
Epiphany

TEXT PROJECTION

A companion to the Core Specification

Version 0.8.0 — The genesis operation vocabulary reaches the grammar

Normative for the text form it defines

Contents

1	About This Companion	2
1.1	What This Document Covers	2
1.2	The Subject of the Projection	2
2	The Canonical Text Form	4
2.1	Encoding and Character Set	4
2.2	Atoms	4
2.3	Layout	5
3	What Is Projected	6
3.1	Derive, or Carry — Never Both	6
3.2	Document Structure	7
3.3	Canonical Blobs	7
3.4	Profile Declarations	8
3.5	Extension Declarations	9
3.6	Ordering What the Binary Form Ordered Physically	9
3.7	Projecting Canonical Values	10
3.8	Reduced State	13
3.9	The Canonical Base Snapshot	13
4	Requirements	15
4.1	Canonicity	15
4.2	Round Trip	15
4.3	Strict Parsing	16
4.4	Conformance	16
5	Grammar	17
6	A Worked Example	21
7	Revision History	22

About This Companion

The *Text Projection* is a companion to the Epiphany Core Specification. It fulfils the delegation the core specification makes in Chapter 8, TEXT PROJECTION (`sec:format:textproj`), which declares that “the format admits a deterministic projection to a canonical s-expression text form” and that “the text projection is normative” — while leaving the form itself unwritten. The Binary Format companion likewise excludes it: “the canonical s-expression form — that is the *Text Projection* companion’s”.

This document supplies that form.

1.1 What This Document Covers

- The canonical text syntax: atoms, byte strings, text, and the layout that makes the projection deterministic.
- What is projected, and what is deliberately not.
- The projection and parse requirements, including the bidirectional round-trip the core specification demands.

It does *not* cover the binary encoding of anything — that is the Binary Format companion’s — nor the semantics of any operation, which is the Operation Catalog’s. The projection is a *re-presentation* of the canonical document, never a second definition of it. Wherever the two could disagree, the binary form is normative and the projection is wrong.

1.2 The Subject of the Projection

The projection’s subject is the **canonical document**, not the file. Per the core specification’s Chapter 8 TEXT PROJECTION, the projection **MUST NOT** be required to preserve chunk offsets, compression choices, cache chunks (operation indexes, layout caches, integrity indexes), garbage bytes from prior commits, or superblock generation numbers, slot assignments, and CRCs.

Consequently two bundles that differ only in physical layout project to the same text, and a text re-serializes to *a* bundle rather than to *the* bundle it came from. That is the intent, not a limitation: the physical file is an encoding of the document, and the projection is of the document.

A bundle **MAY** cache its own projection in a `TextProjection` chunk named by `Manifest.text_projection_root`. That chunk is a **non-canonical accelerator** (core specification Chapter 8, SCHEMA VERSIONING): a reader need not understand it, and a writer preserves it verbatim or discards it. A cached projection that disagrees with the operations it claims to project is *stale*, not authoritative.

The Canonical Text Form

2.1 Encoding and Character Set

REQUIREMENT 2.1

A text projection **MUST** be UTF-8, with no byte-order mark. Every text field it carries **MUST** be in Unicode NFC, matching the canonical-text rule of the core specification's Appendix D, TEXT AND UNICODE. A parser **MUST** reject non-NFC text rather than normalize it.

2.2 Atoms

An *atom* is a symbol, an integer, a byte string, or a text string.

Symbols are lowercase ASCII words, possibly hyphenated: `envelope`, `insert-event`, `strict-inverse`. They name constructors and enumeration cases. A symbol is never quoted.

Integers are written in base ten, with a leading `-` for negative values, no leading zeros, and no leading `+`. Zero is `0`, never `-0`.

Byte strings carry every identifier, hash, and opaque payload.

REQUIREMENT 2.2

A byte string **MUST** be written as `#x` followed by an even number of **lowercase** hexadecimal digits, one pair per byte, in the order the bytes appear in the canonical binary form. The empty byte string is `#x`. A parser **MUST** reject uppercase digits, an odd digit count, and any separator within the digits.

RATIONALE

One rule for every byte string. Hexadecimal has no alphabet variant and no padding to canonicalize, it is greppable, and a corrupted character is locally obvious. Identifiers are 16 or 32 bytes, so its expansion costs nothing where it is read; only an inlined snapshot pays. The core specification permits “base64 or another canonical text form”; a second

encoding would buy a quarter of the bytes of the one body nobody reads, at the price of pinning an alphabet, a padding rule, and a line-wrapping rule, and of choosing which encoding applies where. Ratified at 0.1.0.

Text strings are double-quoted. Inside a string, `"` denotes a quotation mark, `\` a backslash, `\n` a line feed, and `\t` a tab; no other escape exists.

REQUIREMENT 2.3

A text string **MUST** escape exactly the characters that require it: the quotation mark, the backslash, U+000A, and U+0009. Every other character **MUST** appear literally. A parser **MUST** reject an escape sequence outside this set, and **MUST** reject a literal character that the writer was required to escape.

RATIONALE

“Escape exactly” rather than “escape at least”: the text is canonical, so two spellings of one string cannot both be valid. This is the same injectivity the binary form rests on (Binary Format, `req:binfmt:decode-vectors`), stated for text.

2.3 Layout

REQUIREMENT 2.4

A projection is a sequence of lines separated by a single U+000A, with a final U+000A and no other trailing whitespace. Each line is one complete s-expression. Tokens within a line are separated by exactly one space; there is no other whitespace, and no indentation. Each operation envelope **MUST** occupy exactly one line.

RATIONALE

The core specification’s stated use case is that “merge conflicts surface at the operation-envelope level, which is the meaningful level for collaborative editing”. One envelope per line makes a line-based three-way merge conflict *exactly* an envelope conflict — never a conflict inside an envelope, which could otherwise produce a syntactically valid operation that neither side wrote. It also disposes of indentation: there is no whitespace to canonicalize, so the “identical semantics project to identical text” requirement below has nothing to hide in.

The lines are long. Readability is a *tooling* concern, and a pretty-printer is free to reformat for display; what it must not do is write the reformatted text back and call it a projection. Ratified at 0.1.0.

What Is Projected

3.1 Derive, or Carry — Never Both

The manifest’s references are *physical*. A `ChunkRef` carries an offset, a compressed length, and a compression algorithm; a `BlobRef` carries the same. Those are exactly the things the projection **MUST NOT** preserve. But a reference also carries an *identity* — a chunk id, a content hash, a blob id — and every one of those is a function of the content.

One rule resolves both, and it is the rule Requirement 3.11 already applies to reduced state.

REQUIREMENT 3.1

A projection **MUST** carry exactly what the document does not determine, and **MUST NOT** carry anything it does.

- **Physical attributes** — a chunk’s or blob’s `offset`, `compressed_length`, and `compression`, and a chunk’s `uncompressed_length` — **MUST NOT** appear. A serializer chooses them freely.
- **Derivable identities** — `ChunkId`, `ContentHash`, `BlobId` — **MUST NOT** appear. They are re-derived from the content by the derivations the Binary Format companion pins (`CONTENT HASHING`, `DOMAIN-SEPARATED PREIMAGES`).
- **Non-derivable identities** **MUST** appear. In schema major 0 there is exactly one: `SnapshotId`, which the Binary Format companion declares opaque, with readers forbidden from deriving it (`req:binfmt:snapshot-id-opaque`).
- **Content and semantic attributes** **MUST** appear: a chunk’s `kind` and `schema_version` and its uncompressed payload; a blob’s media type, declared maximum uncompressed length if any, and its payload.

A parser **MUST** reject a projection carrying a value this requirement forbids.

RATIONALE

Carrying a derivable identity would reproduce, at the level of a chunk, the defect Requirement 3.11 rules out at the level of the document: two sources of truth for one fact, with nothing to stop them disagreeing. Carrying a physical attribute would make two encodings of one document project to two texts, breaking Requirement 4.1.

`SnapshotId` is the sole exception, and it is an exception for a stated reason rather than an oversight: `vo` has no snapshot producer, so the identity has nothing to derive *from*, and the Binary Format companion accordingly pins it as sixteen opaque bytes that a reader **MUST NOT** attempt to verify. A projection must carry what it cannot recompute.

3.2 Document Structure

A projection is, in order:

1. a `(text-projection <version>)` header line, naming the version of *this companion* the text conforms to;
2. a `(document #x<document-id>)` line, and a `(lineage #x<lineage-id>)` line if the manifest declares one;
3. zero or more `(profile ...)` lines, in canonical order;
4. zero or more `(extension ...)` lines, in canonical order;
5. at most one `(canonical-base ...)` line;
6. zero or more `(blob ...)` lines, in canonical order;
7. zero or more `(envelope ...)` lines, in canonical operation order (core specification Appendix D).

REQUIREMENT 3.2

A parser implementing this companion **MUST** accept exactly one header version: `(0 8 0)`, the version of the companion it implements. It **MUST** reject any other version at line one.

Multi-version acceptance and text migrate-on-read are deferred in the same posture as op-payload migrate-on-read: support belongs in an explicit, version-keyed migration path when a real consumer requires it. The current parser **MUST NOT** speculate by accepting another version.

Every sequence is written in the normative order its binary counterpart uses. The projection introduces no ordering of its own.

3.3 Canonical Blobs

REQUIREMENT 3.3

A blob referenced by a canonical operation or by canonical reduced state is itself canonical (core specification Chapter 8, CANONICAL AND NON-CANONICAL ROOTS). Every such blob **MUST** be projected as a `(blob ...)` line carrying its media type, its declared maximum uncompressed length if it declares one, and its uncompressed payload.

Its `BlobId`, content hash, offset, lengths, and compression are re-derived (Requirement 3.1). The blob lines are ordered and de-duplicated by their projected form (Re-

quirement 3.7), not by the binary order, which reads the offset.

A blob referenced only by acceleration structures is non-canonical and **MUST NOT** be projected.

Parser handling of unreferenced blob lines is specified by Requirement 3.4.

RATIONALE

An embedded image, font, or audio recording that a canonical operation references is part of the document. Omitting it would make the projection lossy for exactly the documents most in need of archival — and lossy *silently*, since the operations that reference the blob would still be there, pointing at a blob id the text no longer contains. This requirement was absent from version 0.1.0 of this companion, and from the core specification’s own list of what the projection preserves; both are corrected.

REQUIREMENT 3.4

A parser **MUST** reject every `(blob ...)` line whose blob is unreferenced by canonical state (Requirement 3.3). At companion version 0.8.0, neither a canonical operation nor canonical reduced state can carry a `BlobId`; canonical state therefore cannot reference a blob, and a parser **MUST** reject every `(blob ...)` line.

RATIONALE

A blob-bearing text at this version is necessarily non-canonical. Accepting one stages a blob into the bundle that the next projection silently drops, causing data loss and falsifying $\text{project}(\text{serialize}(\text{parse}(T))) = T$ for that text. Forward compatibility belongs to header-version gating (Requirement 3.2), not to leniency here.

3.4 Profile Declarations

A `(profile ...)` line carries the profile’s identity, its semantic version, and its constraints.

REQUIREMENT 3.5

A profile identity is a symbol for each closed-vocabulary profile (`full`, `read-only`, `lite`), and `(custom #x<registry-id>)` for `ProfileId::Custom`, whose registry id is sixteen bytes.

RATIONALE

Version 0.1.0 required a symbol for every profile, which made a custom profile unrepresentable and its claim to preserve “all profile declarations” false.

3.5 Extension Declarations

REQUIREMENT 3.6

An `(extension ...)` line **MUST** carry every field of the declaration: its identity, its semantic version, whether it is required, its affected object kinds, its edit barriers, and its preserved chunk roots. The affected object kinds and the edit barriers are opaque to the bundle and are projected as byte strings, verbatim.

Fields appear in the ratified declaration order (core specification Chapter 8, EXTENSION DECLARATIONS).

Each preserved chunk root is projected as its `kind`, its `schema_version`, and its uncompressed payload (Requirement 3.1), never as a `ChunkRef`: a `ChunkRef` is a physical reference, and the projection has no file to point into. The chunk list is ordered and de-duplicated by projected form (Requirement 3.7), because `ChunkRef`'s binary order breaks ties on the offset.

RATIONALE

`affected_object_kinds` and `edit_barriers` have ratified structured shapes (`ObjectKind`, `EditBarrier`) and canonical byte encodings, and the bundle stores them opaquely: it preserves them across reads and writes without interpreting them. At 0.3.0 the projection does the same, carrying their canonical bytes, on the principle that the projection interprets nothing the bundle does not.

This is a deferral, not a conclusion. A later revision **MAY** project them structurally under Requirement 3.10; because their canonical bytes are unchanged by that, doing so does not change the document, only its text — and that is a version-gated change to this companion, not to the format.

RATIONALE

Version 0.1.0 carried only identity, the required flag, barriers, and a list of chunks, dropping the semantic version and the affected object kinds outright and leaving the chunks' representation undefined. An extension's declaration governs how the canonical document is *interpreted*; a projection that loses part of it does not determine the document.

3.6 Ordering What the Binary Form Ordered Physically

"Keep the binary form's order" is available only where the binary order is a function of data the projection preserves. Two sequences fail that test.

REQUIREMENT 3.7

Where a sequence's binary order depends on attributes Requirement 3.1 erases, the projection **MUST** order its elements by their **projected form**, ascending, comparing the UTF-8 bytes of the rendered element; and **MUST** emit at most one element per distinct

projected form.

In schema major 0 this applies to exactly two sequences:

- the (blob ...) lines, whose binary counterpart `blob_roots` is sorted by the full `BlobRef` encoding — which contains the offset, the compressed length, and the compression algorithm; and
- an extension's preserved chunk roots, sorted in binary by `ChunkRef`'s order, whose key is the kind, then the content hash, then the **offset**, with the compressed length, the uncompressed length, and the compression as further tie-breakers.

Every other projected sequence keeps the binary order, because every other binary order reads only preserved data. In particular the profile and extension declarations are sorted in binary by the semantic keys (`profile_id, version`) and (`extension_id, version`), and the envelopes by canonical operation order.

RATIONALE

Two bundles that differ only in physical layout are the *same document*, and Requirement 4.1 obliges them to project to byte-identical text. Inheriting the binary order for these two sequences would let a chunk's file offset decide the order of the text — so relocating a chunk, which changes no semantics, would change the projection. That is precisely the failure the requirement forbids.

The de-duplication is the same point from the other side. A chunk is content-addressed: two entries with identical kind, schema version, and payload *are* one chunk, and appear twice only because the writer stored the bytes twice. A blob with identical media type, declared maximum, and payload is one blob. Their projected forms are identical, so the binary form's distinction between them is a physical fact — and a duplicated line would smuggle that physical fact into a text that claims to have erased it. Emitting one line is not a loss; it is the erasure working.

Ordering by the projected form is total and deterministic, and it reads nothing but what is projected. For chunks it coincides with ordering by the derived `ChunkId`, since that id is a function of exactly the kind, schema, and payload the line carries — which is a pleasing check that the rule is reading the right thing.

3.7 Projecting Canonical Values

An operation payload embeds canonical values from the core specification's Chapter 5 — an *Event*, a *Pitch*, a *Region*, a *TimeSignature*. This document does *not* restate their shapes. It states one rule for turning any of them into text, and the shapes stay where they are ratified.

REQUIREMENT 3.8

The core specification's Chapter 6 operation vocabulary — the envelope, its stamp and causal context, the payload, the operation kinds, and their sub-vocabularies — **MUST** be projected by the productions of Chapter 5, not by Requirement 3.10. The value-projection rule governs exactly the `value` positions those productions contain, and no

other positions.

An operation kind’s payload record **MUST** be inlined into its production. The record exists so each variant can name a type; it is not a modelling distinction, and the binary form agrees: `OperationKind`’s encoding writes the kind tag and then delegates to the record, adding no bytes for the wrapper. It adds no text here either, for the same reason clause 2 of Requirement 3.10 makes a newtype transparent.

REQUIREMENT 3.9

The projection is **schema-directed**. At every position a reader knows the type it expects, from the ratified schema and from the position itself, exactly as the binary decoder does. The grammar of Chapter 5 describes the *shape* of the text; it is not a standalone unambiguous language, and a reader **MUST NOT** attempt to recover a value’s type from its shape.

Three consequences follow, and a reader resolves each by the type it expects:

- `()` is the empty sequence, and it is the absent option.
- A bare symbol is a fieldless variant, and it is a struct with no fields.
- A parenthesised list whose first element is a symbol is a struct, and it is a sequence whose first element is a symbol.

RATIONALE

This states what the ratified rules already require rather than adding a new constraint. A struct is `(<type-name> <field>...)` and a sequence is a parenthesised list of its elements; a sequence whose first element is a fieldless variant is therefore shape-identical to a struct. The collision is *irreducible* without new syntax, and it is reachable from the first pitched note a score contains: a `PitchedEvent` carries articulations and ornaments, sequences of a zero-field `ArticulationMark` and `OrnamentMark`, alongside an optional `DynamicMark` — so one `insert-event` line holds an empty sequence and an absent option, both spelled `()`, and a two-element sequence spelled like a two-field struct.

Making the text standalone-unambiguous would buy nothing, because Requirement 4.3 already obliges a parser to reject “a duplicate in a set-typed field” — which it cannot do without knowing that the field is set-typed. The parser consults the schema either way. Spending syntax to remove a shape ambiguity, while leaving the schema dependency it was meant to remove, would make every line longer and no tool simpler.

The binary form is schema-directed for the same reason and pays the same price: its bytes do not say what they are either.

REQUIREMENT 3.10

A canonical value is projected thus:

1. A **struct** becomes `(<type-name> <field>...)`, where the type name is the core specification’s name in lower-case hyphenated form and the fields appear *positionally*, in the order that specification’s ratified listing declares them. Field names are not

written. A struct with *no* fields is the bare symbol `<type-name>`, as a fieldless variant is; it encodes to no bytes in the binary form and carries no value here.

2. A **newtype** — a struct of exactly one unnamed field — is projected as that field alone, with no wrapper. This mirrors the binary form, in which a newtype delegates to its field and adds no bytes.
3. A **tagged union** becomes `(<variant-name> <field>...)`. A variant with no fields is the bare symbol `<variant-name>`.
4. An **option** is `()` when absent and `(some <value>)` when present.
5. A **sequence**, **set**, or **map** is a parenthesised list of its elements, *in the order the binary form writes them*, except where Requirement 3.7 applies; a map entry is `(<key> <value>)`. An empty one is `()`, which Requirement 3.9 distinguishes from an absent option by the expected type. The projection invents no ordering of its own, and a set that the binary form writes strictly increasing is written strictly increasing here (Requirement 4.3).

A **byte string** is not a sequence. Where the binary form writes a length-prefixed run of bytes rather than a counted sequence of elements — an opaque extension payload, a `SoundConfiguration` — the projection writes a byte string, not a list of integers.

6. **Leaves.** An identifier or hash is a byte string. An integer is an integer. A boolean is `true` or `false`. Canonical text is a quoted string. A rational is `(ratio <numerator> <denominator>)`, in lowest terms with a positive denominator and the sign on the numerator; zero is `(ratio 0 1)`. A `CanonicalF64` is a byte string of its eight canonical little-endian IEEE 754 bytes.

RATIONALE

One rule, not forty productions. Spelling out a production per value type would restate the entire Chapter 5 data model in a second normative document, and two normative listings of one struct is precisely the drift this project has already been bitten by. A rule cannot drift from the listing it reads.

Positional fields. The declaration order is already normative — the binary form depends on it — so field names would be redundant, would double the length of every line, and would create a second thing to keep in step with a rename. The constructor name carries the context a reader needs.

Newtypes are transparent for the same reason they are transparent in the binary form: a `MusicalPosition` *is* a rational, and wrapping it would put a distinction in the text that the document does not make.

Floats are bytes, never decimal. A decimal rendering of an `f64` is not canonically unique — shortest-round-trip and seventeen-significant-digit forms both round-trip, and `-0.0` has two spellings — so a decimal float would break Requirement 4.1 at the first tempo mark. The core specification already forbids computed floats in canonical state and stores the eight bytes; the projection carries those eight bytes. This costs readability in exactly one place, and buys canonicity everywhere.

3.8 Reduced State

REQUIREMENT 3.11

The projection **MUST** preserve canonical reduced state *by preserving the operations that determine it*. It **MUST NOT** carry a second, literal copy of the reduced state.

RATIONALE

Reduced state is a deterministic function of the operation set and the canonical base (core specification Chapter 6, DESIGN PRINCIPLES). A text that carried both would have two sources of truth for one fact, and nothing could stop them disagreeing — a projection with an internally contradictory document is worse than no projection. This reading is what the core specification’s own round-trip clause already implies: text that “parses to identical canonical document semantics” must re-serialize to bundles with identical semantics, and reduced state is a function of semantics.

Read the core specification’s “all canonical reduced state” as *determines*, not *contains*. Ratified at 0.1.0.

3.9 The Canonical Base Snapshot

REQUIREMENT 3.12

If the manifest declares a canonical base, the projection **MUST** carry a (canonical-base . . .) line bearing:

- the `SnapshotId`, verbatim — the one identity that is carried rather than derived (Requirement 3.1);
- the causal frontier it materializes, as opaque bytes;
- its reduction-algorithm version;
- the profile under which it was produced;
- its root chunk’s `schema_version`; and
- *the root chunk’s uncompressed payload, inline*, as a single byte string. That payload is the canonical byte form of the reduced state the snapshot materializes.

The root chunk’s kind is `Snapshot` by role and is not written. The `SnapshotRef`’s `root` and `hash` are **re-derived**: the chunk’s content hash is `hash(Snapshot, schema, payload)` under the Binary Format companion’s chunk-hash preimage, and it is both the root’s `ChunkId` and the `SnapshotRef`’s `hash`. A parser **MUST** perform that derivation rather than read it from the text.

RATIONALE

A canonical base exists precisely so that the operations before its frontier need not be retained. Where they have been pruned, the snapshot is *not* derivable from anything

else in the document, and a projection that carried only a reference would be **lossy** — the text would no longer determine the document, which is the one thing it is for. The core specification permits the payload to be “encoded compactly or referenced externally”; inline and compact is the choice that keeps the text self-contained.

The snapshot diffs as one opaque atom. That is acceptable: merges happen among operations, and a base snapshot changes only when the document is compacted, at which point the whole line changes anyway. A later revision **MAY** project the snapshot structurally; doing so does not break the round trip, because the document it denotes is unchanged. Ratified at 0.1.0.

Schema major 0 has no snapshot producer — pruning and canonical-base creation are deferred (Binary Format, `SNAPSHOTID`) — so this requirement binds whoever writes the first one, and cannot be exercised before then.

Requirements

4.1 Canonically

REQUIREMENT 4.1

Two bundles whose canonical document semantics are identical **MUST** project to **byte-identical** text. A projector **MUST NOT** have any freedom the document does not determine: no optional whitespace, no alternative spelling of an atom, no ordering choice.

4.2 Round Trip

REQUIREMENT 4.2

Parsing a projection and re-serializing it to binary **MUST** yield a bundle whose canonical document semantics are identical to the original's. The bundle's physical layout, chunking, and compression **MAY** differ.

Equivalently, and more usefully to an implementer: for every bundle B ,

$$\text{semantics}(\text{parse}(\text{project}(B))) = \text{semantics}(B),$$

and for every valid projection T ,

$$\text{project}(\text{serialize}(\text{parse}(T))) = T.$$

The second equation is the text's own injectivity: it is *stronger* than the first, and it is the one a conformance test can check with byte equality.

4.3 Strict Parsing

REQUIREMENT 4.3

A parser **MUST** reject any text that is not the canonical projection of the document it denotes. It **MUST NOT** normalize: not whitespace, not letter case in a byte string, not an escape sequence, not an out-of-order sequence, not a duplicate in a set-typed field. Accepting non-canonical text and normalizing it *is* accepting it, and does not satisfy this requirement.

Rejecting a duplicate in a set-typed field requires knowing that the field is set-typed. A parser therefore reads the schema (Requirement 3.9); shape alone never suffices.

RATIONALE

This is the same discipline the binary decoders carry, and it exists for the same reason: a lenient parser makes two texts denote one document, and the projection's contract is that a text *determines* its document. The Binary Format companion learned this concretely — a whole-value re-encode guard catches the fields a decoder normalizes and is blind to order-preserving sequences, and a guard on an outer value can *mask* a lenient inner codec rather than fix it (Binary Format, THE DECODE VECTOR CORPUS). A text parser inherits both hazards, and the cheapest total defence is the same one: re-project the parsed document and compare, *and* check per-site the orders that re-projection would restore.

4.4 Conformance

REQUIREMENT 4.4

An implementation claiming Text Projection conformance **MUST** implement both directions. A projector alone does not conform: the round trip (Requirement 4.2) is the requirement, and half of it is not checkable.

Grammar

```
; Notation. X* is zero or more X, X? is zero or one. Within a line, adjacent
; elements of a repetition are separated by exactly one U+0020, and an empty
; repetition contributes nothing -- so "(" value* ")" spells () when empty. The
; line productions below carry their own trailing LF, and are simply
; concatenated. LF is U+000A. Terminals are quoted; U+XXXX names a codepoint.
```

```
projection      ::= header document lineage? profile* extension*
                  canonical-base? blob* envelope*

header          ::= "(text-projection " version ")" LF
version         ::= "(" integer " " integer " " integer ")" LF

document        ::= "(document " bytes ")" LF
lineage         ::= "(lineage " bytes ")" LF

profile         ::= "(profile " profile-id " " version " " constraints ")" LF
profile-id      ::= "full" | "read-only" | "lite" | "(custom " bytes ")"
constraints     ::= "(constraints " integer " " retention ")"
retention       ::= "(retention " integer " " option " " bool ")"

extension       ::= "(extension " bytes " " version " " bool
                  " (" chunk* ") " bytes " " bytes ")" LF
                  ; id, version, required, chunks, affected-kinds, barriers
                  ; (the ratified declaration order)

chunk           ::= "(chunk " chunk-kind " " schema " " bytes ")"
chunk-kind      ::= "operation-envelope-block" | "operation-index" | "snapshot"
                  | "blob" | "extension-data" | "text-projection"
                  | "layout-cache" | "integrity-index" | "manifest"
schema          ::= "(schema " integer " " integer ")"

canonical-base ::= "(canonical-base " bytes " " bytes " " integer
                  " " profile-id " " schema " " bytes ")" LF
                  ; snapshot-id, frontier, reduction version, profile,
                  ; root schema, root payload

blob           ::= "(blob " string " " option " " bytes ")" LF
                  ; media type, declared max uncompressed length, payload

envelope       ::= "(envelope " bytes " " bytes " " stamp " " causal
```

```

    " " option " " payload ")" LF
    ; id, author, stamp, causal context, transaction, payload
stamp      ::= "(stamp " integer " " integer " " bytes ")"
causal     ::= "(causal (" replica-seen* ") (" bytes* "))"
replica-seen ::= "(" bytes " " integer ")"

payload    ::= "(primitive " kind ")"
            | "(resolve-conflict " bytes " " action ")"
            | "(undo " bytes " " policy ")"
            | "(resolve-equivocation " bytes " " bytes ")"

action     ::= "accept-loser" | "keep-winner" | "dismiss"
            | "(override " bytes ")" | "(reanchor " bytes ")"
            | "(registered " bytes ")"

policy     ::= "strict-inverse" | "best-effort" | "cascade"

; --- Operation kinds inline their payload records as required by
; --- req:textproj:operation-vocabulary. Fields are the Operation Catalog's
; --- payload schema, positionally, in declaration order. Embedded Chapter-5
; --- values occur only at <value> positions and follow
; --- req:textproj:value-projection.

kind       ::= "(insert-event " bytes " " value ")"
            | "(delete-event " bytes " " tuple-comp ")"
            | "(respell-pitch " bytes " " value ")"
            | "(create-cross-cutting " cross-cutting ")"
            | "(change-region-time-model " bytes " " value
              " (" bytes* ") " remapping ")"
            | "(set-user-system-break " bytes " " value " " bool ")"
            | "(declare-transaction " bytes " " string " " option ")"
            | "(registered " bytes " " bytes ")"
            | "(modify-event " value ")"
            | "(transpose (" bytes* ") " integer ")"
            | "(insert-identified-pitch " bytes " " value ")"
            | "(delete-identified-pitch " bytes ")"
            | "(modify-identified-pitch " bytes " " value ")"
            | "(delete-cross-cutting " bytes ")"
            | "(modify-cross-cutting " cross-cutting ")"
            | "(create-region " value ")"
            | "(delete-region " bytes ")"
            | "(create-staff-instance " bytes " " value ")"
            | "(delete-staff-instance " bytes ")"
            | "(create-voice " bytes " " value ")"
            | "(delete-voice " bytes ")"
            | "(set-metadata " value ")"
            | "(set-metric-grid " bytes " " option ")"
            | "(set-user-page-break " bytes " " value " " bool ")"
            | "(create-staff " value ")"
            | "(set-time-signature " bytes " " value " " option ")"
            | "(set-tempo-segment " option " " value " " option ")"
            | "(set-staff-layout " bytes " " option " " option " " bool ")"
            | "(create-repeat-structure " value ")"
            | "(delete-repeat-structure " bytes ")"

```

```

| "(transpose-interval (" bytes* ") " value ")"
| "(create-instrument " value ")"

tuple-comp ::= "not-in-tuplet" | "(replace-with-rest " value ")"
| "(rewrite-tuplets (" bytes* "))"
| "(cascade-delete-tuplets (" bytes* "))"
cross-cutting ::= "(tie " value ")" | "(slur " value ")"
| "(beam " value ")" | "(spanner " value ")"
remapping ::= "preserve-time" | "(reassign (" reassign-entry* "))"
reassign-entry ::= "(" bytes " " ratio ")" ; event id, musical position

; --- Values and leaves.

; A struct, a sequence, a set, a map and an option all have this one shape. The
; reader tells them apart by the type it expects, never by the shape:
; req:textproj:schema-directed. Meaning is req:textproj:value-projection.
value ::= "(" value* ")"
| symbol | bytes | integer | bool | string | ratio | option
option ::= "()" | "(some " value ")"
ratio ::= "(ratio " integer " " integer ")"

bytes ::= "#x" hexdigit* ; even count, lowercase
integer ::= "-"? digit+ ; no leading zeros, no "-0"
bool ::= "true" | "false"
symbol ::= [a-z] [a-z0-9-]*
string ::= "'" schar* "'"
digit ::= [0-9]
hexdigit ::= [0-9a-f]
schar ::= unescaped | escape
unescaped ::= <any Unicode scalar value other than
U+0022, U+005C, U+000A, U+0009>
escape ::= U+005C U+0022 ; the two characters \"
| U+005C U+005C ; the two characters \\
| U+005C "n" ; the two characters \n
| U+005C "t" ; the two characters \t

```

Observe what does *not* appear: no offset, no compressed length, no compression algorithm, no chunk id, no content hash, no blob id. Every one is either physical or derivable, and Requirement 3.1 forbids both. The lone opaque identity the grammar carries is the `SnapshotId`.

Every production is expanded. Version 0.1.0 left *kind*, *action*, *policy*, *constraints*, and *barrier* derived-but-unwritten, and said so; that gap is closed. Barriers and affected object kinds are byte strings, because the bundle holds them opaquely and the projection interprets nothing the bundle does not.

At exactly the `value` positions delimited by Requirement 3.8, the one thing still *read* rather than restated is a Chapter-5 `value`'s field list. That is deliberate: Requirement 3.10 is a rule applied to the core specification's ratified listings, not a copy of them. A rule cannot drift from what it reads. An implementation that projects a `value`'s fields in an order other than the declaration order disagrees with the core specification, not with this document.

Note the operation-kind names are the *Operation Catalog*'s section names (`create-region`,

`create-staff`), not the `OperationKindTag` names (`InsertRegion`, `InsertStaff`). The tag space renamed three pairs for reasons of its own (`Binary Format`, `OperationKindTag`); the projection follows the semantics, not the tag.

A Worked Example

Non-normative. The byte strings below are illustrative but *well-formed*: every one is a grammar-valid `bytes` terminal, because an example that cannot be parsed teaches the wrong lesson. The conformance vectors that pin *real* bytes are a deliverable of the implementation.

A document of one operation — a transposition of two pitches up a perfect fifth, over a compacted base — projects to five lines:

```
(text-projection (0 8 0))
(document #x05050505050505050505050505050505)
(profile full (0 1 0) (constraints 67108864 (retention 1 () true)))
(canonical-base #x1f8b0000000000000000000000000000 #x 1 full (schema 0 1) #x0000)
(envelope #x000000000000000070000000000000001 #x00000000000000000000000011223344 (
  stamp 42 7 #x000000000000000070000000000000001) (causal ((#x00000000000000001 3)
) (#x000000000000000020000000000000009)) (some #
x000000000000000070000000000000005) (primitive (transpose-interval (#
x000000000000000070000000000000001 #x00000000000000007000000000000002) (
transpose-interval 4 7))))
```

The envelope's targets are a *set*: strictly increasing, no duplicates (Operation Catalog, `req:opcat:transpose-interval` Binary Format seq[†]). A parser **MUST** reject a duplicate rather than absorb it, exactly as the binary decoder does.

Revision History

Date	Section	Change
July 2026	24, All	0.1.0 — Initial companion. Supplies the canonical s-expression form the core specification’s Chapter 8 TEXT PROJECTION declares normative and leaves unwritten, and which the Binary Format companion excludes as “the <i>Text Projection</i> companion’s”. Ratified: reduced state is preserved by <i>determining</i> it, never by a second literal copy (req:textproj:reduced-state-derived); a canonical base snapshot is inlined as one opaque byte string, because a pruned document’s base is derivable from nothing and a reference-only projection would be lossy (req:textproj:base-snapshot-inline); lowercase hex is the single byte-string encoding (req:textproj:hex); one envelope per line, so a line-based merge conflict is exactly an envelope conflict (req:textproj:envelope-per-line). Parsing is strict-canonical (req:textproj:strict-parse) — normalizing non-canonical text is accepting it — and conformance requires <i>both</i> directions (req:textproj:conformance). No implementation yet; this document is the design gate.

Date	Section	Change
July 2026	24, Chapters 3, 5	0.2.0 — Canonical-manifest coverage. 0.1.0 was <i>lossy for documents that are valid today</i>, and its claim to preserve the manifest’s canonical roots was false in three ways: a canonical blob (an embedded image, font, or recording referenced by a canonical operation) had no representation at all; an <code>ExtensionDeclaration</code> lost its semantic version and its affected object kinds, and left its preserved chunk roots undefined; and <code>ProfileId::Custom</code> was unrepresentable, a symbol being required where a sixteen-byte registry id is carried. The three share one cause, now stated as <code>req:textproj:derive-or-carry</code>: a <code>ChunkRef</code> and a <code>BlobRef</code> are <i>physical</i> references — offset, compressed length, compression — which the projection may not preserve, and they also carry <i>derivable</i> identities, which it may not duplicate. Carry the content and the semantic attributes; re-derive the rest. The sole non-derivable identity in schema major 0 is <code>SnapshotId</code>, which the Binary Format companion pins as opaque and forbids readers to derive. Consequently: <code>req:textproj:canonical-blobs</code> (a canonical blob is projected; a non-canonical one is not), <code>req:textproj:profile-id</code> ((custom #x...)), <code>req:textproj:extension-declaration</code> (every field, chunks as kind + schema + payload), and <code>req:textproj:base-snapshot-inline</code> extended to say what the inlined payload <i>is</i> and how the root chunk id and the snapshot hash are re-derived rather than read. The core specification’s own list of what the projection preserves omitted canonical blobs; it is corrected there too. The 0.1.0 ratifications — reduced state derived, base inlined, hex, one envelope per line, strict parsing — stand unchanged.

Date	Section	Change
July 2026	24, Chapters 3, 5	0.3.0 — Every production expanded. 0.1.0 left <code>kind</code>, <code>action</code>, <code>policy</code>, <code>constraints</code> and <code>barrier</code> <code>derived-but-unwritten</code> and admitted it; all are now written. Barriers and affected object kinds are byte strings, because the bundle holds them opaquely and the projection interprets nothing the bundle does not. Operation payloads embed Chapter-5 canonical values, and those are projected by <i>one rule</i> rather than by forty productions (<code>req:textproj:value-projection</code>): a struct is (<code><type-name> <field>...</code>) with fields positional in the ratified declaration order; a newtype is transparent, as it is in the binary form; a tagged union is (<code><variant> <field>...</code>); an option is <code>()</code> or <code>(some v)</code>; a sequence keeps the binary form's order. Restating the Chapter-5 model here would have put two normative listings on one struct, which is the drift P13-I1 was opened to close — and a rule cannot drift from what it reads. Two leaf decisions follow from canonicity rather than taste. A rational is (<code>ratio n d</code>) in lowest terms with the sign on the numerator. A <code>CanonicalF64</code> is the byte string of its eight canonical IEEE 754 bytes, never a decimal: decimal float text is not canonically unique, so a decimal tempo would break <code>req:textproj:canonical-text</code> at the first tempo mark. Operation-kind names are the Operation Catalog's section names, not the <code>OperationKindTag</code> names, which renamed three pairs for reasons of the tag space. Still no implementation.

Date	Section	Change
July 2026	24, Chapters 3, 5	0.4.0 — Two normative corrections found in review. <i>The grammar contradicted its own escape requirement.</i> <code>req:textproj:string-escapes</code> obliges a writer to escape the backslash and a parser to reject a bare one, while <code>unescaped</code> admitted it. The escape productions are now spelled out as two-character sequences and <code>unescaped</code> excludes U+0022, U+005C, U+000A, and U+0009 by codepoint. Both characters of an escape are written as codepoints where they are the delimiter or the introducer: a quoted terminal for the backslash reads as <i>two</i> backslashes and would make every escape three characters long. For the same reason the mono font no longer applies TeX ligatures, which rendered U+0022 as a right curly quote and -- as an en dash — a document that specifies a text syntax must not misprint it. <i>“Keep the binary order” does not work for every sequence.</i> It is available only where the binary order reads preserved data, and two sequences fail that test: <code>blob_roots</code> is sorted by the full <code>BlobRef</code> encoding, which contains the offset, the compressed length, and the compression; and an extension’s preserved chunk roots are sorted by <code>ChunkRef</code>’s order, whose key is kind, then content hash, then <i>offset</i>. Under the blanket rule, relocating a chunk — which changes no semantics — would have changed the text, and two entries indistinguishable after erasure would have produced duplicate lines. <code>req:textproj:derived-ordering</code> orders and de-duplicates those two sequences by their <i>projected form</i>. Every other sequence keeps the binary order: the profile and extension declarations sort on semantic (<code>id</code>, <code>version</code>) keys, and the envelopes on canonical operation order. Also: a committed grammar-completeness test now checks that no nonterminal is undefined or unreachable, that the escape rule excludes the four codepoints and admits exactly the four two-character sequences, that the mono font substitutes no glyphs, and that the operation-kind and chunk-kind productions are exactly the tag vocabularies — derived from <code>OperationKindTag</code> and <code>ChunkKind</code>, not transcribed. Every locator in it finds its production by name; the checks it replaced were anchored to a column, and a reflow would have silently switched them off. The 0.3.0 claim of a “machine-checked” grammar was true of one run and of nothing durable. Still no implementation.

Date	Section	Change
July 2026	24, Chapters 3, 5	0.5.0 — Found by starting the implementation: the grammar could not derive an ordinary pitched note. <i>The projection is schema-directed</i> (<code>req:textproj:schema-directed</code>), and now says so. <code>value</code> had no alternative for a sequence at all, though clause 5 required one; and once added, a sequence whose first element is a fieldless variant is shape-identical to a struct, while <code>()</code> is both the empty sequence and the absent option. Both collisions are reachable from the first pitched note in any score — <code>PitchedEvent</code> carries articulations and ornaments over zero-field marks, beside an optional <code>DynamicMark</code>. The collision is irreducible without new syntax, and new syntax would buy nothing: <code>req:textproj:strict-parse</code> already obliges a parser to reject a duplicate in a set-typed field, which it cannot do without the schema. So <code>value</code> now collapses to <code>"(" value* ")"</code> or a leaf — all shape can honestly say — and the requirement assigns meaning by expected type. <i>Three consequences, stated.</i> A struct with no fields is the bare symbol, as a fieldless variant is. A byte string is not a sequence: where the binary form writes a length-prefixed run of bytes, so does the projection, never a list of integers. And the grammar's repetitions now carry a notation rule — adjacent elements are separated by exactly one space — without which <code>"(transpose (" bytes* ") "</code> spelled two targets as one run of hex. Still no implementation; this is what the first day of writing it found.

Date	Section	Change
July 2026	24, Chapters 3, 5, 6	0.6.0 — The operation vocabulary is grammar-directed, and the boundary is now normative (<code>req:textproj:operation-vocabulary</code>). The envelope, stamp, causal context, payload, operation kinds, and their sub-vocabularies follow the grammar’s productions; <code>req:textproj:value-projection</code> applies exactly at the <code>value</code> positions those productions name. Operation-kind payload records are inlined because they name variant types but add neither a binary wrapper nor a modelling distinction. <code>transpose-interval</code> now takes its target byte strings followed by one value. Its <code>TranspositionInterval</code> therefore has the same (<code>transposition-interval ...</code>) spelling it has at every other value position, and the grammar has no special <code>interval</code> production. The committed grammar gate locks the requirement, its reliance-point citations, and that absence.
July 2026	24, Chapters 3, 5	0.7.0 — Canonical blob rejection and single-version header gating. At this version no canonical operation or canonical reduced state can reference a <code>BlobId</code>; every (<code>blob ...</code>) line is therefore unreferenced and non-canonical, and a parser must reject it (<code>req:textproj:reject-unreferenced-blobs</code>). Accepting it would stage content the next projection silently drops, lose data, and falsify the text-to-binary-to-text round trip. Forward compatibility belongs to header-version gating, not lenient blob acceptance. The parser accepts exactly the implemented companion header, <code>(0 7 0)</code>, and rejects every other version (<code>req:textproj:header-version</code>). Multi-version acceptance and text migrate-on-read remain deferred, in the same posture as op-payload migrate-on-read: a future real consumer must bring an explicit, version-keyed migration path rather than teaching the current parser to speculate. The worked example now uses the implemented header and contains only grammar-valid, canonical lines.

Date	Section	Change
July 2026	24, Chapter 5	0.8.0 — The genesis operation vocabulary reaches the grammar. The kind production gains <code>"(create-instrument " value ")"</code>, the first operation kind appended since the header was gated to a single version at 0.7.0 (<code>req:textproj:operation-vocabulary</code>). The bump is forced rather than cosmetic. A parser MUST accept exactly the version of the companion it implements (<code>req:textproj:header-version</code>), so extending the grammar while holding the version would leave two mutually incompatible grammars both claiming <code>(0 7 0)</code> — an older parser meeting the new production would fail with no version signal to explain why, which is the precise failure the single-version gate exists to prevent. Cached projections at <code>(0 7 0)</code> do not migrate and are not expected to. A <code>TextProjection</code> chunk is a non-canonical accelerator a writer may discard, so a stale projection is regenerated from the canonical document rather than converted. Multi-version acceptance and text migrate-on-read remain deferred, unchanged.