Literium — Specification

The literium project

version 1.0-draft

About this document. Normative reference for implementers. An implementation is conformant if its observable behavior matches this document. Rationale, rejected alternatives, and the *why* behind each decision live in `HISTORY.md`, the internal decision log; back-references take the form `HISTORY §N`. When this document and `HISTORY.md` disagree on a normative point, this document wins and `HISTORY.md` is to be updated.
Status. v1 foundation. The v0 provisional layer crystallized on 2026-07-05 after two independent implementations realized it byte-identically (HISTORY §14); the remaining *(pending)* markers resolve to the open rows of §10.

Contents

0 Conventions

— MUST, SHOULD, MAY follow RFC 2119 [1] usage.
— All file content is UTF-8. Any other encoding is out of scope.

— Hashes are SHA-256, encoded as 64-character lowercase hexadecimal. Every hash is computed over the object's **raw bytes** — with the single version-object carve-out of §6.7/§8 (the version hash covers the hashed pre-image, excluding the manifest region). Storage framing (§6.8 — headers, compression, length prefixes) is never part of any hash pre-image.
— **No Unicode normalization is performed anywhere.** Content extraction, segmentation, and hashing operate on the writer's bytes as written; an é stored precomposed (NFC) and an é stored decomposed (NFD) are different byte sequences, hence different atoms with different hashes. Implementations MUST NOT normalize. (HISTORY §6 *No Unicode normalization, anywhere*.)
— **Unicode data is pinned.** Wherever this specification invokes a Unicode algorithm or property (UAX-29 [3] sentence boundaries, the `Sentence_Break` property), the reference is Unicode **16.0.0** [4]. See §4 for the repository-recorded segmentation version and the mismatch rule.
— Byte sequences in examples use `\n` for U+000A LINE FEED, `\r` for U+000D CARRIAGE RETURN, and U+xxxx for other code points.
— Where a concept has settled Latin or Greek register forms (see `TERMINOLOGY.md`), implementations SHOULD accept them as CLI aliases alongside the English working term (HISTORY §11).

1 Repository and modes

A document is in exactly one mode: **`prose`** or **`poetry`**. The default is `prose`.

1.1 Mode attribution

Mode is set per-file via a repo-root **attributes file**. The attributes file contains zero or more entries, one per non-blank, non-comment line:

```
<glob>  <mode>
```

— `<glob>` is a path glob relative to the repo root. Glob syntax and matching semantics MUST follow the `.gitignore` pattern rules of §1.2, except that `!` negation is not permitted — each line is a mapping, not an exclusion. Patterns containing whitespace use backslash escaping, per gitignore [2].
— `<mode>` is the literal string `prose` or `poetry`.
— Whitespace separates the columns; runs of space or tab are equivalent.
— A line whose first non-whitespace character is `#` is a comment.

The first matching entry wins. A file matched by no entry is `prose`. (First-match-wins deliberately inverts `.gitattributes`' last-match-wins: a two-column mapping reads top-down like a case statement, specific rules above general ones. HISTORY §3 *Attributes glob semantics*.)

The attributes file is itself versioned as a normal document in `prose` mode. This is an **override**, not a default: the attributes file (and every `.litignore`, §1.2) resolves to `prose` unconditionally, before any attributes entry is consulted — a matching glob, including a

catch-all like `* poetry`, MUST NOT re-mode a control file. (Attributes resolution is otherwise self-referential, and mode is identity-load-bearing; HISTORY §3 *Control files are always prose*.)

Mode is load-bearing for identity: it selects the segmentation pair (§4) and therefore participates in every atom, fasciculus, spina, and atom-spina hash. Two consequences are normative:

— **Historical mode resolution.** Any operation on a *historical* version — reconstruction, diff, `lit fsck` — MUST resolve modes from the attributes document **as it was in that version** (its identity trio is in the version object, §8), not from the working tree's current attributes file.

— **A mode change is a re-segmentation.** Editing the attributes file re-segments every affected document at the next conscribe, even when the document's bytes are unchanged; the resulting version pair classifies as a *re-segmentation* edit (§7.2) whenever any derived hash changes. (A mode change that happens to yield identical segmentation lands in the no-change row.)

The attributes file is **`.litattributes`** at the repository root — a dotfile sibling of `.lit/`, mirroring `.litignore`. (HISTORY §3, §14 *Attributes file*.)

1.2 Ignore patterns

A repo-root file named `.litignore` lists path patterns excluded from any operation that scans the working tree (`lit status`, `lit conscribe`, etc.). Pattern syntax and semantics MUST follow `.gitignore` verbatim — comments (`#`), blank lines, globs (`*`, `**`, `?`, `[...]`), trailing `/` for directory-only matches, leading `/` to anchor at the file's directory, and `!` for negation.

The `.lit/` directory is always implicitly ignored. `.litignore` files MAY appear in subdirectories; a subdirectory's patterns apply to that subtree. The `.litignore` file itself is versioned as a normal document in `prose` mode — unconditionally, overriding any matching attributes entry (§1.1).

> Low priority for v1. The decision to adopt `.gitignore` syntax/semantics verbatim is settled; full edge-case parity (interaction with mid-conscribe updates, force-add semantics, exact pattern grammar) MAY be deferred to v1.x. A minimal v1 implementation supporting basic globs, comments, and blank lines is sufficient.

1.3 Repository discovery and creation

Discovery. Every lit command MUST locate its repository by walking upward from the invocation directory (inclusive) to the nearest ancestor containing a `.lit/` directory. That ancestor is the repository root; all working-tree scans, `.litignore` anchoring, and attributes-file resolution are rooted there, regardless of which subdirectory the command runs in. `lit conscribe` takes the creation path below, or the descendant-scan dispatch of §1.4, **only when the upward walk finds no enclosing repository**. A conscribe inside an existing repository MUST NOT create a nested repository. (HISTORY §1 *Repository discovery*.)

Documents. A **document** is a working-tree file (after `.litignore`) whose bytes decode as valid UTF-8 **and** whose content form (§3–§4) is non-empty. Files that fail either test — binaries, invalid UTF-8, empty files, whitespace-only files — are not documents and are skipped by every scan. Symlinks are not documents and MUST NOT be followed by any working-tree or subtree scan (including §1.4's `.lit/` scan). (HISTORY §1.)

Creation. A literium repository is created by the first successful `lit conscribe` in a directory. There is no separate `lit init` command. (HISTORY §1 *Repository creation: no `lit init`.*) Creation MUST be refused if the working tree contains no documents. Creation writes `.lit/` and an empty historia file (§9.1) together.

On the first conscribe in a fresh directory, the implementation MUST prompt the user to confirm the default mode (`prose`) for the attributes file, and MUST write the attributes file before any version is recorded. The initial file content is `* <mode>` + LF. The mode MUST be selectable non-interactively via `lit conscribe --mode prose|poetry` (HISTORY §14 *Attributes file*).

1.4 Project scope: elevation

If `lit conscribe` is invoked in a directory that is not inside an existing repository (§1.3 discovery) and whose recursive subtree contains one or more `.lit/` directories — **proper descendants only**; an enclosing repository's own `.lit/` never triggers this dispatch — the implementation MUST scan the subtree, list every `.lit/` found, and prompt the user to confirm before proceeding.

- **Zero `.lit/` in the subtree** — ordinary first-conscribe path (§1.3).
- **One `.lit/` in the subtree** — the user MAY confirm **elevation** of the descendant project: its root moves up to the invocation directory. Elevation rewrites the manifest of every version object in place to be relative to the new root; version hashes are unchanged (§8). Elevation is one-way — there is no operation that descends a project's root.
- **Two or more `.lit/` in the subtree** — the user MAY confirm folding the descendants into a single project rooted at the invocation directory. This is **project merge**: the merge primitive of §9.4 applied to the descendants, with theca union (free, since content-addressed), the history fold of §9.4, and manifest reconciliation. Manifest conflicts (two documents claiming the same post-elevation path) MUST be resolved by a prompted re-path before the fold concludes.

Control files. A descendant's attributes file and `.litignore` travel as ordinary documents to their new subtree paths, but their *effect* is root-anchored, so elevation MUST fold their entries into the new root's control files with the descendant's root-relative prefix applied to each glob. The fold MUST preserve match depth under gitignore anchoring rules: an unanchored descendant rule `*.txt poetry` becomes `novella/**/*.txt poetry` (a prefixed glob contains a slash and is therefore anchored; `novella/*.txt` would narrow an any-depth rule to one level). The folded result is presented at the confirmation prompt. Conflicting entries are surfaced at the same prompt, first-match order chosen by the writer. (HISTORY §1.)

Implementations MUST NOT proceed without explicit user confirmation; implicit elevation is a footgun (an invocation from a deep ancestor — e.g., ~ — would otherwise fold every project the user owns).

2 Atoms

The **atom** is the smallest unit with content identity, an ordinal position, and edit participation.

Mode	Atom	Grex (instance)
prose	sentence	paragraph (blank-line-separated)
poetry	line	stanza (blank-line-separated)

The cross-mode abstract term for the grouping is **grex** (Latin/English; Greek *thiasos* θίασος) — a gathering of atoms between blank-line separators. *Paragraph* and *stanza* are mode-specific instance names. (HISTORY §11 *grex / thiasos*.)

In poetry mode, *line* and *stichos* (Greek στίχος, verse line) are interchangeable terms for the atom. *Line* is the primary working term; *stichos* its literary alias.

Atoms MUST NOT span grex boundaries: a single sentence MUST NOT cross a paragraph break, and a single line MUST NOT cross a stanza break. (Content extraction §3 makes this trivially true: blank lines define grex boundaries.)

3 Content extraction

Content extraction is applied **only** when computing hashes and diffs. The on-disk byte sequence MUST be preserved verbatim across read/write round-trips.

Extraction produces an intermediate **extracted text** from the verbatim bytes; segmentation (§4) then splits it into grex-instances and atoms; the **content form** of atoms and documents is defined from the segmentation output (§3.4).

3.1 Common rules (apply in both modes)

Apply in order:

1. `\r\n` → `\n`. Any remaining `\r` → `\n`.
2. A **line** is a maximal run of bytes between LFs (or between start/end of input and an LF). A **blank line** is a line whose content consists only of U+0020 SPACE and U+0009 TAB, or is empty.
3. Every blank-line run (one or more blank lines, whatever bytes they carry) is replaced by a single *empty* blank line, so that in the extracted text every grex separator is exactly the byte pair `\n\n`.
4. Leading blank lines and trailing blank lines are removed.
5. Non-ASCII whitespace (U+00A0 NBSP, U+2009 THIN SPACE, U+202F NARROW NO-BREAK SPACE, etc.) survives extraction verbatim. Implementations MUST NOT collapse, normalize, or remove non-ASCII whitespace during extraction. At segmentation

time, occurrences at **atom edges** are stripped (§4.3); non-ASCII whitespace therefore persists in content form only atom-interior — which is where its deliberate uses live (French punctuation spacing, number–unit, initials, typographic thin/figure spaces). (HISTORY §6 *Atom edges*.)

Note a consequence of rule 2: a line containing only non-ASCII whitespace (e.g. a lone NBSP) is **not** blank — it does not separate grex-instances, so the text above and below it forms one grex. Deterministic and accepted; writers who want a grex boundary use a genuinely blank line.

3.2 Prose-only rules

Within a grex (a run of non-blank lines):

6. Runs of `[U+0020 SPACE | U+0009 TAB | U+000A LF]` collapse to a single U+0020 SPACE.

A grex in extracted text is therefore a single line.

3.3 Poetry-only rules

Within a grex:

6. Line breaks (U+000A LF) between non-blank lines are preserved.
7. Within a single line, runs of `[U+0020 SPACE | U+0009 TAB]` collapse to a single U+0020 SPACE.

Per-line edge whitespace (including indentation) is handled by the atom-edge strip at segmentation time (§4.3), not here.

3.4 Content form

Content form is **defined by construction** from the segmentation output (§4):

— An **atom's content form** is its segment substring with the atom-edge strip applied (§4.3).
— A **document's content form** is the join: atoms within a grex joined with a single U+0020 SPACE (prose) or a single U+000A LF (poetry); grex-instances joined with a single blank line (`\n\n`); terminated with exactly one `\n`.

Because content form is defined as this join, the reconstruction invariant of §6.5 holds by construction. Note the join is deliberately not byte-faithful to the extracted text at atom boundaries: `One!Two.` (a zero-width UAX-29 boundary after the STerm) has content form `One! Two.` — the verbatim bytes live in the volumen, and only there. A document with zero grex-instances has no content form and is not a document (§1.3).

4 Segmentation

The segmentation layer is parameterized by two boundary functions, together called a **mode**:

- the **grex-boundary** function maps a document's extracted text to an ordered list of grex texts — v1: split at blank lines, in both modes (§3.1);
- the **atom-boundary** function maps one grex's text to an ordered list of atom substrings — v1 prose: UAX-29 sentence boundaries (§4.1); v1 poetry: U+000A LF (§4.2).

Implementations MUST expose this parametrization as a public extension point so that derived projects can supply their own pair of functions without forking the engine. Any boundary function MUST be deterministic and language-agnostic; heuristic or per-locale tailoring is rejected at the API layer (HISTORY §2 *Segmentation strictness / Library extensibility*). The atom-edge strip (§4.3) applies to every mode's output.

Segmentation version. Segmentation is pinned to Unicode **16.0.0** (§0). Every repository MUST record a **segmentation version** — a single identifier covering the UAX-29 revision, the UCD version, and any adopted refinements (§10 #10) — at **`.lit/segmentation`** (the identifier + LF; §6.9). The v1 identifier is `uax29-16.0.0`. Tooling whose own segmentation version differs from the repository's MUST NOT silently re-segment: it MUST at minimum warn, MUST refuse hash-producing operations, and MAY offer re-segmentation only as an explicit migration command *(pending — §10 #10)*. (HISTORY §2 *Segmentation is pinned to a Unicode version*. Implementation notes, non-normative: Perl's `\b{sb}` tracks the interpreter's bundled UCD — perl 5.42 ships Unicode 16.0.0; Rust's `unicode-segmentation` crate at exactly version 1.12.0 carries UAX-29 sentence-boundary tables at Unicode 16.0.0, freezable via an exact-version dependency pin. The two have been verified boundary-equivalent — HISTORY §2 *Implementation language: Rust core, Perl oracle*.)

4.1 Prose

Each grex's extracted text is segmented into sentences using the **UAX-29 sentence-boundary algorithm** (Unicode 16.0.0), applied without tailoring.

- No abbreviation lists.
- No dialogue heuristics.
- No language-specific rules.
- No fix-up passes that depend on lookup tables.

Implementations MUST NOT silently correct UAX-29 mis-segmentations of `Mr.`, `etc.`, dialogue boundaries, or similar. These are accepted as known limitations of the algorithm and are part of the specification by reference. (HISTORY §2 *Segmentation strictness*.)

A sentence's content form is the substring between two adjacent UAX-29 sentence boundaries, with the atom-edge strip (§4.3) applied.

4.2 Poetry

Each grex's extracted text is segmented at U+000A LF. Each non-empty line between LFs is one atom. No further segmentation is applied.

A line's content form is the substring between two adjacent LFs, with the atom-edge strip (§4.3) applied. Note this strips leading indentation (including non-ASCII indentation such as NBSP) from the *content form* only — the volumen preserves it, and an indentation change classifies as a formatting edit (§7.2).

4.3 Atom-edge strip

The content form of every atom, in every mode, strips leading and trailing characters whose UAX-29 `Sentence_Break` property (Unicode 16.0.0) is `Sp`, `Sep`, `CR`, or `LF`. This is a fixed Unicode property class — deterministic and language-agnostic — covering U+0020, TAB, NBSP, THIN SPACE, NARROW NO-BREAK SPACE and the other `Sp` spaces, the ASCII stragglers FF (U+000C) and VT (U+000B), and the `Sep` separators NEL (U+0085), LINE SEPARATOR (U+2028), and PARAGRAPH SEPARATOR (U+2029). Atom-interior occurrences are untouched (§3.1 rule 5).

An atom whose content form is empty after the strip (a segment consisting only of strippable characters) is not an atom; it contributes nothing to the fasciculus.

> Why: with a U+0020-only strip, boundary-adjacent NBSP/THIN/FF/LS made the §6.5 reconstruction MUST empirically unsatisfiable — `lit fsck` would have flagged well-formed repositories. The property-class strip plus content-form-by-construction (§3.4) closes the whole class. This is part of atom identity; it cannot change after repositories exist. (HISTORY §6 *Atom edges: boundary whitespace, and content form by construction.*)

5 Identity

- **Atom identity** is `sha256(content(atom))`, encoded per §0, where `content(atom)` is the atom's content form (§3.4, §4.3).
- **Atom position** is the pair `(grex_index, atom_index_within_grex)`, both zero-based, where `grex_index` counts grex-instances (paragraphs in prose, stanzas in poetry). The pair indexes into the spina (for the grex) and into the fasciculus at that grex (for the atom within it).
- **Edits** reference positions relative to a named **parent state** (a version hash). A position is meaningful only relative to its parent. (The first version of a document has no parent state and carries no edits — §8.5.)
- **Document identity** is `sha256(verbatim volumen bytes)` (§6.1). Two documents whose verbatim bytes differ are distinct documents, even if their content forms are equal. Byte-identical files at two working-tree paths are **one document** listed at two manifest paths (§8) — there is nothing else a second copy could be.

Derived objects — atoms, fasciculi, spina, atom-spina — are a function of *(volumen bytes, resolved mode, segmentation version)*, not of bytes alone (§1.1, §4).

In addition to document identity, two stored-object hashes are used:

— **Spina hash** is `sha256(spina bytes)` (§6.4). Equal between two versions of a document iff their fasciculus sequences are identical.
— **Atom-spina hash** is `sha256(atom-spina bytes)` (§6.6). Equal between two versions of a document iff their atom-hash sequences are identical.

Together, document identity, spina hash, and atom-spina hash form a document's **identity trio**. A version object carries one trio per document (§8); comparing a document's trios across two versions enables O(1) classification of edit categories (§7.2). All three are content-addresses of stored objects; version objects carry them as ordinary pointers.

No identifier is assigned to an atom. Lineage across versions ("this sentence used to read…") is reconstructed at display time, not stored — exactly, by atom-hash equality (§7.4), not by similarity heuristics.

6 Storage

Each document is represented as a set of content-addressed objects of five types; a sixth type, the **version object** (§8), sits above them, one per snapshot of the working tree. All six types share one type-keyed namespace, the **theca** (§6.7).

— **Volumen** — verbatim bytes of one document.
— **Atom** — content form of one atom.
— **Fasciculus** — ordered list of atom hashes for one grex (paragraph in prose, stanza in poetry).
— **Spina** — ordered list of fasciculus hashes for one document.
— **Atom-spina** — flat list of every atom hash of one document in document order (§6.6).
— **Version** — per-document identity trios + manifest for one working-tree snapshot (§8; carve-out in §6.7).

The per-document types (volumen, spina, atom-spina) are *per-document in role*; like every content-addressed object they are shared on hash collision — two versions of a document with an identical atom sequence share one atom-spina object, and byte-identical files share one volumen. Atoms and fasciculi additionally dedupe *across* documents by design.

Fasciculus names the storage object only; the corresponding abstract textual unit is *grex* — a paragraph in prose, a stanza in poetry. The two are not interchangeable: a fasciculus is a list of pointers; a grex is text in the volumen. (HISTORY §11.)

6.1 Volumen

The **volumen** is a single content-addressed object holding the **verbatim bytes** of the whole document — the file as written, byte-for-byte, with no content extraction applied. Its identity is `sha256` of its bytes; this is also the document's identity (§5).

6.2 Atom

An **atom** object holds the content form (§3, §4) of a single atom, encoded as bytes. Atoms are content-addressed by SHA-256 of those bytes (atom identity, §5) and stored at `theca/atom/<aa>/<hash>` (§6.7). Atom objects are deduplicated repository-wide: two atoms with identical content form share one object.

Atomotheca is a colloquial label for the atom-typed slice of the theca (HISTORY §5 *Storage namespace*); it is not a separate store.

6.3 Fasciculus

A **fasciculus** object represents the atom membership of one grex (paragraph in prose, stanza in poetry): an ordered list of atom hashes. Fasciculi are content-addressed by SHA-256 of their bytes and stored at `theca/fasciculus/<aa>/<hash>` (§6.7).

Fasciculus grammar (informal, ABNF-like):

```
fasciculus = hash *( LF hash ) LF
hash       = 64( "0"-"9" / "a"-"f" )
LF         = U+000A
```

Concretely: one atom hash per line, in grex order, terminated by a single LF. A fasciculus MUST contain at least one atom hash; an empty grex has no fasciculus and no spina entry. (A document with zero grex-instances is not a document at all — §1.3; the delete-last-atom edit path is defined in §7.1.)

Fasciculus objects are deduplicated repository-wide: two grex-instances (paragraphs, stanzas, or one of each) with identical atom-hash sequences share one fasciculus object.

Fasciculotheca is a colloquial label for the fasciculus-typed slice of the theca (HISTORY §5 *Storage namespace*); it is not a separate store.

6.4 Spina

The **spina** is a per-document object: an ordered list of fasciculus hashes.

Spina grammar (informal, ABNF-like):

```
spina = hash *( LF hash ) LF
hash  = 64( "0"-"9" / "a"-"f" )
LF    = U+000A
```

Concretely: one fasciculus hash per line, in document order, terminated by a single LF.

Example (a document of three grex-instances):

```
…
fasc111…
fasc222…
fasc333
```

Each `…fascNNN` resolves in the fasciculotheca to a fasciculus listing that grex's atom hashes. The spina is itself stored content-addressed (`sha256` of the spina bytes); this is the *spina hash* of §5.

6.5 Reconstruction invariant

For every document, the content form of the document (§3.4) MUST equal the byte sequence produced by:

1. Walking the spina in order to obtain a sequence of fasciculus hashes.
2. For each fasciculus hash, dereferencing in the fasciculotheca to obtain a sequence of atom hashes.
3. For each atom hash, dereferencing in the atomotheca to obtain the atom's content bytes.
4. Joining atoms within a grex with a single U+0020 SPACE (in prose) or U+000A LF (in poetry); joining grex-instances with a single blank line (`\n\n`); terminating with a single LF.

Since §3.4 *defines* the content form as this join, the invariant holds by construction; it is stated so that `lit fsck` (§10 #7) has its verification target. The substantive check for fsck is **re-derivation**: given a version's volumen, its resolved mode (from the attributes document *at that version*, §1.1), and the repository's segmentation version, the derived spina, fasciculi, and atoms MUST equal the stored objects.

This invariant relates the tree (spina + fasciculotheca + atomotheca) to the *content form* of the document, not to its verbatim form. The verbatim form is preserved only in the volumen and is not byte-for-byte derivable from the tree.

6.6 Atom-spina

The **atom-spina** is a per-document content-addressed object: the flat sequence of every atom hash in document order, equivalent to concatenating each fasciculus's atom-hash list in spina order.

Atom-spina grammar (informal, ABNF-like):

```
atom-spina = hash *( LF hash ) LF
hash       = 64( "0"-"9" / "a"-"f" )
LF         = U+000A
```

Concretely: one atom hash per line, in document order, terminated by a single LF.

The atom-spina is informationally redundant with `walk(spina, fasciculotheca)`; its bytes can be derived. It is stored anyway for two normative roles:

— **Edit-category classification (§7).** `sha256(atom-spina)` is the third hash of a document's identity trio (§5). Comparing it across two versions reduces "did atoms change" to a single hash equality.

— **Ordinal-atom addressing.** The atom-spina is the stable, deterministic ordinal index over a document's atoms used by citation primitives (e.g., resolving "*Tom Sawyer* atom #123" at version V to a specific atom hash). Implementations SHOULD expose ordinal-atom addressing in their CLI / API surface; the form is the `aK` coordinate of §7.4, resolved by `lit show` (§8.11).

Two consequences worth naming explicitly:

— A reconstruction invariant complementing §6.5: the atom-spina MUST be byte-identical to the byte sequence produced by walking `spina → fasciculotheca` and concatenating each fasciculus's atom-hash bytes. Implementations MUST verify this in `lit fsck` (§10 #7).
— Atom-spina objects are content-addressed; two versions whose atom sequences are identical share a single atom-spina object in storage.

6.7 Theca and corpus

All six object types (§6) share one content-addressed namespace, keyed by `<type>/<sha256>`. Locally this namespace is the **theca**, rooted at `.lit/theca/`. Each object MUST be stored at:

`.lit/theca/<type>/<aa>/<hash>`

where:

— `<type>` is one of `atom`, `fasciculus`, `volumen`, `spina`, `atom-spina`, `version`;
— `<hash>` is the lowercase 64-character SHA-256 digest of the object's bytes (§0) — with the one carve-out below;
— `<aa>` is the first two characters of `<hash>`, used as a sharding prefix to keep any single directory bounded.

Version-object carve-out. For type `version`, `<hash>` is the version hash: the digest of the version object's **hashed pre-image** (the per-document identity trios plus other identity-bearing metadata, §8), which excludes the manifest region. The manifest region is the **sole** mutable-in-place region anywhere in the theca — elevation (§1.4), project merge (§9.4), and rename re-inscription (§8.5) rewrite it without the object's address changing. `lit fsck` verifies a version object's hash against the hashed pre-image and checks the manifest structurally, not by digest. (HISTORY §5 *Version objects live in the theca, with a carve-out*.)

When **atoms** are published to a Litorium **corpus** (a platform-layer surface; publication is an explicit, consent-gated act — PROTOCOL §7.6), they are addressable at:

`corpus.litorium.org/atom/<hash>`

The corpus holds atoms only. Non-atom object types (volumen, fasciculus, spina, atom-spina, version) MUST NOT be exposed on the corpus; they live in per-project storage at the platform layer, accessed through per-project URLs with platform-level access controls and TOS-bound publisher responsibility. The corpus is not a remote mirror of the theca but a distinct public surface — a sentence-level citation/dedup primitive. Rationale (copyright posture; dedup-utility): HISTORY §5 *Storage namespace: theca and corpus — Why atoms only*.

The URL surface MUST NOT include a shard prefix; backend storage MAY shard internally for operational scale. Companion endpoints under the same `corpus.litorium.org/<type>/` shape (e.g., `/meta/<hash>`, `/rev/<hash>`) are anticipated but unspecified in v1.

Atomotheca and *fasciculotheca* are colloquial labels for the atom-typed and fasciculus-typed slices of the local theca; they are not separate stores. (HISTORY §5 *Storage namespace*.)

6.8 Object framing and packs

There is **no object framing**: a stored object's file content is exactly the object's raw bytes — no header, no compression, no length prefix. This keeps every v1 repository auditable with nothing but a SHA-256 tool. Framing bytes, should any future storage form introduce them, are outside every hash pre-image (§0), so identities cannot be disturbed retroactively. (HISTORY §14 *Object framing*.)

Local storage stores **loose objects only**, permanently; there is no literium pack format and none is pending. Bulk transfer of objects is a storage-substrate concern resolved below this specification's surface: remote cellae batch and compact deposita into substrate-level packs (*sarcinae* — the storage substrate's pack specification, consumed by the platform sync layer), which is identity-safe precisely because framing lives outside every hash pre-image (§0). Rationale and rejected alternatives: HISTORY §5 *Packs resolved out of the core*.

6.9 Repository layout

The `.lit/` directory contains exactly:

```
.lit/format                repository format marker: "1" + LF
.lit/segmentation          segmentation-version record (§4): identifier + LF
.lit/historia              the historia file (§9.1, §9.3)
.lit/experimenta/<name>    one file per experiment, same shape (§9.2, §9.3)
.lit/tabula                the tabula (§8.1); an absent file is the empty
   tabula
.lit/contextio             the combining-merge variant ledger — the
   apparatus (§9.6); absent = none
.lit/theca/                the object store (§6.7)
```

Tooling MUST refuse repositories whose format marker it does not support. Pending-revision state is **never stored**: the pending set is a pure function of (target-history tip, tabula) and is recomputed on demand (§8.8). (HISTORY §14 *Repository on-disk layout*.)

6.10 Garbage collection

`lit gc [--dry-run]` deletes unreachable objects. An object is **reachable** iff it is named — directly or through a version object's trios — by at least one inscriptio in any existing history, by the live tabula (§8.1), or by the contextio ledger's atom hashes (§9.6). The mark closure follows storage pointers only: version/tabula → identity trios (volumen, spina, atom-spina) → spina → fasciculi → atoms; the sweep removes every stored object of every type (§6.7) the closure did not visit. (HISTORY §5 *Garbage collection designed*.)

Collection MUST be deterministic (two conformant implementations remove exactly the same set) and MUST NOT alter any reachable object or identity. The report names per-type counts in §6.7's type order; `--dry-run` previews without deleting. Garbage arises from deleted experiments (§9.2) and from replaced or discarded tabula snapshots (§8.2's immediate object writes).

7 Edits

7.1 Operations

The edit operation set is closed.

Atom-level (operate on a single atom hash within a fasciculus):

- `insert(grex_index, gap_index, atom_hash)` — `gap_index` ranges 0 through the grex's atom count: before the first atom, between any two, or after the last.
- `delete(position)` — `position` is the `(grex_index, atom_index)` pair of §5.
- `modify(position, new_atom_hash)` — same position semantics.

Each atom-level op causes its containing grex's fasciculus to be replaced by a new fasciculus reflecting the change; the spina entry for that grex is rewritten to point at the new fasciculus. **Deleting a grex's last atom removes the grex**: no new fasciculus is created and the grex's spina entry is deleted (§6.3). A `delete` whose result would be a document with zero grex-instances is invalid — that state is a document *removal* (§7.2, §8.5), not a version of the document.

Structural (operate on fasciculi; do not touch any atom hash):

- `split-paragraph(grex_index, gap_index)` / `merge-paragraph(grex_index)` (prose mode)
- `split-stanza(grex_index, gap_index)` / `merge-stanza(grex_index)` (poetry mode)

A split partitions one fasciculus's atom-hash list into two new fasciculi at `gap_index` (an *interior* gap: 1 through atom count − 1); the spina entry is replaced by two entries pointing at the new fasciculi. `merge-paragraph(i)` concatenates the fasciculi of grex *i* and grex *i+1* into one new fasciculus; the two adjacent spina entries are replaced by one. No atom hash is added, removed, or changed by a structural op.

There is no `move` operation: a move is reconstructed at diff-display time from a `delete` and a matching `insert` of the same atom hash, possibly across versions. There is no `split-sentence` / `merge-sentence` operation in v1: these are recorded as `delete` plus two `inserts` (split) or its inverse (merge). Whole-grex deletion and new-grex creation are likewise composites (n deletes; insert-into-neighbor + split). In every case the editorial relationship is reconstructable at display time. (HISTORY §7.)

There is no first-class operation for **formatting changes** (changes to the verbatim volumen that leave spina, fasciculi, and atoms unchanged). A formatting change is observable as a difference between two volumens with no corresponding change at any other layer; it requires no entry in the edit op set.

An edit carries a **parent-state reference** (a version hash). All positions in the edit are interpreted in that parent state. The first version of a document has no parent state and carries no edits (§8.5).

7.2 Edit categories

Edit-category classification is **per document** within a version pair: each document that appears in both versions classifies by comparing its identity trio (§5) across the two.

`sha256(volumen)`	`sha256(spina)`	`sha256(atom-spina)`	category
equal	equal	equal	(no change)
differ	equal	equal	**formatting**
differ	differ	equal	**structural**
differ	differ	differ	**substantial** (may also include structural and/or formatting)
equal	differ	differ	**re-segmentation** (mode change, §1.1)

(In the substantial row the spina necessarily differs: the atom-spina is the concatenation of the fasciculi the spina lists, so it cannot change while the spina stays fixed. *Structural* covers exactly the order-preserving regroupings — split, merge; an atom moved between grex-instances reorders the flat sequence and therefore classifies substantial, with diff display presenting the matching delete/insert as a move. *Re-segmentation* — same bytes, different reading — arises only from an attributes-file change in v1; HISTORY §7.)

Documents present in only one of the two versions are **manifest-level changes** — *document added*, *document removed* — as is a *document rename* (same trio, different manifest path; §8.5). Manifest-level changes are not edit categories; they are reported alongside them. Note the rename's observability scope: (same trio, different path) is visible when classifying *pending* changes — working tree against a stored version — but not between two historical inscriptiones, because a pure rename re-inscribes one version whose single stored manifest was updated in place; there the pairing is recoverable only from the inscriptio message (§8.5).

A version MAY contain edits of multiple categories. Classification is per-version-pair, not encoded in the version object. Implementations SHOULD surface the category — and detected moves and atom splits/merges — in diff display (§10 #8).

Note: the derived objects are a function of (bytes, mode, segmentation version), so **for a fixed mode, equal volumen implies all three hashes equal** — a document untouched between two versions lands in the "(no change)" row, and no (equal, differ) combination other than re-segmentation is reachable. Rows where the volumen differs but the spina is equal while the atom-spina differs are impossible by construction (§6.6).

7.3 Wire format

Edit-op encodings remain *(pending — §10 #4 residue)*: v1 stores no edits — versions are snapshots (§8) and revisions are recomputed, never persisted (§6.9) — so `insert`/`delete`/`modify` and the structural ops have no v1 serialization. Op semantics are fixed by §7.1; the encodings become real with v2+ transport. The v1 serializations that *do* exist — version object, inscriptio, tabula — are specified in §8.10, §9.3, and §8.1. (HISTORY §14 *Serializations*.)

7.4 Diff presentation — the collation

A diff is a **collation** (*collatio*): a typed comparison of two witnesses of a text, every difference sorted into the classical categories textual critics have always used — **transpositio** (a move: the same atom or grex at a new position), **additio** (an atom present on one side only), **omissio** (an atom dropped), **substitutio** (an atom whose content changed — a different hash in the same aligned slot), and the grex-structure pair **divisio / coniunctio** (a paragraph split or merged with its atoms unchanged). These are exactly what literium detects by hash equality (below); the classification is deterministic, never similarity-scored (HISTORY §17). Output is plain deterministic bytes (no color, no locale dependence): the byte grammar of a given rendering (below) is part of the conformance surface — two implementations MUST render the same comparison identically. (HISTORY §7 *Diff presentation*, §17 *The collation*.)

Categories from the mechanics. The alignment and annotations specified in this section *are* the categories: the LCS-anchored replace runs (§8.8) with atoms on both sides are **substitutiones**; a run empty on one side is an **additio** or **omissio**; the exact-hash move match is a **transpositio**; the ~ regroup lines are **divisio** (1→N) and **coniunctio** (M→1); the atom split/merge annotations refine a substitutio where one atom is the whitespace-join of *k*. Within a substitutio, an implementation MAY descend to a **word-level collation** — the same categories over UAX-29 *word* boundaries at the §4 Unicode pin, a view-time segmentation that stores and hashes nothing — to show which words moved, dropped, or changed (HISTORY §17 *recursive descent*).

Renderings. One collation, several renderings, selected by flag; each is byte-deterministic. The default is the **apparatus** form — a clean reading text (the lemma side) with the variants recorded beneath, keyed by coordinate in the classical critical-apparatus grammar `<locus> <lemma> ] <reading> <siglum> : <reading> <siglum>`, where the *siglum* names the hand carrying each reading (a history name or a collaborator handle — provenance, §9.5). `--leiden` renders inline with the Leiden ASCII editorial brackets (^ insertion caret, `[ ]` a supplied/variant span, `{ }` an editorial deletion). A marks tier uses the Unicode critical-edition marks (Supplemental Punctuation U+2E00–) where the font carries them. **`--unified`** is the legacy git-style -/+ form specified in the remainder of this section — retained, but no longer the default. The exact byte grammar of the apparatus and Leiden renderings is *(pending — §10)* — the apparatus output implementations emit today is v1-provisional until that grammar is frozen; the `--unified` grammar below is normative and stable.

The remainder of this section specifies the `--unified` rendering (atoms in place of lines), whose alignment and annotation machinery is the shared engine all renderings draw from.

Coordinates. Display coordinates are 1-based: `gN` is the Nth grex of a document, `gNaM` the Mth atom of the Nth grex, `aK` the Kth atom in flat document order (the atom-spina ordinal — §6.6's citation primitive). Internal positions (§5) stay zero-based. The same coordinate labels are used by `lit status`, `lit diff`, `lit show` (§8.11), and the contextio ledger (§9.6).

Document order and headers. Documents render in the §8.5 ordering: paired paths (bytewise), then renames (`old -> new: renamed`), removals (`path: removed`), additions (`path: added`). A paired document with no change renders nothing. A changed one opens with `path: <category>` (§7.2), then by category:

— **formatting** — `path: formatting (<ranges>)` where `<ranges>` lists the touched grex ordinals (§8.6 verbatim ranges differing), consecutive runs collapsed: `(g2, g5-g7)`. No body.
— **re-segmentation** — the header line only.
— **structural** — one `~` line per regroup cell, old-side ordinals left of the arrow, new-side right: `~ g3 split -> g3,g4` (1→N), `~ g3+g4 merged -> g3` (M→1), `~ g2+g3 regrouped -> g2,g3,g4` (M→N). No `-`/`+` lines: the flat atom sequence is provably unchanged.
— **substantial** — hunks (below), with regroup cells still rendered as `~` lines in position.

Structural and substantial documents append one trailing `formatting: <ranges>` line when touched-but-hash-equal anchors exist alongside the real changes.

Hunks. The two sides align per §8.8 (LCS anchors, Rule A/Rule B cells). A hunk is a maximal run of consecutive pair/deletion/insertion cells; its context is the whole grex on each side (paragraphs are short; there is no context flag). The header is `@@ -<old grexes> +<new grexes> @@` with each side a single ordinal (`g3`), a range (`g3..g5`), or `.` when the side is empty (pure insertion/deletion of grexes). Body lines are one atom each: a two-character sigil column (`- `, `+ `, or two spaces for context), the atom's own side's `gNaM` coordinate (context atoms show the new side's), two spaces, the atom's content form (LF-free by construction, so one atom is always one line). Within a pair cell, atom-level `-`/`+`/context comes from the LCS over the two atom-hash sequences with the §8.8 leftmost trace.

Exact annotations (appended to the line, space-separated, bracketed; each names the counterpart location on the *other* side):

— **Moves.** After all cells are rendered, deleted and inserted atoms with equal hashes are matched in order (first unmatched deletion to the first identical-hash unmatched insertion, document-wide — moves cross grexes). The deletion gains `[-> gXaY]` (its new location), the insertion `[<- gXaY]` (its old location). Detection is exact hash equality; implementations MUST NOT use similarity scoring.
— **Atom splits and merges.** Within each replace run (the deletions-then-insertions block the leftmost trace emits), scanning left to right: a deleted atom whose content form equals the **single-space join** of k ≥ 2 consecutive not-yet-consumed inserted atoms is a **split** (content forms collapse interior whitespace to single spaces, and a split inserts a boundary at whitespace — so the words-preserved test is the space join in every mode; in poetry this detects line splits exactly) — the deletion gains `[split into gXaY,gXaZ,...]`, each insertion `[from gNaM]`; the mirror case (k ≥ 2 consecutive deletions joining to one insertion) is a **merge** — deletions gain `[into gXaY]`, the insertion `[merged from gNaM,gNaM',...]`. Equality is exact by content-form construction; partial overlap stays plain `-`/`+` (§8.8's accepted coarseness).

— **Precedence.** Move matching runs first; move-annotated atoms are skipped by split/merge detection.

8 Versions

A **version** is a saved snapshot of the **working tree**; lit's analogue of git's *commit*. Producing a version splits into two steps: `lit conscribe` snapshots the working tree onto a **tabula** (§8.1), and `lit inscribe` finalizes the tabula (or any named version) onto a **history** as an **inscriptio** (§8.3, §8.4, §9). A third verb, `lit exscribe` (§8.9), copies a version back out into the working tree. (HISTORY §8.)

A version object MUST carry, **for each document present at conscribe time**, that document's identity trio (§5): the document identity (`sha256` of the verbatim volumen), the spina hash, and the atom-spina hash. All trio members are content-addresses of stored objects, so version metadata holds them as ordinary pointers; comparing a document's trios across two versions enables O(1) edit-category classification (§7.2) without dereferencing any tree object. A single-document project's version object carries exactly one trio. A version MAY carry zero trios only when it records the removal of a repository's last remaining documents (§8.2). (HISTORY §8 *One version, whole tree*.)

A version object MUST NOT carry a parent-version reference. History order is held in histories (§9), not in the version object. This keeps version-object hashes content-stable across history operations and preserves a privacy property: shipping one version's objects to a recipient does not expose ancestor versions. (HISTORY §8 *Version objects don't chain to parents*.)

A version object additionally carries a **manifest** — a record binding each working-tree path to one document identity at conscribe time. The manifest is part of the version object's data but MUST NOT contribute to the version hash. The **version hash** is `sha256` of a serialization (the *hashed pre-image*) that covers the identity trios (§5) and other identity-bearing metadata; the manifest bytes MUST be excluded from that pre-image. Two version objects with equal hash MAY differ in their manifest; both are valid representations of the same version. Version objects are stored at `theca/version/<aa>/<versionhash>` (§6.7). (HISTORY §8 *Manifest is in the version object, excluded from the hash*.)

The manifest is mutated in place by elevation (§1.4), project merge (§9.4), and rename re-inscription (§8.5): these operations rewrite paths without disturbing version hashes. Across machines (v2+), two clones MAY hold "the same version" (equal hash) with different manifests, since workspace layout is not part of content state.

The inscription message and timestamp are not part of the version object — they live in the inscriptio (§8.4) that names the version in a history. The serialization and the hashed/unhashed field split are specified in §8.10; the **segmentation version is part of the hashed pre-image** (derived objects are a function of it — two snapshots differing only in segmentation version are different versions). Additional version-object fields (author, etc.) are *(pending — §10 #12)* and would sit outside the pre-image.

8.1 Tabula

The **tabula** is a single-slot scratchpad holding the prepared next version. An implementation MUST hold at most one tabula state per repository. A successful `lit conscribe` (§8.2)

MUST replace any existing tabula state with the new working-tree state — there is no append, stack, or named-buffer model.

The tabula holds a full working-tree snapshot, not a selection. There is no per-file or per-hunk staging vocabulary.

An implementation MUST provide an operation that **discards** the tabula without inscribing it — a writer who conscribes and then abandons the draft (restoring the working tree by hand or via exscribe, which leaves the tabula untouched) must not be left with a stale snapshot that conscribe's no-op refusal (§8.2) can never replace and that a later bare `lit inscribe` would silently offer up. The CLI surface is **`lit tabula`** (inspect) and **`lit tabula discard`** (discard). (HISTORY §14 *Tabula lifecycle*.)

The tabula is stored at **`.lit/tabula`** (§6.9), serialized in the version-object text form of §8.10 but unhashed — the tabula is a scratchpad, not a stored version. `lit conscribe` writes all derived objects into the theca immediately; the tabula file carries only the trios + manifest record of the snapshot. An absent file is the empty tabula. (HISTORY §14 *Serializations*.)

8.2 Conscribe

`lit conscribe` snapshots the current working-tree state onto the tabula (§8.1).

In a directory with no enclosing repository, conscribe takes the creation path (§1.3) or elevation dispatch (§1.4). Inside a repository, `lit conscribe` MUST refuse if either:

— There are **no pending changes**: the working tree's document set, manifest (paths), and every document's bytes are identical to the most recent version on the historia. Additions, deletions, and renames are pending changes even when every present document's bytes are unchanged. (A conscribe that records the deletion of the repository's last documents is valid; the resulting version carries zero trios.)

— The tabula already holds a state byte-identical to the current working-tree state.

The comparison baseline is the latest version on the **historia**; there is no current-history pointer (§9). If the historia is empty (first conscribe already happened but nothing is inscribed), the baseline is the empty document set, and any working tree with documents differs from it.

If the tabula was non-empty before a successful `lit conscribe`, the previous tabula state is discarded. Implementations MAY surface this in the CLI for confirmation but MUST NOT persist the prior tabula content.

8.3 Inscribe

`lit inscribe` takes a source — the tabula by default, a named version, or an experiment's latest version — and appends it as an **inscriptio** (§8.4) to a target history.

CLI signature stack:

Form	Source	Target
`lit inscribe`	tabula	historia
`lit inscribe <target>`	tabula	named target (historia or experiment)
`lit inscribe --fons <source>`	named source	historia
`lit inscribe <target> --fons <source>`	named source	named target

The `--fons` flag (English alias `--from`; short form `-f`) accepts a version hash or a history name. Resolution is syntactic: a 64-character lowercase-hexadecimal argument is a version hash; the reserved name `historia` resolves to the historia's latest inscribed version; anything else is an experiment name, resolving to the experiment's latest inscribed version. Experiment names MUST NOT be exactly 64 lowercase hexadecimal characters (§9.2). Naming a history with zero inscriptiones is an error. (HISTORY §8 *Source flag: `--fons`*.)

The target argument — when present — MUST resolve to an existing history: either *historia* (always present), or an existing experiment by name. Implementations MUST NOT create an experiment as a side effect of `lit inscribe`; experiment creation is a separate operation (§9.2).

Implementations MUST display source and target in the inscription-message prompt header before accepting the message body, so accidental historia inscriptions are caught at the moment of commitment. The header form is `inscribe: <source> -> <target>` (HISTORY §14 *Inscription prompt header*).

Implementations MAY additionally accept short version-hash prefixes (six or more hex characters, unambiguous within the theca) wherever a version hash is accepted — with the rule that an **existing history name wins over the prefix reading**: resolution stays syntactic, so a legal hex-shaped experiment name resolves as the experiment, never as a prefix. (HISTORY §14 *Source resolution conveniences*.)

After an inscription whose source is the tabula, the tabula MUST be cleared **when the inscription consumed the last pending revision** (`--all`, the editor form, or the final `-<n>`); a partial `-<n>` inscription MUST remove only the inscribed revision from the pending set and leave the tabula otherwise intact (§8.6). After a successful inscription whose source is a named version or an experiment, the tabula state is unchanged.

`lit inscribe` subsumes git's `commit`, `merge`, `cherry-pick`, and `revert` — with one deliberate semantic difference: `--fons` **adopts a whole snapshot**; it never applies a delta onto the target's latest state. (Git's cherry-pick constructs and applies a patch; lit's names an existing state, wholesale. A writer who wants one revision from an experiment without its other changes exscribes and edits, or merges — §9.4/§9.6.) The merge-vs-direct-inscription distinction is recoverable from the multiset of inscriptiones across histories for adoption merges (§9.5), not encoded as a structural variant of the operation. (HISTORY §8 *What `inscribe` collapses*.)

8.4 Inscriptio

An **inscriptio** is one entry in a history (§9): the per-event record produced by an `inscribe` call. An inscriptio MUST carry:

— the **version hash** being inscribed (a content-address of a version object);

— a **timestamp** of the inscription event;
— a **message** (free-form text the writer attaches; per-revision and `--all` rules in §8.6).

A history's order is its **entry order** — the file's sequence, not the timestamps. Timestamps are provenance data (§9.5) and need not be unique. Timestamp and message encoding are specified in §9.3; additional fields (author, etc.) are *(pending — §10 #12)*.

8.5 Revisions

A **revision** is the set of all pending edits within a single grex: **at most one edit-bearing revision per grex** (paragraph in prose, stanza in poetry), and a revision's *edits* never cross grex boundaries. (v1 simplification; HISTORY §8 *Revisions, sharpened*. The non-edit-bearing revision kinds below are the defined exceptions to the one-grex scope.)

Four further kinds of pending change are each one revision, so that every pending difference is inscribable:

— A **formatting-only delta** (volumen bytes changed; spina, fasciculi, atoms unchanged) is one revision, of category formatting, per contiguous run of touched grex ranges — a run MAY span multiple grex-instances (e.g. a whole-document re-wrap is one revision). A delta confined to separator bytes belongs to the grex whose range contains them (§8.6's range rule).
— A **rename** (manifest-only delta) is one manifest revision. Inscribing it updates the stored version object's manifest in place and appends a re-inscription of the **same version hash** to the target history (re-inscription is ordinary — §9.5); the message records the old → new pairing. Note the observability limit: a pure rename is detectable as (same trio, different path) only in *pending* classification — working tree against a stored version. Between two historical inscriptiones it appears as a re-inscription of one version, its pairing recoverable only from the inscriptio message (§7.2, HISTORY §8).
— A **document removal** is one revision: the version it produces omits the document's trio and manifest entry. (A removal is a pending change per §8.2; without a revision kind it would be uninscribable in the default mode.)
— The **first version of a document** — a document addition; no parent state — is one revision covering the whole document.

Ordering. Pending revisions are ordered by the document's manifest path (the *former* path, for removals and renames), then by grex order within a document. This is the "document order" of §8.6/§8.7 and the index order of `lit status` (§10 #13).

Parent binding. The parent version — for revision decomposition, cumulative construction, and splicing (§8.6) — is the **latest version on the target history**. `lit status` reports pending revisions against a named history, historia by default (§10 #13). (The §8.2 conscribe refusal baseline remains historia; it is only a no-op guard.)

Decomposing (parent version, tabula) into revisions uses the deterministic alignment algorithm of §8.8.

8.6 Inscription modes

— **Default — per-revision inscription.** Each pending revision becomes its own version, with its own message. Per-revision versions are **cumulative**: each inscription applies its revision to the previously inscribed state (the first to the parent of §8.5), so inscribing all pending revisions one by one, in document order, is equivalent to the editor form, and the final version is byte-identical to the tabula state.
Intermediate volumen construction (splice rule). Inscribing revision *n* alone produces a version whose bytes never existed on disk. Its volumen MUST be constructed by grex-splicing over **verbatim grex ranges**, defined so that every byte of a volumen belongs to exactly one range: line ends and blank lines are recognized per §3.1 rules 1–2 *for delimiting only* (the spliced bytes themselves are never altered); a grex's range runs from its first byte through the last byte before the next grex's first byte — i.e. it **includes its trailing separator run** — with the document's leading blank lines belonging to the first range and everything after the last grex to the last range. The intermediate volumen is the parent volumen with the ranges of the grex-instances revision *n* touches replaced by the corresponding tabula ranges; a grex the revision *inserts* has its tabula range spliced in at the corresponding parent boundary, and a grex the revision *deletes* has its parent range removed. Because ranges include their separators, no doubled or missing `\n\n` can result. (The parent↔tabula grex correspondence comes from the §8.8 alignment; one-revision-per-grex, §8.5, keeps it well-defined. HISTORY §8 *Revisions, sharpened*.)
Two refinements make the rule total and order-independent (HISTORY §14 *Revision detection, splice, index stability*): **(a) the phantom-grex fold** — a raw grex deriving zero atoms (e.g. a lone-NBSP line: not blank per §3.1 rule 2, but every segment strips empty per §4.3) has no spina entry; its bytes MUST fold into the preceding atom-bearing range, or into the first range when no atom-bearing grex precedes it, so the ranges still partition the volumen; **(b) the splice is subset-based** — the intermediate volumen for any set of inscribed revisions is a re-concatenation over the §8.8 cell list, taking tabula ranges for cells whose revision is in the subset and parent ranges otherwise, never patching a previous splice product. The result depends only on the subset, so per-revision inscription is order-independent, and the full subset reproduces the tabula bytes exactly.
Verbatim atom spans. The same partitioning discipline carries one level down: within a grex's verbatim range, each atom has a **verbatim span** — a half-open byte range running from the atom's first contributing byte to the next atom's first contributing byte (the grex range's end, for the last atom), with any bytes before the first atom belonging to the first span. Spans partition the grex range; each span owns its trailing whitespace. Contributing bytes are recovered through the content-extraction offset map: §3.1's CRLF folding and §3.2/§3.3's whitespace collapse are deterministic transforms, so every extracted character maps to a source byte range and segmentation boundaries (§4) map back to volumen offsets. Atom spans are derivable (not stored) and are consumed by the combining merge's materialization (§9.6). (HISTORY §9 *The combining merge, designed — Materialization*.)

— **Opt-in — `lit inscribe --all`.** A single version covering all currently pending revisions, with a single message.

These modes apply when the source is the tabula. With `--fons <source>`, the source already names one version and `--all` is not applicable.

8.7 Inscribe forms (tabula source)

The tabula-source CLI forms, all producing version objects per §8.6:

- `lit inscribe -<n> -m "<message>"` inscribes the n-th pending revision. Indices are 1-based and come from the most recent `lit status` output (the §10 #13 contract; indices renumber on every recomputation, §8.8).
- `lit inscribe` with no arguments opens `$EDITOR` with all pending revisions laid out, each followed by a message line. On save and exit, revisions are inscribed in document order.
- `lit inscribe --all -m "<message>"` inscribes everything as one version, non-interactively (the form scripted integrations use); `lit inscribe --all` without `-m` prompts for the message.
- `lit inscribe -m "<message>"` (bare `-m`) is accepted when exactly one revision is pending, and MUST refuse otherwise — with several intents pending, the implementation does not guess which one the message describes.

`-m` / `--message` supplies the message wherever non-interactive use needs one. To inscribe pending revisions to a non-default target, name the target: `lit inscribe <target> -<n> -m "<message>"`, `lit inscribe <target> --all -m "<message>"`, and `lit inscribe <target>` for the editor form.

8.8 Revision detection and index stability

The decomposition of (parent version, tabula) into the pending revisions of §8.5 MUST be the following deterministic, similarity-scoring-free algorithm (HISTORY §14 *Revision detection, splice, index stability*):

Document pairing. Same manifest path pairs first. Among the leftover paths, equal identity trios pair as **renames**; when one trio sits at several paths, the tie-break is sort-and-zip (sort both sides' paths bytewise, pair positionally). A rename accompanied by content edits decomposes as **removal + first-version** — once the trio changed there is no rename to detect (§7.2); atom dedup preserves lineage structurally. Consequently a path *swap* decomposes as two content edits (same-path pairing wins; only vanished/appeared paths can pair as renames — two swap-renames would not be independently inscribable, since the intermediate manifest would strand one document without a path). Remaining leftovers are removals and first-versions. A paired document whose volumen hash is unchanged contributes no revisions (a mode flip with unchanged bytes rides the attributes document's own revision — §1.1, §7.2).

Grex alignment (per changed paired document). Compute the longest common subsequence over the two fasciculus-hash sequences, with a **match-eager, parent-advance-on-tie** traceback — the unique leftmost trace. Matched positions are **anchors**; an anchor

whose verbatim byte ranges (§8.6) differ on the two sides is *touched*. Each gap between anchors resolves by two rules:

— **Rule A** — when both sides of the gap are non-empty and their concatenated atom-hash lists are equal as flat lists, the gap is one *structural* revision (a split/merge regrouping).
— **Rule B** — otherwise, positional pairing: min(M, N) *modify* revisions plus leftover pergrex deletions and insertions, one revision each. (Split-plus-edit therefore decomposes as modify + insert: accepted coarseness, per the strictness principle of §4.)

Formatting revisions. Each maximal run of adjacent touched anchors is one formatting revision (§8.5's contiguous-run rule).

Ordering is §8.5's document order; indices are 1-based.

Index stability. Indices renumber on every recomputation: the pending set is a pure function of (target-history tip, tabula) and is never stored (§6.9). `lit inscribe -<n>` MUST echo the resolved revision before committing, so a stale index is caught at the prompt rather than silently inscribing the wrong intent.

8.9 Exscribe

`lit exscribe` materializes a version from a history into the working tree: for each document in the version, its volumen bytes are written at its manifest path. It is the inverse direction of inscribe — the third *scribere* compound (*con-* gathers onto the tabula, *in-* drives into the history, *ex-* copies back out). (HISTORY §8 *Exscribe: history → working tree*; §11.)

— **Not a checkout.** No pointer moves; no version becomes "current"; no repository state changes other than the working-tree files. The next conscribe/inscribe proceeds ordinarily, exactly as if the writer had typed the restored text.
— **Source naming** follows `--fons` conventions (§8.3): a version hash, or a history name (`historia`, or an experiment) resolving to its latest inscribed version.
— **Document-set fidelity.** A whole-version exscribe makes the working tree's document set match the version: documents present in the tree but absent from the version are **deleted** (they are covered by the same guard below). Non-document and `.litignore`-excluded files are untouched. Without this, "restore an old version" would silently carry newer documents forward.
— **Dirty-tree guard.** If the working tree differs from both the latest historia version and the tabula state, the implementation MUST prompt before overwriting or deleting; it MUST NOT silently destroy unrecorded work.
— **Single-document restore** rides the same verb with a path argument: only that document's bytes are written; the rest of the tree (and the document set) is untouched.

The CLI surface is `lit exscribe <fons> [<path>]`; the dirty-tree guard prompts, and `--force` answers yes for scripted use. (HISTORY §14 *Exscribe CLI*.)

8.10 Version-object serialization

A version object is stored at `theca/version/<aa>/<vhash>` (§6.7) in this text form (HISTORY §14 *Serializations*):

```
lit-version 1
segmentation <id>
trio <volumen> <spina> <atom-spina>
...one trio line per document, sorted by volumen hash...
manifest
<volumen-hash> <encoded path>
...one line per manifest entry, sorted by path...
```

The **hashed pre-image** is everything before the `manifest` separator line; the **version hash** is SHA-256 of those bytes (§8's split, made concrete). The manifest region follows the separator and is excluded from the hash — it is the sole mutable-in-place region (§6.7). Manifest paths are UTF-8 with `%`, LF, and CR percent-encoded (`%25`, `%0A`, `%0D`); no other bytes are transformed.

8.11 Read surface

The coordinate-addressed read surface over stored versions (HISTORY §7 *Diff presentation — Read surface*):

- `lit show` / `lit show <history>` — the history listing (ordinal, short hash, timestamp, first message line).
- `lit show <version>` — the version's manifest listing.
- `lit show <version> <path>` — the document's verbatim volumen bytes.
- `lit show <version> <path> --atoms` — the citation listing: one atom per line as `aK gNaM <content form>` (both the flat atom-spina ordinal and the hierarchical coordinate, §7.4).
- `lit show <version> <path> <coord>` — coordinate resolution, where `<coord>` is `gN` (the grex's atoms, one content form per line), `gNaM`, or `aK` (that atom's content form). With `--hash`, the object hash is printed instead of content — the resolver behind the §6.6 citation primitive ("*Tom Sawyer* atom #123" → atom hash; `gN --hash` yields the fasciculus hash).

Out-of-range coordinates are errors naming the document's actual extent. Version arguments follow §8.3 resolution (including short prefixes).

9 Histories and merge

A repository carries:

- exactly one **historia** (§9.1) — the canonical line of development; hard-coded name, undeletable;
- zero or more **experimenta** (§9.2; English: *experiment*; Greek: πεῖρα *peira*) — alternative working lines; user-named, deletable.

A history's state is an **append-only sequence of inscriptiones** (§8.4) in inscription order. Walking the history yields the version sequence; each version hash resolves through the theca (§6.7) to the version object. Because version objects do not chain to parents (§8), histories are the sole source of version ordering. Viewing a history is direct file iteration with no graph walk. There is no separate `lit log` command; the read surface (viewing a history, printing a version's content) is `lit show` (§8.11) and diff display (§7.4). (HISTORY §8 *Snapshot-primary*; HISTORY §9.)

9.1 Historia

A repository MUST contain exactly one **historia**, with the hard-coded name *historia*. It MUST be present in every repository and MUST NOT be deletable or renamable. It is the default target of `lit inscribe` (§8.3). The historia file is created, empty, at repository creation (§1.3).

The first inscriptio in a fresh repository's historia is produced by the first successful `lit inscribe` after the first conscribe (§1.3, §8.2).

9.2 Experiments

An **experimentum** (English: *experiment*; Greek: πεῖρα *peira*) is an alternative history alongside historia. A repository MAY contain zero or more experiments, distinguished by user-chosen names. An experiment name MUST NOT be `historia`, MUST NOT be exactly 64 lowercase hexadecimal characters (§8.3), MUST be non-empty, and MUST NOT contain `/`, whitespace, or control characters or begin with `.` — the minimal rules that keep names safe as file names under `.lit/experimenta/` and unambiguous as CLI arguments. (HISTORY §14 *Experiments CLI*.)

Experiments MAY be created and deleted. Deleting an experiment removes its history file (the inscriptiones), but the version objects it named remain in the theca and are still reachable from any other history that inscribed them. Deleting an experiment also removes its entries from provenance queries (§9.5): a version's apparent origin can shift to the earliest *surviving* entry. This is accepted v1 behavior.

The CLI surface is `lit experiment create <name>` and `lit experiment delete <name>` (implementations MAY add `list`). Experiments otherwise share historia's shape (append-only inscriptio sequence) and behavior under `lit inscribe` (§8.3). (HISTORY §14 *Experiments CLI*.)

9.3 History file shape

A history is a line-oriented sequence of inscriptiones in inscription order, one entry per inscriptio. Each entry MUST encode the version hash, timestamp, and message of §8.4.

This is the same artefact Litorium publishes as the project's historia. The historia file is byte-identical at the lit-side (where it is a local history file in `.lit/`) and at the Litorium-platform layer (where possession of the file, together with the project identifier, is the read-access capability for the project). One artefact, one name; the platform-layer behavior is specified where it lives, not as a separate object.

The per-entry encoding is one line per inscriptio (HISTORY §14 *Serializations*):

`<version-hash> TAB <timestamp> TAB <escaped message> LF`

The timestamp is ISO-8601 UTC with microseconds (`2026-07-02T21:55:03.123456Z`) — lexicographic order equals chronological order, which §9.5's provenance rule consumes. The message is backslash-escaped: `\\`, `\t`, `\n`, `\r`. Additional per-entry fields (author, etc.) remain *(pending — §10 #12)*.

9.4 Merge as inscribe

A "merge" is a single `lit inscribe` operation with `--fons <source>`: append an inscriptio naming the source's latest version (or any named version) to the target history. This is **adoption** — the target takes the source's version as-is (the fast-forward analogue). There is no merge commit; no special version object is produced. When both histories have changed the same document since divergence and a *combined* state is wanted, the combining merge of §9.6 materializes it into the working tree and the result reaches the target history through the ordinary conscribe → inscribe path.

The merge primitive imposes **no common-ancestor requirement** on its inputs. Any two histories can be merged, including histories with empty intersection. There is no analogue of git's `--allow-unrelated-histories`.

Project merge. Folding two repositories into one — the situation produced when the §1.4 scan finds multiple `.lit/` descendants — unions the thecae (free, since content-addressed) and folds the histories by **designation**: the writer designates one descendant's historia as the merged repository's historia (at the §1.4 confirmation prompt); every other descendant's historia becomes an experiment named for that descendant's former root-relative directory (e.g. `novella-historia`), and descendants' experiments are imported under the same collision-avoiding prefix. Nothing is interleaved by timestamp. Manifest conflicts are resolved by prompted re-path (§1.4). The writer can then merge the demoted historiae into the designated one — adoption or combining — at their own pace. (HISTORY §9 *Project merge is ordinary merge*.)

9.5 Provenance via inscriptio multiset

A version V appears as `(V, t)` in every history that inscribed it, where `t` is that inscription's timestamp. Provenance — *where did this version originally come from?* — is the **earliest-by-timestamp** entry across all existing histories; the full multiset is V's life-path through the system. Ties at equal timestamps resolve deterministically: historia outranks experiments; experiments tie-break by name. Within one history, order is entry order (§8.4), not timestamp order.

Provenance is queryable from the history files; no provenance is stored as a structural special case in version objects. Note the scope: merge-ness is recoverable from the multiset for **adoption** merges (§9.4), where the version appears in both histories; a combining merge (§9.6) produces a fresh version whose multiset has a single entry, and its merge-ness lives in the inscriptio's message. Deleting an experiment removes its entries from the multiset (§9.2).

The query surface is **`lit stemma`** (στέμμα — the *stemma codicum* of textual criticism; the same form in both registers, like *historia*). Output is plain deterministic bytes, part of the conformance surface like §7.4. (HISTORY §9 *Provenance surfaced*.)

— `lit stemma [<version>]` — the version's **life-path** (default: the historia's latest; version arguments follow §8.3 resolution). A `version <hash>` header, then one line per inscriptio naming the version, across all histories, sorted by (timestamp, historia-before-experiments, experiment name, entry order): two-space indent, history name, `#<ordinal>` (1-based entry ordinal within that history), timestamp, and the first message line, with the first line — the **origin** — suffixed `(origin)`. A version inscribed in no surviving history prints `not inscribed in any history`.

— `lit stemma <version> <path> <coord>` — **atom provenance** (beyond this section's version scope; HISTORY §9). The coordinate (`gNaM` or `aK`, §7.4) resolves to an atom hash; membership of that hash in each inscribed version's atom-spina is exact, so the output reports the atom's hash and snippet, its **first** and **last** inscriptio (same sort as above, restricted to versions containing the atom), and `present in <k> of <m> inscriptiones across <h> histories` (the multiset, versions counted once per inscriptio).

v1 is local-only and assumes a single clock; with the tie-break above, provenance is deterministic. v2+ remoting introduces clock skew and is deferred. (HISTORY §9 *One-clock semantics*; HISTORY §13.)

9.6 The combining merge: `lit contexe`

`lit contexe <source> [<target>] [--base <version>] [--force]` runs the combining merge and materializes the result into the working tree; the merged state then becomes a version through the ordinary conscribe → inscribe path, so document identity is always `sha256` of bytes that actually sat in the working tree. The source follows §8.3 resolution; the target names a history and defaults to *historia*; the working tree is guarded exactly as in §8.9 (prompt when the tree differs from both the latest historia version and the tabula; `--force` answers yes). Latin *contexo*, Greek *synhyphainō* (συνυφαίνω) — the operation's noun is **contextio**. (HISTORY §9 *The combining merge, designed*; §11.)

Base selection. With A = the target's latest version and B = the source, the base O defaults to the **latest common version**: among version hashes inscribed in both histories, the one whose most recent inscriptio in the *target* history has the greatest entry index. `--base <version>` overrides. When the source is a bare version hash (no source history to intersect) and `--base` is absent, or when no common version exists, the merge is **baseless** and degenerates to a 2-way: atoms common to both sides keep; divergent regions become variants.

Document pairing is by manifest path, with three-way set logic per path: present in all three → content merge (below); added on one side → kept; removed on one side with the other side unchanged from O (equal trio) → removed; removed on one side but modified on the other → a **modify/delete variant** (the surviving side's document is kept; the

variant goes to the ledger); added on both sides with different content → a baseless per-document 2-way. Renames pair as removal + first-version (§8.8's accepted coarseness; rename-aware pairing is a recorded refinement, HISTORY §13).

Content merge (per document; A, B against O). Trio equality gives O(1) fast paths: A = B → take A; A = O → take B; B = O → take A. Otherwise a **two-level diff3**:

1. **Fasciculus level.** LCS(O, A) and LCS(O, B) over the fasciculus-hash sequences (§8.8's algorithm and tie-break) build a skeleton of O-grexes anchored identically in both alignments. Per skeleton gap: a region changed on exactly one side takes that side's grexes; a region equal on both sides takes A's; a region changed on both sides descends to —
2. **Atom level.** A flat diff3 over the region's concatenated atom-hash sequences, with the same LCS/tie-break machinery: sub-regions changed on one side compose; sub-regions changed identically compose; sub-regions changed differently are **contested** — each is a **variant** (HISTORY §17).

Formatting never produces a variant. Wherever the chosen atoms equal the other side's atoms but the verbatim bytes differ (equal fasciculus hash, different bytes — a formatting-only divergence), the **target's bytes win silently**: the hashes prove no words changed. Implementations MUST NOT surface formatting-only divergence as a variant.

Variants materialize inline. A contested sub-region materializes with both readings adjacent — the target's atoms first, then the source's — as plain text, with **no markers of any kind**; the file remains an ordinary, always-readable document (HISTORY §17). Each variant is recorded in the **contextio ledger** — the merge's **apparatus** — at `.lit/contextio` (§6.9): the source, target, and base identities, and per variant the path, the `gNaM` coordinate range of the inline region in the materialized file, the kind (`text`, `modify-delete`, `delete-modify`), and both sides' atom hashes. Ledger lifecycle: `lit conflicts` lists it (readings labeled by history name — the siglum of the hand carrying each; the protocol peer is `variants`, PROTOCOL §7.2); `lit conflicts clear` discards it; `lit status` MUST show a warning block while it is non-empty; `lit inscribe` MUST prompt before inscribing while it is non-empty and MUST clear it when an inscription consumes the last pending revision. Resolution is *recensio* by ordinary editing — delete the reading you reject (keep/adopt), keep both, or write a new reading informed by both (*emendatio*); the next conscribe sees it as a normal revision. *The variant-presentation details and the recension UX are v1-provisional pending iteration against real editorial workflows (HISTORY §13, §17; §10 #9); the algorithm, base selection, and ledger format above are normative.*

Materialization. Every output grex becomes a byte chunk: a grex untouched by the merge or changed on exactly one side contributes that side's verbatim grex range (§8.6) right-trimmed of its trailing separator run (the target's copy for untouched and formatting-divergent grexes); a both-side (merged or contested) grex assembles from **verbatim atom spans** (§8.6) — each contributing atom's bytes taken from the atom's *first contributing byte* to its span's end (identical to the span except for a grex's first atom, whose

leading range bytes are dropped), right-trimmed of the trailing run of §4.3 edge-class characters, joined with the mode joiner. Grex boundaries in merged regions follow each atom's contributing side (a boundary falls after atom x iff x's own side had one after it), except that a variant's two readings join within one grex. Chunks join with `\n\n`, and a content-merged document is terminated with a single LF. Atom-interior bytes survive exactly; only inter-atom and inter-grex whitespace is normalized, and only in documents the merge actually content-merged (fast-path documents keep their exact bytes).

The structural scoping that makes all this cheap (HISTORY §9 *Merge: structure scopes the work*): identical fasciculi compose in O(1) via hash comparison; the atom-level diff3 runs over a bounded handful of hashes per touched paragraph; moves across fasciculi are exact hash matches, never similarity scores.

9.7 The exemplar exchange

Collaboration in v1 is correspondence (HISTORY §10): a version travels to another hand as its **raw files**, and returns the same way. Because a version's documents fully determine every derived artefact under the pinned segmentation (§2, §4), nothing else needs to travel; the recipient needs no repository, no account, and possibly no lit.

The archive. An **exemplar** is a gzip-compressed tar archive containing the version's documents at their manifest paths, plus a metadata file `.exemplar` at the archive root — three lines, each `<key> SP <value> LF`:

```
exemplar 1
basis <64-hex version hash>
segmentatio <segmentation identifier (§4)>
```

`basis` names the version the exemplar was cut from: on return it is the diff target, and — when the target history has moved past it — the base for `lit contexe --base` (§9.6). `segmentatio` lets a lit-using recipient derive identical atoms in their own repository. The archive carries no attribution; in correspondence, the sender is the envelope. Entries appear in manifest order with epoch timestamps; the archive is a convenience form, not an identity-bearing artefact — no part of it enters any hash pre-image.

`lit exemplar <source> [-o <file>]` materializes the resolved version's documents (§8.3 source resolution; §8.11 materialization) into an archive, default name `exemplar-<12-hex>.tgz`. With no `<source>`, in a directory containing an `.exemplar` file, it repacks that directory's files together with the existing `.exemplar` verbatim — the recipient's return trip. Plain `tar` remains the equivalent old-fashioned form in both directions.

`lit collate <file> [--force]` runs inside a repository. It MUST reject an archive without a well-formed `.exemplar` entry, MAY warn when `segmentatio` differs from the repository's (informative only — the author collates bytes, not atoms), and MUST apply the §8.9 dirty-tree guard before writing. It then writes the archive's documents into the working tree with exscribe's document-set fidelity (§8.11): documents present in the tree but absent from the archive are deleted; `.exemplar` itself is never written into the tree. It reports the basis and counts. Review and adoption are the ordinary verbs: `lit diff` against the basis, then conscribe and inscribe — or `lit contexe --base <basis>` when the historia has moved past the basis.

10 Pending specification

These items were intentionally unspecified in v0; resolving each was a prerequisite for the implementation slice that depended on it. Most crystallized on 2026-07-05, after two independent implementations realized the provisional answers byte-identically (HISTORY §14); resolved rows keep their numbers (references throughout the corpus use them) and point at the normative text. Every "*(pending — §10 #N)*" pointer in the body resolves to an open row here.

#	Item	Status
1	Repository on-disk layout	**Resolved** — §6.9 (layout, format marker, segmentation record, no stored pending state), §8.1 (tabula), §9.3 (history files). Residue resolved 2026-07-06: no literium pack format, ever — bulk transfer is substrate-level (§6.8; HISTORY §5 *Packs resolved out of the core*).
2	Attributes file path and filename; non-interactive mode flag	**Resolved** — §1.1 (`.litattributes`), §1.3 (`--mode`).
3	Object framing	**Resolved** — §6.8 (none; raw bytes).
4	Wire formats	**Resolved** for everything v1 stores — §8.10 (version object incl. hashed/unhashed split and manifest encoding), §9.3 (inscriptio), §8.1 (tabula). **Open residue:** edit-op encodings for v2+ transport (§7.3; HISTORY §13).
5	Revision-detection algorithm and index stability	**Resolved** — §8.8 (pairing, LCS alignment with leftmost trace, Rule A/B, formatting runs, renumber-on-recompute), §8.6 (phantom-grex fold, subset-based splice).
6	Garbage collection (`lit gc`)	**Resolved** — §6.10 (mark-and-sweep; roots = surviving inscriptiones + live tabula + contextio atom hashes; deterministic; `--dry-run`). (HISTORY §5 *Garbage collection designed*.)
7	Verification walk (`lit fsck`)	**Resolved** — the walk is the already-normative checks: content-object hashes (§0/§6.7), version pre-image + structural manifest check (§6.7), atom-spina = walk equality (§6.6), re-derivation under historical modes (§6.5, §1.1), plus every inscriptio resolving to a stored version (§9.3).

#	Item	Status
8	Diff display format and the read surface	**Resolved** — §7.4 (coordinates, grex hunks, exact move/split/merge annotations), §8.11 (show forms, `--atoms`, coordinate resolution). Reframed 2026-07-07 as the typed **collatio** in the classical categories (transpositio/additio/omissio/substitutio + divisio/coniunctio) with the apparatus/`--leiden` /marks renderings (§7.4; HISTORY §7, §17). **Open residue:** the exact byte grammar of the apparatus and Leiden renderings, and the word-level sub-collation view.
9	Merge algorithm, materialization, invocation surface, variant presentation	**Resolved** — §9.6 (`lit contexe`, multiset base selection, two-level diff3, verbatim-atom-span materialization) and the **variant/apparatus/recension** model (HISTORY §17): a merge auto-resolves the mechanical categories and leaves only contested **substitutiones** in the apparatus, worked by *recensio* (keep/adopt/both/emend), the trunk always readable. **Open residue:** the recension UX (the four resolution moves' CLI/protocol surface) and the apparatus byte grammar (with #8). The multi-descendant **project merge** of §1.4 remains unimplemented pending this machinery's exercise.
10	Allowed segmentation refinements (deterministic, language-agnostic only) — leading candidate: closing-quote bundling — and the explicit **segmentation-version migration** command (§4)	Open — refinement on §4
11	CLI surface for ordinal addressing	**Resolved** — §7.4 coordinates (`gN`, `gNaM`, flat `aK`; 1-based display) and §8.11 (`lit show <version> <path> <coord>`, `--atoms`, `--hash` as the §6.6 citation resolver). The compact `g10a2` form supersedes the earlier colon candidates. (HISTORY §7 *Diff presentation*.)

#	Item	Status
12	Version-object fields beyond identity trios — author, per-revision ordering in `--all` mode. Inscriptio fields beyond version hash + timestamp + message — author, etc. (Parent-version reference is ruled out, §8; ordering lives in histories, §9; the segmentation identifier is settled as part of the hashed pre-image, §8.10.)	Open — `lit inscribe`
13	`lit status` output contract	**Resolved** — a per-revision numbered list in §8.5 document order: index, category or manifest kind, path, grex ordinal(s), first-atom snippet; manifest-level changes appear in the same numbered list (each is one inscribable revision). (HISTORY §14.)
14	Experiment CLI surface	**Resolved** — §9.2 (name rules; `lit experiment create/delete`).
15	Exscribe CLI details	**Resolved** — §8.9 (`lit exscribe <fons> [<path>]`, `--force`).
16	Inscription-message prompt format	**Resolved** — §8.3 (`inscribe: <source> -> <target>`).
17	Provenance query CLI shape	**Resolved** — §9.5 (`lit stemma`: version life-path with marked origin; atom-level first/last-inscribed via coordinates). (HISTORY §9 *Provenance surfaced*.)

Each open item back-references HISTORY §13 where the question is named with broader context; resolved items are recorded with rationale in HISTORY §14.

References

[1] S. Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119, BCP 14, 1997.

[2] Git project. gitignore — specifies intentionally untracked files to ignore. Git documentation.

[3] Unicode Consortium. Unicode standard annex #29: Unicode text segmentation. UAX #29, revision for Unicode 16.0.0, 2024.

[4] Unicode Consortium. The unicode standard, version 16.0.0, 2024.

Index