terms do not change the original signature subject.	Verify carried signatures against `stream:sourceFrame`; do not treat them as signatures over the compacted frame.

1.26.2 22.2 Unknown extension-key behavior

An extension key is a text-string map key not defined by that map's CDDL production. Defined reserved keys such as `"id"`, `"sig"`, `"prev"`, `"t"`, `"d"`, `"x"`, `"pub"`, and `"to"` are not extension keys and MUST NOT be repurposed by profiles.

Readers MUST include unknown extension keys when recomputing Header and frame preimages. A reader MUST NOT reject a Header, frame, codec, recipient, term, payload, opaque-node, diagnostic, or profile-registration map solely because it contains an unknown extension key. Unknown keys have no core fold semantics unless a supported profile or extension defines them.

Re-emit behavior depends on the operation:

- Byte-preserving operations such as raw `cat`, copying, mirroring, or serving preserve unknown keys naturally because they preserve the original bytes.
- A tool that decodes and re-emits a Header or frame while claiming to preserve the same logical item MUST copy unknown extension keys verbatim before recomputing `"id"` values.
- A tool that cannot preserve unknown extension keys MUST treat the operation as lossy re-authoring, MUST recompute affected `"id"` and `"prev"` values, and MUST NOT claim existing frame signatures remain attached to the rewritten frames.

- A compactor or other re-authoring tool MAY preserve old frame signatures only as detached provenance (§10.1), where the old frame id remains the explicit signature subject.

Because extension keys participate in preimages, extension authors can add tamper-evident metadata without changing core GTS grammar. They cannot change header/frame grammar, hash preimages, signature subjects, or fold semantics (§2.1, §13).

1.27 23. References

1.27.1 23.1 Normative references

- [**RFC 2119**] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, March 1997.
- [**RFC 8174**] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, May 2017.
- [**RFC 8949**] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, December 2020.
- [**RFC 8742**] Bormann, C., "Concise Binary Object Representation (CBOR) Sequences", February 2020.
- [**RFC 9052**] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, August 2022.
- [**RFC 9053**] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", August 2022.
- [**RFC 9277**] Bormann, C. and M. Nottingham, "On the Use of Structured Suffixes in Media Types", June 2022.
- [**RFC 6838**] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, January 2013.
- [**RFC 3339**] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", July 2002.
- [**BCP 47**] Phillips, A. and M. Davis, "Tags for Identifying Languages", September 2009.
- [**BLAKE3**] O'Connor, J., Aumasson, J-P., Neves, S., and Z. Wilcox-O'Hearn, "BLAKE3: one function, fast everywhere" (256-bit output used here).
- [**RDF 1.2**] W3C, "RDF 1.2 Concepts and Abstract Data Model", Candidate Recommendation Snapshot, 07 April 2026, https://www.w3.org/TR/2026/CR-rdf12-concepts-20260407/ — the RDF terms, dataset model, triple-term, and `rdf:reifies` substrate imported by §7.

1.27.2 23.2 Informative references

- [**RFC 7049**] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", October 2013 (obsoleted by [RFC 8949]; cited only for its legacy length-first "canonical" ordering, §4).
- [**RFC 8610**] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL)", June 2019.
- [**RFC 9111**] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", June 2022 (the caching directives of §16.4).
- [**RFC 6906**] Wilde, E., "The 'profile' Link Relation Type", March 2013 (the `Accept-Profile` future extension noted in §16.4).

GTS is intentionally a transport format, not an ontology or graph store. A conforming implementation preserves the append-only, content-addressed fold so independent projections can be regenerated from the same bytes.