Literium — User Guide (working draft)

The literium project

version 1.0-draft

About this document. Working draft, not a polished guide. A scratchpad for ideas, workflows, mental models, and frequently-asked questions — anything that captures how Literium feels from the user's seat. Add to it freely; entries that hold up over time can be promoted into the eventual real guide.

Contents

1 How versions and histories work

When you save your work in lit, you're not creating a file. You're making a **version**: a snapshot of your manuscript at a moment, preserved alongside every other version you've ever saved. Versions don't replace each other; they accumulate. Months later you can read any of them, compare any pair, or pull a sentence from one into another.

This chapter walks through how versions are made, where they live, and how to move between them.

1.1 The writing tablet (*tabula*)

Lit gives you a small staging space called the *tabula* — Latin for "writing tablet," the wax board Roman writers used for drafts. The tabula holds your *next* version: the one you're about to commit but haven't yet.

You fill the tabula with one command:

```
lit conscribe
```

This snapshots your working files onto the tabula. If you've changed something since the last time, that change is now staged; if you haven't, lit will tell you there's nothing new.

You can `conscribe` repeatedly while you work. Each new conscribe replaces the previous tabula state — the tabula always holds your *latest* in-progress draft. There's no stack, no list, no "stage these but not those." The tabula is just one slot.

This is also where you keep work that isn't ready to commit. If you need to switch contexts — answer email, write something else, take a break — `conscribe` captures your current state on the tabula. You come back later, keep editing, and conscribe again.

1.2 Inscribing a version

When you're ready to commit the tabula as a real, named version, you *inscribe* it:

```
lit inscribe
```

This finalizes whatever is on the tabula into a version — a permanent snapshot — and writes a new entry into your project's *historia* (more on that below). Lit asks you for a message describing the state you're saving ("first complete draft," "after Anna's notes," "before the cuts") and that message becomes part of the entry.

The Latin contrast is intentional. *Conscribere* — "to gather, write together" — describes the soft, draft-stage act of putting something on a wax tablet. *Inscribere* — "to write into" — is the verb classical Roman writers used for inscriptions on stone. The tabula is wax; the inscription is permanent.

After an inscribe that consumes everything pending, the tabula is empty again, ready for the next draft. (Inscribing a single revision with `-<n>` consumes only that revision; the rest stays pending on the tabula.)

1.3 *Inscriptiones*: the entries in your history

Each call to `lit inscribe` produces one *inscriptio* — an entry in a history. An inscriptio carries:

- the version itself (the snapshot of your manuscript),
- a timestamp (when you inscribed it),
- your message ("first viable draft," "before the cuts").

The same word, *inscriptio* / *inscription*, names both the entry as a whole and the message text on it — the same way a stone inscription is both the carved entry and the words on it.

1.4 Two kinds of histories: *historia* and *peirae*

Every project has one **historia** — its canonical line of work, the spine of the manuscript's life. Every inscription you make goes there by default. The historia is the official record: when your project ships to a publisher or goes on Litorium, the historia is what gets shared.

Most writers will only ever use the historia. Inscribe drafts as you go; the historia accumulates them. Done.

For cases where you need to develop work in parallel — an editor's pass alongside your own draft, an exploratory rewrite that may or may not work, a publisher's copyedit you want to keep separate — lit gives you **experiments** (Latin *experimentum*; Greek *peira*, "a trial, an attempt"). An experiment is a separate history, just like the historia, but with a name you choose (`peira-1`, `editor-pass`, `anna-notes`) and the freedom to delete it later.

To inscribe onto an experiment instead of the historia (the experiment must already exist — inscribe never creates one as a side effect; the creation command is its own small operation, still being shaped — SPEC §10 #14):

```
lit inscribe peira-1
```

You can move work between histories any time. A version inscribed in one history can be brought into another, including the historia, with `--fons` (Latin "source"):

```
lit inscribe historia --fons peira-1
```

That command writes peira-1's latest version into the historia. Other tools call this a fast-forward "merge"; in lit it's just naming the source — **adoption**.

When both lines have changed the same document, adoption isn't enough — you want the two strands *woven together*. That's **contexe** (Latin *contexere*, "to weave together" — *texere* is the root of *text*):

```
lit contexe peira-1
```

Contexe finds the last version the two histories share (or takes `--base`), weaves both sides' changes into your working files, and — where both of you rewrote the same sentence — puts **both variants next to each other in the text**, plain prose, no markers. `lit conflicts` lists the apparatus — each variant with its g/a coordinates and the hand behind each reading; you resolve by deleting the reading you don't want (or writing a better one), exactly like any other edit, then conscribe and inscribe as usual. One practical note: seed a fresh experiment with the state you're diverging from (`lit inscribe peira-1 --fons historia`) before working in it, so contexe can find the common ancestor later.

1.5 Returning to an earlier state

Returning to an earlier state is two independent acts — updating the *record* and updating your *files* — and lit gives each its own verb.

To record the return in your history, inscribe the earlier version:

```
lit inscribe --fons <earlier-version>
```

The version's content doesn't change — it's the same snapshot it always was — but a new inscription appears in the historia, dated now, with whatever message you choose. The history records "we returned to this state at this time."

To actually put that state back in front of you — the files in your working tree — *exscribe* it:

```
lit exscribe <earlier-version>
```

Exscribe (Latin *exscribere*, "to copy out" — the third *scribere* verb: *con-* gathers onto the tablet, *in-* drives into the record, *ex-* copies back out) writes the version's documents into your working tree, exactly as if you had typed them. If you have unsaved work in the tree, lit asks before overwriting. Nothing else changes — no pointer moves, nothing becomes "current"; you just have those bytes in your files again, and your next conscribe/inscribe proceeds as always. You can also restore a single file: `lit exscribe <version> <path>`.

1.6 What this means in practice

A few patterns to internalize:

Linear writing. Most days, you write, conscribe, inscribe. Conscribe captures a draft; inscribe finalizes it. Your historia grows; nothing else needs to exist.

Parallel exploration. When you're not sure whether a rewrite will land, open an experiment. Inscribe iterations into it. If the rewrite works, inscribe its latest version into the historia. If it doesn't, delete the experiment — its versions remain in storage, but its history line is gone, and the historia is untouched.

Editorial review. An editor working with their own copy can inscribe their pass into an `editor-pass` experiment. You compare the experiment's tip against the historia's tip. Bring the changes you want into the historia; leave the rest in the experiment.

Time travel. Any version is a real, complete snapshot. To see your manuscript as it was three months ago, find that version in the historia and view it; to *work on* it again, exscribe it into your working tree. To use a paragraph from then in now's draft, copy the atom hashes — the content doesn't need to be moved, only re-referenced.

2 Lit and git complement each other

Literium and git aren't competitors. They're optimized for different content: git is line-oriented and works best on code; lit is sentence-oriented (line-oriented in poetry mode)

and works best on prose. When a project has both — a codebase with documentation, a research tool with a paper, a novel with companion notes — using both side by side is often the right call.

There are two patterns for running them together. Pick the one that matches how you want to think about commits.

2.1 Pattern A: git drives, lit shadows (recommended for most code+docs repos)

You commit normally with git. A `post-commit` hook runs `lit conscribe && lit inscribe` against the same message, and lit silently no-ops when the commit didn't touch any prose file (`conscribe` refuses on a directory with no pending changes, so the hook is naturally idempotent on code-only commits).

Setup is one ignore entry plus the hook:

— `.gitignore` adds `.lit/`.

— `.litignore` excludes the non-prose files (e.g. everything except `*.md`, `docs/**/*.md`, etc.). `.git/` is excluded too — each tool already implicitly ignores its own metadata directory, but neither knows about the other's.

— `.git/hooks/post-commit` (rough shape):

```
#!/bin/sh
msg=$(git log -1 --pretty=%B)
lit conscribe 2>/dev/null && lit inscribe --all -m "$msg" || true
```

Now `git commit` is your single workflow. Code commits roll past lit untouched; doc commits get a lit inscriptio mirroring git's message. You get git's line-level history on everything, plus lit's sentence-level history (atoms, fasciculi, structural-vs-substantial classification, content-tracking across files) on the prose subset, without ever running a second commit verb.

The price is that lit's commit cadence is git's cadence. Lit's default is one inscriptio per revision (per coherent intent within a version); under this pattern you get one inscriptio per `git commit` instead, and the message is whatever git got. If you want to split a doc edit into multiple intents, run `lit conscribe` and `lit inscribe` directly between git commits — the hook only fires when git commits, so manual lit work is additive.

2.2 Pattern B: disjoint tracking (each tool sees its own slice)

If you'd rather keep the two histories fully independent — e.g. you want git's history on code to stay clean of doc churn, or you commit prose on a different cadence than code — split the working tree by ignore files:

— `.gitignore` excludes `*.md` (and `.lit/`).

— `.litignore` excludes everything *except* `*.md` (and `.git/`).

Now `git status` and `lit status` describe disjoint slices of the working tree. Code changes go into git; documentation changes go into lit. Each tool tracks what it's good at

— line-level diffs and code-conventional commits for the source, sentence-level diffs and per-revision messages for the docs — without either tool stepping on the other.

A few practical notes for this pattern:

- The two repositories share a working tree but are otherwise independent. They can be branched, merged, and conscribed/committed on completely different schedules.
- Using both at once doesn't slow either down — each scans only the files it cares about.
- The cost is double management: every commit is a decision about which tool to run, and an edit that genuinely spans both (e.g. a code change documented in the same breath) needs two commits with two messages.

2.3 Picking between them

Pattern A is the right default when most prose changes accompany code changes anyway — typical for README/docs in a software project. One workflow, no double management; you only lose intent-level granularity inside a single git commit, and you can still run lit directly when you want it.

Pattern B is the right default when prose evolves on its own rhythm — a research repo where the paper has its own draft cadence, or a novel with companion build scripts. Two workflows, but each tool's history reflects only what it's good at tracking.

Both patterns generalize beyond codebases. A novelist whose project also has cover art and reference PDFs can keep prose in lit and binary assets in git. A researcher can keep the paper in lit and the analysis code in git. Anywhere a project has prose alongside non-prose content, one of these two patterns applies.

3 What lit can do that git can't

Two structural advantages worth naming, because they're the kind of thing git couldn't reach without rewriting its storage model.

3.1 Tracking content across files is structural, not heuristic

When Linus Torvalds talks about git tracking content across renames and moves, what he's describing is `-M`/`-C`/`--find-copies-harder` — similarity scoring at *diff display time*, applied to whole-file blob diffs against a configurable threshold. The "tracking" is reconstructed every time you ask, only ever surfaces in the diff view, and isn't queryable as data. Git's blobs are whole files; two files sharing a sentence share no hashes, and the engine has nothing to grip.

Lit inverts this. Atoms are content-addressed at the sentence level (the line level in poetry mode) and deduplicated across the whole repository. A sentence that moves from chapter 1 to chapter 5 is *literally the same atom* in both places — same hash, same storage object. Asking "where has this paragraph appeared in this manuscript's history?" becomes a graph query over which fasciculi reference which atoms, not a similarity search. The answer is exact and instantaneous, not a heuristic that may or may not surface.

This works because paths are not part of a version's content identity. A version says *which atoms exist*; the version's *manifest* (a separate listing inside the version object) says

where those atoms are currently arranged on disk. The same sentence at a different path is the same sentence.

3.2 Merging two manuscripts

If you've been writing two related projects in two separate lit repositories — say a novel and a companion novella that share a setting, or a paper and its supplementary appendix — and decide they should be one project, you can. Run `lit conscribe` from a directory that contains both project directories; lit will list the two `.lit/` repositories it found and ask whether to fold them into one. At the same prompt you pick which project's historia carries forward as the merged project's historia; the other project's historia arrives as an experiment named for its old directory (e.g. `novella-historia`), ready to be merged in at your own pace — or left as a parallel line forever. You end up with a single repository whose theca is the union of both and whose version manifests place each document under its original sub-path.

Sentences that appear in both manuscripts collide in the unified theca by content hash automatically — you can now query "where else does this sentence appear" across material that used to be in separate stores. Citations by atom hash that worked in either project before still work after, unchanged.

Git fundamentally can't do this cleanly. Git's commit objects bake path information into the commit hash via the tree pointer; merging two unrelated repositories into one means rewriting every commit (a `git filter-branch`-class operation with brand-new hashes throughout, breaking every external reference). Lit's structural separation of content state from workspace layout makes project merge a primitive operation, not a recovery operation.

4 Merging is recension, and merges have no conflicts

When you weave one line of work into another — `lit merge` (contexe) — literium does not stop you at a wall of `<<<<<<<` markers. There is no "conflicted" state you must clear before you can move. The reason is structural: because every sentence (atom) and every paragraph (grex) is tracked by content hash, the machine can *see* what happened and sort it, and most of what git would flag as a conflict isn't a decision at all.

A merge settles the mechanical cases silently, in the language textual critics have used for centuries:

- a **transposition** — a paragraph or sentence *moved* — is recognized by its hash and simply applied; git, diffing lines, can't tell a move from a delete-and-add, and conflicts on the overlap;
- an **addition** or **omission** — a sentence one side added or dropped — is a keep-or-drop;
- a **division** or **conjunction** — a paragraph split or joined — is applied structurally.

What's left after all of that is the only thing that ever needed *you*: a **substitution** — two hands wrote the same sentence differently. That is a **variant**, not a conflict. It carries no failure, no ours-versus-theirs; it's the ordinary substance of editing, and literium presents

it the way an editor already reads: your document stays whole and legible, and the competing readings sit in an **apparatus** beside it — the critical-edition layout, `sentence ] your reading (historia) : their reading (anna)`, each tagged with the hand that wrote it.

Working through the variants is **recension** (*recensio* — the scholar's word for establishing a text from its witnesses). You take them in any order — they're independent, so nothing cascades the way git's conflicts do — and at each one you **keep** (your reading stands), **adopt** (take theirs), **both** (they weren't competing — keep the two sentences), or **emend**: write a *new* reading, better than either. That last is what a good editor actually does, so it's a first-class move, not a manual retype. `lit status` counts the open variants; when it reaches zero the merge is just an ordinary version to record.

And it descends the way you'd read a proof: accept the paragraph moves first, then look only at the paragraphs that changed, then the sentence moves within them, and only inside a genuinely rewritten sentence do you drop to the words — which is where a diff finally shows you text you can read rather than hashes you can't. `lit diff` shows the same apparatus; `lit diff --leiden` shows it in the typewriter-era editorial brackets (`^` to insert, `[ ]` a variant span, `{ }` a deletion) if that's the hand you learned. (Design in progress — HISTORY §17.)

5 Literium for Git users

If you've used git before, the concepts above will feel familiar — but the vocabulary, and a few of the underlying choices, are deliberately different. This chapter maps lit terms to git terms and walks through the deliberate departures.

5.1 The git verb works — type either name

Every operation is reachable by both its Literium name and its plain-English git name. The two are equal **peers** — neither is the "real" one — and both appear in `lit --help`. Type whichever is in your fingers:

Type this…	…or this	It does
`lit add` (or `stage`)	`lit conscribe`	snapshot the working tree onto the tabula
`lit commit`	`lit inscribe`	finalize the tabula onto a history
`lit checkout` / `restore`	`lit exscribe`	materialize a version into the tree
`lit merge`	`lit contexe`	combining merge
`lit branch`	`lit experiment`	create / list / delete an experiment
`lit log`	`lit show`	view a history or a version
`lit blame`	`lit stemma`	atom provenance

`status`, `diff`, `gc`, `fsck`, `push`, `pull`, and `clone` are spelled the same in both worlds, so they just work.

The git verbs Literium *doesn't* have redirect you instead of failing with a blank error:

```
$ lit rebase
lit: literium has no rebase. Histories are linear event logs — there is
   nothing to replay onto a new base. (TERMINOLOGY: rebase.)
```

```
$ lit revert 9…f2c
lit: literium has no revert primitive. Re-inscribe the old version: `lit
   inscribe --fons <version>`. (TERMINOLOGY: revert.)
```

The same holds for `init`, `cherry-pick`, `switch`, `stash`, `fetch`, `tag`, and `remote` — each prints the one-line Literium way to do it.

One caveat: an alias is a *name*, not a reimplementation of git's semantics. `lit blame` takes `stemma`'s `<version> <path> <coord>` arguments, not `git blame`'s `<path>`; `lit branch` is `experiment`'s subcommands (`create` / `delete` / `list`), not git's. The peer gets you to the right operation; the operation keeps its own shape. (The English/git register is also the API surface the Literium editor plugins bind against — HISTORY §15.)

5.2 Vocabulary mapping

Git	Literium
`commit` (the saved snapshot)	*version*
`commit` (the verb)	`lit inscribe`
commit message	*inscription*
staging area / index, plus stash	*tabula* (single slot, no per-hunk selection)
`git add` (snapshot to staging)	`lit conscribe`
`main` / `master`	*historia* (hard-coded name; cannot be renamed)
feature branch / topic branch	*experimentum* / *peira*
`git merge` (fast-forward)	`lit inscribe --fons <experiment>` (adoption)
`git merge` (true merge)	`lit contexe <experiment>` (weaves both sides in; moves/adds auto-resolve, contested sentences become *variants* in an apparatus — no conflicts, no markers; see *Merging is recension* above)
conflict / conflict markers	*variant* (a contested substitution) + the *apparatus* (the reading beside the text, not `<<<<<<<` in it)
`git cherry-pick`	`lit inscribe --fons <version-hash>`
`git revert`	`lit inscribe --fons <old-version-hash>`
`git rebase`	(no analogue)
`git log`	(no command; the history file *is* the log — `lit show` views it)
`git log --all` / blame-ish provenance	`lit stemma [<version>]` (a version's life-path with its origin marked) and `lit stemma <version> <path> g3a2` (which inscription first carried this sentence — exact, via atom hashes)
`git checkout <commit> -- .` / `git restore`	`lit exscribe <version>` (writes files; moves no pointer)
`HEAD`, `git checkout <branch>`	(no analogue; no version or history is "current")

5.3 The deliberate departures

Versions don't have parent pointers. A git commit names its parent (or two, for merges); a lit version is a pure snapshot, with no link to "what came before." History order lives in the history files (the historia and any experiments), not in the version objects themselves. This has several consequences worth knowing:

- **No DAG.** Lit histories are flat lists, not a directed acyclic graph. There's no graph to walk, no merge-base to find, no "common ancestor" to discover.
- **Cherry-pick is just copying a hash.** In git, cherry-picking constructs a derived patch and applies it as a new commit. In lit, it's the same primitive as direct inscription — name a version's hash and a target history.
- **Revert is just inscribing an old version.** No inverse-patch construction; the version's content is unchanged. The new inscriptio's timestamp records when you returned.
- **Rebase doesn't exist.** Git's rebase exists to keep history linear-looking despite the underlying DAG. Lit's histories are linear by construction — they're append-only event logs. What rebase does interactively (squash, fixup, drop, reorder) is just "edit the history file."

Histories are event logs of inscriptiones. Each entry records `(version-hash, timestamp, message)`. For adoption merges (taking a version from another history as-is), provenance — *where did this version come from?* — is recovered from the multiset of inscriptiones across all histories: the earliest entry wins, with deterministic tie-breaking. There are no merge commits; a merge inscription is structurally indistinguishable from a direct one. (A *combining* merge — both sides changed the same text — produces a fresh version through the ordinary conscribe → inscribe path; its merge-ness lives in the message.)

No staging area, narrowly defined. The tabula plays the role of git's staging area *and* git's stash, but it does not have git's index complexity. There is no `add -p`, no per-file selective add, no multi-state index. The tabula is always a single full-document snapshot that replaces on each `conscribe`.

No `lit log`. Git's `git log` iterates over commits via parent pointers and presents them as a list. In lit, the list *is* the data. Viewing a history is direct file iteration; nothing earns its own command.

Inscription messages name states, not deltas. Git commit messages describe *what changed* ("Add feature X," "Fix bug Y") because git presents commits as patches. Lit inscriptions describe *what state was reached* ("first viable draft," "after Anna's notes"). Change-shaped messages still work where they fit (e.g. "merged peira-2"), but they're optional, not the default mold.

The merge algorithm works at the sentence level, not the line level. Git's 3-way merge runs over lines; lit's runs over sentences (atoms). When two histories have both edited a

paragraph, the merge looks at a handful of sentence hashes, not a thousand-line stream. This makes a real difference on prose where lines wrap and re-wrap freely.

The `--fons` flag is the source knob. In git, sources of cherry-picks and merges are positional or implied (HEAD, branch tip). In lit, the source for `lit inscribe` is always `--fons <thing>` (alias `--from`, short `-f`). The target (historia or experiment) is positional.

5.4 Translating common workflows

`git commit -m "message"` — conscribe first, then inscribe:

```
lit conscribe
lit inscribe --all -m "message"
```

(Or: open `lit inscribe` interactively and message each pending revision in the editor — lit's default is one message per intent, finer-grained than a git commit.)

`git checkout -b feature && ... && git checkout main && git merge feature` — work in a peira, then inscribe its tip into historia:

```
(create the experiment — command still being shaped, SPEC §10 #14)
lit inscribe peira-feature        # work happens here
... more inscriptions ...
lit inscribe historia --fons peira-feature
```

`git cherry-pick <commit>` — name the source by hash:

```
lit inscribe --fons <version-hash>
```

`git revert <commit>` — find the version *before* the bad one, inscribe it back into the historia, and exscribe it into your working tree. There's no separate revert primitive because there's no separate operation needed; the version's hash *is* the revert.

```
lit inscribe --fons <good-version-hash>
lit exscribe <good-version-hash>
```

`git stash` — just `lit conscribe`. Switch contexts as needed; come back, keep editing, conscribe again.

`git diff old new` — `lit diff <version-hash> <version-hash>`. Works for any two versions, anywhere, in any history; no parent walk, no common ancestor.

6 Ideas to develop

Stubs for future sections — add freely.

- Why use lit instead of (or alongside) Word track-changes, Google Docs version history, or git for prose
- What sentence-level diff actually looks like in practice
- Citation primitive: referring to atoms by ordinal index across versions ("*Tom Sawyer* atom #123")
- Repetition and quote queries: what falls out of the atomotheca and fasciculotheca for free
- Mode selection: when to mark a file `poetry`, when to keep `prose`
- Migrating an existing manuscript into lit
- Working with markdown / LaTeX where whitespace can be syntactic
- Recovering from accidents: lost working file, corrupted object, mistaken conscribe

A Terminology

Working table of all literium concepts in three registers — **English** (the everyday working term), **Latin**, and **Greek**. All three are first-class; the CLI is intended to accept any of them. `TBD` marks an open question to settle as we go.

Selection of Latin and Greek is guided by **philological and literary fit** — etymology, classical/scholarly usage, coherence with the existing literary vocabulary — not by the first translation in a dictionary. Rationale lives in `HISTORY.md` §11. When a TBD is filled in (or a candidate proposed), update both this table and the relevant entry in HISTORY §11.

A.1 Concepts inherited or adapted from git

For each git concept that survives into literium, both a Latin and a Greek form are wanted alongside the English working term.

Two peer registers (HISTORY §15). Every operation is reachable by both its Latin/Greek human-register name and its plain-English git name; the two are equal peers (neither is the "real" one), and both appear in `lit --help`. The English/git register is also the programming surface an editor binding embeds against. The **CLI peer** column below names the git verb that resolves to each command; the git verbs Literium omits by design get a guidance hint instead (listed after the table).

For the **editorial surface** (collation, variants, review) git has no vocabulary, so the settled English working terms are **standard copyediting / proofreading** terms — *move, insertion, deletion, substitution, split, join; variant, reading, source, review; keep/accept/both/revise* — fixed as the protocol register (PROTOCOL §0.1, HISTORY §16). The Latin/Greek cells for these are **recorded but deferred**: the English column is settled first, the philological pass refines the registers later.

Git term	Latin	Greek	Description
`commit` (noun)	`versio`	TBD	A saved snapshot of the working tree, carrying one identity trio per document plus the manifest (SPEC §8). English working term: *version*.
`commit` (verb) / CLI	`inscribo` (imperative `inscribe`)	`ō`epigraph (ἐπιγράφω)	The act of finalizing a snapshot into a history, and the CLI command that does so (`lit inscribe`). English working term: *inscribe*. **CLI peer `commit`** (HISTORY §15). (HISTORY §8.)
`commit message`	`inscriptio`	`ē`epigraph (ἐπιγραφή)	Free-form text a writer attaches to an inscription; same word also names the per-event entry as a whole. English working term: *inscription*.
`stage` / `stash` (verb) / CLI	`conscribo` (imperative `conscribe`)	TBD	The act of snapshotting the working tree onto the *tabula* (the single-slot scratchpad), and the CLI command that does so (`lit conscribe`). English working term: *conscribe*. **CLI peer `add`** (HISTORY §15). *Redefined* from earlier design (where conscribe produced a version directly); that role moved to *inscribe*.
`staging area` / `index` / `stash`	`tabula`	`pinax` (πίναξ)	Single-slot scratchpad holding the prepared next version. Plays the role of both git's staging area and stash, but in a single-slot full-document replace-on-rewrite shape (no per-hunk selection vocabulary). See *staging area* under "Concepts intentionally not adapted" below for what was rejected and what was kept. English working term: *tabula*.
`patch`	TBD	TBD	An atomic operation, bound to a parent state. **Atom-level:** `insert` / `delete` / `modify` of an atom hash within a fasciculus. **Structural:** `split-paragraph` / `merge-paragraph` (prose) and stanza variants, operating on fasciculi. English working term: *edit*.
`branch` (canonical: `main` /`master`)	`historia`	`historia` (ἱστορία)	The canonical line of development for a project: append-only event log of inscriptiones. Hard-coded name (cannot be renamed), undeletable, default target of `lit inscribe`. One per project. (HISTORY §9.)

Git term	Latin	Greek	Description
`branch` (working: feature/topic)	`experimentum`	`peira` (πεῖρα)	Alternative working line: an event log of inscriptiones parallel to historia. Zero or more per project, user-named, deletable. **CLI peer `branch`** (`lit experiment create|delete|list`, HISTORY §15). *Branch* doesn't survive as a single concept — historia and experimentum are distinct kinds of histories.
`merge`	`contexo` (imperative `contexe`)	`ōsynhyphain` (συνυφαίνω)	Reconciliation of two histories into one. Adoption (fast-forward analogue): `lit inscribe --fons <source>`. Combining (both sides diverged): **`lit contexe <source>`** — Latin *contexere*, "to weave together" (Cicero/Quintilian, of weaving discourse; *texere* is the root of *text*); Greek συνυφαίνω, the same weaving metaphor. The operation's noun is **contextio**; the ledger of variants it leaves is the **apparatus** (§17). Materializes the combined state into the working tree, then ordinary conscribe → inscribe (SPEC §9.4/§9.6). **CLI peer `merge`** (the git verb resolves to `lit contexe`; its git semantics — a merge *commit* — do not, HISTORY §15). HISTORY §9 *The combining merge, designed*, §11.
`pull request` / `merge request`	— (correspondence: *exemplar* out, collation back)	—	Not a platform object and not bolted on: collaboration is correspondence (SPEC §9.7, HISTORY §10). A version travels to another hand as its raw files (the **exemplar**); the returned copy is collated (`lit collate`) against the recorded *basis* and adopted with the ordinary verbs (`lit diff`, `lit inscribe`, `lit contexe --base`).
`cherry-pick`	—	—	No separate primitive. `lit inscribe --fons <version-hash>` adopts a version into the target history (historia unless named) — a whole snapshot, not a delta applied onto the target's state. (HISTORY §8 *What `inscribe` collapses*.)

Git term	Latin	Greek	Description
`revert`	—	—	No separate primitive. `lit inscribe --fons <old-version-hash>` re-inscribes an earlier version into the target history (historia unless named). The hash is unchanged; only the inscriptio's timestamp is new. No inverse-patch is constructed. *Exscribe* independently restores the state into the working tree.
`rebase`	—	—	No analogue and none needed. Histories are linear by construction (event logs); rebase exists in git to keep history linear-looking despite an underlying DAG.
`status`	`status` (already Latin)	TBD	The state of the working tree relative to the latest version on the historia: pending revisions and manifest-level changes (SPEC §10 #13). Git and Literium names coincide.
`checkout` / `restore` (files)	`exscribo` (imperative `exscribe`)	`ōekgraph` (ἐκγράφω)	Materialize a version from a history into the working tree: each document's volumen bytes written at its manifest path (`lit exscribe`; SPEC §8.9). Latin *exscribo*: "to write out, copy out" — the classical verb for copying a document out (Cicero, of records and letters). Completes the *scribere* prefix triad: *con-* gathers onto the tabula, *in-* drives into the history, *ex-* copies back out. **Not a checkout in git's sense**: no pointer, no current position — it writes bytes and nothing else changes. English working term: *exscribe*. **CLI peers `checkout` / `restore`** (HISTORY §15). (HISTORY §8 *Exscribe*.)
`log`	—	—	No separate object, but the **CLI peer `log` resolves to `show`** (`lit log historia` = `lit show historia`): a history *is* a log; viewing it is direct file iteration (HISTORY §8 *Snapshot-primary*, §15).

Git term	Latin	Greek	Description
`diff`	`collatio`	`ēantibol` (ἀντιβολή)	The **collation** of two witnesses of a text: a typed comparison sorting every difference into the classical categories (transpositio/additio/omissio/substitutio + divisio/coniunctio — see below), symmetric over any two version hashes (no parent-chain walk, no common-ancestor discovery). Latin *collatio*, *conferre codices* — collating manuscript witnesses; Greek *antibolē*, ἀντιβάλλειν τὰ ἀντίγραφα, the same. A diff is a two-way collation; a merge is a three-way one (HISTORY §17). Renderings: SPEC §7.4 — the apparatus form (default), `--leiden`, the Unicode marks, `--unified` (legacy). Git and Literium `diff` names coincide.
`repository`	TBD	TBD	The `.lit/` directory and its contents.
`HEAD` / `checkout` (position)	—	—	No pointer analogue. No version is privileged (HISTORY §8 *Snapshot-primary*); there is no current-position pointer to advance or detach. Returning to an earlier state is two independent acts: *inscribe* the old version into the target history (the record) and/or *exscribe* it into the working tree (the files).
`gc`	TBD (candidate: *purgo* — purgare, "to cleanse")	TBD (candidate: *kathairō* καθαίρω — ritual cleansing)	`lit gc` (SPEC §6.10): mark-and-sweep deletion of objects unreachable from any surviving inscriptio, the live tabula, or the contextio ledger. The CLI verb stays the English abbreviation (the term writers meet in every VCS conversation); register forms are candidates only. Git and Literium `gc` names coincide. HISTORY §5 *Garbage collection designed*.
`tag`	TBD	TBD	Named pointer to a specific version. No local-ref `tag` in v1; a permanent named pointer is a Litorium *opus* permalink (guidance hint, below). Placeholder for a future local tag.

Git term	Latin	Greek	Description
`blame`	— (query: *stemma*)	`stemma` (στέμμα)	**CLI peer `blame` resolves to `stemma`**: `lit blame <path> <coord>` gives an atom's first/last inscription — provenance, the blame analogue (SPEC §9.5, HISTORY §15).
`fsck`	—	—	`lit fsck` (SPEC §6.5/§6.6): the object-graph integrity walk. Git and Literium names coincide; register forms unproposed.
`clone`	TBD	TBD	`lit clone <project-id>` — fetch a Litorium project into a new working tree (the platform sync layer). Git and Literium names coincide; the *remote*/*origin* deferral below is superseded (remoting shipped 2026-07-07). Latin/Greek TBD.
`push`	TBD	TBD	`lit push` — sync the historia to Litorium (first push creates the project). Names coincide; Latin/Greek TBD.
`pull`	TBD	TBD	`lit pull` — fast-forward the historia from Litorium (fetch-and-apply in one; there is no separate `fetch`). Names coincide; Latin/Greek TBD.

Concepts intentionally not adapted from git

These have no literium equivalent — the design rejects them rather than renaming them.

- `blob` — replaced by the per-document object set (volumen, spina, atom-spina) plus the repository-deduplicated atoms and fasciculi, all in the type-keyed theca. See HISTORY §5.
- `tree` — replaced by the spina + fasciculus tree (spina holds fasciculus hashes; each fasciculus holds atom hashes).
- `staging area` / `index` (selection vocabulary) — rejected. The git-style index with per-file and per-hunk selective add (`add -p`, multi-state index) does not translate. The tabula plays the staging-area role (and stash role) but in a single-slot, full-document, replace-on-rewrite shape with no selection vocabulary. See HISTORY §8 *Why a tabula at all (revisiting "no staging area")*.
- `init` — rejected; the first `conscribe` creates the repository implicitly. There is no separate init command. See HISTORY §1 *Repository creation: no `lit init`*.
- `merge commit` (as a special version object) — rejected. A merge produces an ordinary inscriptio in the target history that names the source's version. Merge-ness is recoverable from the multiset of inscriptiones across histories, not encoded in the version object. See HISTORY §8 *What was rejected*, §9 *Merge as append*.

— `parent pointer` (on commits) — rejected. Version objects do not chain. History order lives in histories (the event-log files), not in version objects. See HISTORY §8 *Version objects don't chain to parents*.

— `remote` / `origin` — the *concept* (a named remote config) has no CLI command: a synced repo records its single project in `.lit/litorium`, and `push`/`pull`/`clone` are the remote verbs (adopted 2026-07-07; the earlier "deferred to v2+" framing is superseded). `origin` in particular does not survive — there is one project per repo, unnamed.

— `packfile` — no literium equivalent. The local theca stores loose objects only, permanently (SPEC §6.8); bulk transfer happens below literium's model, as substrate-level packs of the remote storage layer. The substrate concept has its name in the *pantheca* register, not here: **sarcina** (Latin: the soldier's marching pack, luggage bundled for a journey; *sarcinas conferre*) — the storage substrate's pack specification. Deliberately not a *fasciculus*-family word: fasciculus is a literium storage object, and a sarcina is not a literium concept at all. HISTORY §5 *Packs resolved out of the core*.

A.2 Literium original concepts

Concepts with no direct git counterpart. The English column is the everyday descriptor used in prose.

English	Latin	Greek	Description
atom	`atomus`	`atomos` (ἄτομος)	The smallest unit with content identity. Sentence in prose, line in poetry. Names both the conceptual unit and the storage object (1:1 by content).
atom (poetry mode)	TBD (candidate: *versus* — etymologically related to *versio*; both from *vertere*, "to turn")	`stichos` (στίχος)	Synonym for atom in poetry mode; the verse line.
revision	TBD	TBD	All the pending edits within a single paragraph or stanza — at most one edit-bearing revision per grex (SPEC §8.5; formatting-only and manifest-level revisions are the defined exceptions). The unit a writer attaches a single message to.

English	Latin	Greek	Description
content form	`tenor` (Latin: "the holding/content of a text"; from *tenere*, "to hold"; cf. *tenor sermonis*, "the tenor of the discourse")	TBD (candidate: διάνοια *diánoia* — "the meaning/sense of a passage", used in patristic exegesis as "the meaning" against "the letter"; the verbatim/content split fits that contrast)	Form of an atom or document **defined by construction** from extraction + segmentation (SPEC §3.4, §4.3): an atom's segment with the atom-edge strip applied; for a document, the reconstruction join. Used for hashing, diffing, segmentation; exactly what spina + fasciculi + atoms reconstruct. Verbatim form is preserved in the volumen. *Renamed from "canonical form"; the older name suggested one chosen variant among several, which is wrong.*
grex	`grex` (Latin: flock, herd, troop, troupe — incl. *grex scaenicus*, "the troupe of a play"; gen. *gregis*)	`thiasos` (θίασος; company, band gathered for common purpose — religious/theatrical, e.g. Bacchic θίασος, Plato's "θίασος of philosophers" in *Phaedrus*)	The abstract structural unit between blank-line separators: a gathering of atoms that forms one paragraph (prose), stanza (poetry), or code-block (code, via *litio*). A subset of the volumen bytes (verbatim, before extraction). Used in all three registers (pattern of *volumen* / *spina* / *fasciculus*); the "gr-" initial doubles as a mnemonic for "group of atoms." Distinct from *fasciculus*: a grex is text in the volumen; a fasciculus is the structural-layer storage dual (an ordered list of atom hashes). HISTORY §11.
paragraph	—	—	The grex, in prose mode. Mode-specific instance term; the abstract concept is *grex* and the storage object is *fasciculus*.
stanza	—	—	The grex, in poetry mode. Mode-specific instance term; the abstract concept is *grex*.
code-block	—	—	The grex, in code mode (provided by the derived project *litio*). Mode-specific instance term; the abstract concept is *grex*. Listed here because grex extensibility to code was a sizing constraint on the term choice (HISTORY §11).

English	Latin	Greek	Description
sub-sentential clause — *reserved candidate, not used in v1*	`membrum`	`kolon` (κῶλον)	Reserved Latin/Greek pair for a *sub-sentential* unit (the rhetorical clause: *kolon/membrum* in Cicero, Demetrius, Quintilian — the canonical doublet at the colon/comma/membrum scale). Considered for the grex slot and **rejected**: the rhetorical pair is sub-sentential, not paragraph-scale, so the doublet's pedigree would have been borrowed at the wrong scale. Held in reserve for a future version that subdivides atoms into clausal sub-units. HISTORY §11.
document verbatim object	`volumen` (Latin: scroll, the rolled-up text)	TBD (candidate: *biblion* βιβλίον, the Greek scroll/book root)	Content-addressed object holding the verbatim bytes of a whole document. Document identity = `sha256( volumen bytes)`. One of six storage object types.
local object pool	`theca` (Latin *theca* is a direct borrowing of the Greek; the bare form coincides)	θήκη (*thēkē*; case, container, repository)	The local content-addressed object pool. All six object types live under it as type-keyed paths (`theca/atom/`, `theca/fasciculus/`, `theca/volumen/`, `theca/spina/`, `theca/atom-spina/`, `theca/version/` — the last with the manifest carve-out, SPEC §6.7); on disk: `.lit/theca/<type>/<aa>/<hash>`. See HISTORY §5 *Storage namespace*.
public atom pool	`corpus` (Latin: body, body of writing — *corpus iuris*, *corpus poetarum*)	TBD (candidate: σῶμα *sōma* — body, used by ancient grammarians for a textual corpus; cf. Plutarch's σῶμα ποιημάτων, "body of poems")	The public, atom-only content-addressed pool at platform (Litorium) scope. Holds atoms only — for citation, dedup, and reverse-lookup; non-atom object types live in per-project storage at the platform layer (HISTORY §5 *Storage namespace*). Addressed at `corpus.litorium.org/atom/<hash>` (URLs not sharded, backend shards internally). v2+ surface; shape settled now.

English	Latin	Greek	Description
atom-typed slice (colloquial)	`atomotheca` (Greek roots ἄτομος + θήκη, via Latin *-theca*; cf. *bibliotheca*, *pinacotheca*)	TBD (candidate: *atomotheke* ἀτομοθήκη, pure-Greek form)	*Colloquial — informal shorthand for the atom-typed objects in the theca; previously framed as a separate pool, now superseded by the unified theca namespace (HISTORY §5* Storage namespace*).*
grex storage object	`fasciculus` (Latin: small bundle; diminutive of *fascis*)	TBD (candidate: *phakellos* φάκελ(λ)ος — single-λ Attic, double-λ Hellenistic; both mean "bundle")	Storage-layer object representing one *grex* (paragraph in prose, stanza in poetry, code-block in code): an ordered list of atom-hash pointers, content-addressed by `sha256` of its bytes. **Not interchangeable with grex** — a grex is text in the volumen, a fasciculus is its pointer-list dual at the structural layer. English working term stays *fasciculus*; if replaced, the English term should be a bookbinding term, not a text term.
fasciculus-typed slice (colloquial)	`fasciculotheca` (Latin *fasciculus* + Greek θήκη; cf. *atomotheca*, *bibliotheca*)	TBD (candidate: *phakellotheke* φακελ(λ)οθήκη, pending Greek decision on *phakellos* spelling)	*Colloquial — informal shorthand for the fasciculus-typed objects in the theca; previously framed as a separate pool, now superseded by the unified theca namespace (HISTORY §5* Storage namespace*).*
structural index	`spina` (Latin: spine, backbone)	TBD	Per-document ordered list of *fasciculus* hashes. The document-level structural index. *Semantic shift:* spina was previously defined as "ordered list of atom hashes with paragraph/stanza breaks"; once the fasciculus layer was introduced, spina was reassigned one layer up — see HISTORY §11.2.

English	Latin	Greek	Description
flat atom index	TBD (working term: *atom-spina*)	TBD	Per-document content-addressed object: flat list of every atom hash in document order. Normative storage (one of the six theca object types). Two roles: ordinal-atom addressing for citation ("*Tom Sawyer* atom #123"), and `sha256(atom-spina)` is the third hash of a document's identity trio for O(1) edit-category classification. Latin and Greek may inherit from *spina* once that's settled.
variant *(git: conflict)*	`varia lectio` / `discrepantia`	`ōdiaphnia` (διαφωνία)	Two witnesses reading the same slot differently — a **substitutio** both hands touched (the only genuine adjudication a merge leaves; moves and adds/omits auto-resolve). *Not* a git-style conflict: no failure state, no markers, no ours/theirs; the trunk stays a whole readable text and the readings sit in the **apparatus** beside it (HISTORY §17). The classical *varia lectio* (variant reading); *diaphōnia* διαφωνία, ancient scholarship's word for witnesses disagreeing. Recorded in `.lit/contextio` (the apparatus); worked by *recensio*. SPEC §7.4/§9.6.
review record / the changes	`apparatus` (*apparatus criticus*)	—	The record of open variants and their resolutions — the renamed contextio ledger (`.lit/contextio`). Rendered as the critical-apparatus foot to the reading text: `<locus> <lemma> ] <reading> <siglum> : <reading> <siglum>`, sigla = the hands (provenance, = *stemma*). Derived, not a stored object (atoms persist; the merge is an ordinary inscriptio). HISTORY §17.

English	Latin	Greek	Description
review	`recensio`	—	Working the variants to establish the text — the textual critic's term (*recensere*, to review the witnesses). The merge workflow: descend document → grex → atom → word, resolving each variant **keep / accept / both / revise**, in any order (variants are independent). HISTORY §17.
revise / revision	`emendatio` (verb `emendo`)	—	Writing a *new* reading at a variant, informed by both witnesses — the fourth resolution move, the editor's synthesis. Latin *emendo*, to correct/improve; reserved earlier for the merge-request concept, now at home. HISTORY §17.
move	`transpositio`	—	A move: the same atom or grex at a new position (same hash relocated). Auto-resolved in a merge unless both hands moved it differently. One of the collation categories (SPEC §7.4). Git cannot cheaply name it.
insertion / deletion	`additio` / `omissio`	—	An atom present on one side only (additio) or dropped (omissio) — collation categories, resolved by keep-or-drop.
substitution	`substitutio`	—	An atom whose content changed: a different hash in the same aligned slot. The category git structurally cannot name (to git a modify is delete-plus-add); literium names it via the unchanged-atom anchors. The only category that becomes a *variant* when both hands touch it. SPEC §7.4.
split / join	`divisio` / `coniunctio`	— (candidate: *diairesis* διαίρεσις for divisio)	The grex-structure pair: a paragraph split into two (divisio) or merged into one (coniunctio) with its atoms unchanged — the §7.4 ~ regroup lines. The structural axis beside the four content categories.

English	Latin	Greek	Description
provenance query	`stemma`	`stemma` (στέμμα)	The `lit stemma` verb (SPEC §9.5): a version's life-path through the inscriptio multiset (origin marked), and atom-level first/last-inscribed via coordinates. στέμμα — wreath, garland; in textual criticism the *stemma codicum*, the genealogical record of a text's descent through its witnesses. Borrowed into Latin unchanged, so the two registers coincide, like *historia*. HISTORY §9 *Provenance surfaced*.
edit category	TBD	TBD	Per-document classification of how a document differs between two versions: *formatting* (only volumen bytes change), *structural* (spina or fasciculi change; flat atom sequence unchanged), *substantial* (atom hashes change), *re-segmentation* (same bytes, different mode). SPEC §7.2.
formatting edit	TBD	TBD	Edit where only volumen bytes change; spina, fasciculi, and atoms are unchanged. No first-class operation; observable as a volumen byte delta.
structural edit	TBD	TBD	Edit where spina or fasciculi change but the flat atom sequence is unchanged (paragraph/stanza split or merge). An atom *moved* between grex-instances reorders the flat sequence and classifies substantial; diff display presents it as a move.
substantial edit	TBD	TBD	Edit where atom hashes change (insert, delete, modify of atom content).
re-segmentation edit	TBD	TBD	Edit category for a version pair whose volumen is unchanged while spina and atom-spina differ: the same bytes read under a different mode (attributes-file change; SPEC §1.1, §7.2). Only reachable through a mode change in v1. Latin and Greek: open (candidates wanted alongside the other three categories).

English	Latin	Greek	Description
mode	TBD	TBD	The kind of text being versioned: prose or poetry. Set per-file via the attributes file.
prose (mode)	`prosa`	TBD	Mode where atoms are sentences segmented by UAX-29.
poetry (mode)	TBD	`ēpoisis` (ποίησις)	Mode where atoms are lines segmented at LF.
attributes file	TBD	TBD	Repo-root file mapping path globs to modes.
ignore file	TBD	TBD	Repo-root file (`.litignore`, with optional sub-directory copies) listing path patterns excluded from working-tree scans. Syntax and semantics adopted verbatim from `.gitignore`.
manifest	TBD (candidate: *dispositio* — arrangement, layout; classical rhetorical term for the arrangement of material; from *disponere*, "to set in order"; alt: *manifestum*, the etymological original)	TBD (candidate: *taxis* τάξις — arrangement, order; the Greek counterpart of *dispositio*)	Field of a version object recording where each document lives in the working tree at conscribe time. Part of the version object's data but MUST NOT contribute to the version hash (HISTORY §8 *Manifest is in the version object, excluded from the hash*; SPEC §8). The exclusion is what lets elevation and project merge rewrite paths without disturbing version IDs. *Manifest* in the cargo-listing sense (Latin *manifestus* via shipping-manifest usage): a listing of contents and where each item is stowed. Renamed from *path mapping* (working term); vocabulary repurposed once Litorium retired its own use of *manifest* in favor of *historia* directly.

English	Latin	Greek	Description
elevation	TBD (candidate: *amplificatio* — enlargement, widening of scope; classical rhetorical term, fits the project-scope-grows reading)	TBD (candidate: ἐπαύξησις *epaúxēsis* — increase, growth; cf. *amplificatio*)	One-way operation that moves a project's root upward to encompass its parent directory (HISTORY §1 *Project scope expands by elevation*; SPEC §1.4). Triggered by invoking `lit conscribe` from an ancestor of an existing `.lit/`. The manifest is rewritten relative to the new root; version hashes survive because the manifest is unhashed (HISTORY §8). Elevation of multiple descendant projects at once is a project merge (§9).
writing tablet	`tabula`	`pinax` (πίναξ)	Single-slot scratchpad holding the prepared next version, between `lit conscribe` (working tree → tabula) and `lit inscribe` (tabula → history). Roles of both git's staging area and stash collapsed into one full-document replace-on-rewrite primitive. See HISTORY §8 *Tabula*.
canonical history	`historia`	`historia` (ἱστορία)	Per-project canonical line of development: append-only event log of inscriptiones. Hard-coded name (every repository has exactly one historia, named *historia*; cannot be renamed), undeletable, default target of `lit inscribe`. The Litorium-publishable artefact for a project is the export of its historia. Latin and Greek registers coincide on the bare form (Latin *historia* is a direct borrowing of Greek ἱστορία). See HISTORY §9.

English	Latin	Greek	Description
working history (alt)	`experimentum`	`peira` (πεῖρα)	Alternative working line alongside historia: an event log of inscriptiones, deletable, user-named (e.g. `peira-1`, `editor-pass`, `anna-notes`). *Peira* (Greek: trial, attempt — classical, attested in Homer and Thucydides) was preferred over candidates δοκιμή *dokime* (test/examination) and Latin *temptamen* (attempt) for register fit. Zero or more per project. See HISTORY §9.
inscribe (verb)	`inscribo` (imperative `inscribe`)	`ōepigraph` (ἐπιγράφω)	Verb / CLI command for finalizing a snapshot into a history (§8). Latin *inscribo*: "to write into / inscribe upon"; classical idiom *in marmore inscriptum*. Greek ἐπιγράφω: the verb that produces an *epigraphē*. Morphologically parallel to *conscribo*; the prefix-axis (*con-* gather vs. *in-* drive into) encodes ephemeral collection vs. permanent placement. The wax/stone contrast with *tabula* is a real Roman cultural pairing (a *tabula cerata* was an erasable wax tablet; the lasting opposite was inscription on stone or bronze). See HISTORY §8 *Inscribe*.
inscription (entry / message)	`inscriptio`	`ēepigraph` (ἐπιγραφή)	Per-event record produced by an inscribe call: an entry in a history carrying a version hash, timestamp, and message (§8). Same word covers both the entry as a whole and the message text within it. See HISTORY §8 *Inscriptio*.

English	Latin	Greek	Description
incise (verb) — *reserved candidate*	`incido` (imperative `incide`)	`ōencharass` (ἐγχαράσσω)	Reserved alternative verb for inscribing into historia specifically, distinguishing "permanent" historia inscription from "working" experiment inscription. Latin *incido*: "to cut into, engrave" (used classically of stone and bronze inscriptions). Greek ἐγχαράσσω: from χαράσσω, the root of χαρακτήρ (the engraved mark / character). Considered and rejected for v1 (one primitive for state-into-history is principled; historia's specialness lives in the data model, not the verb). Reserved as a candidate if real use shows historia inscriptions feel under-marked. See HISTORY §8 *What was rejected*.
source (flag for `inscribe`)	`fons`	—	Latin *fons* (source, spring; *fons et origo*). Used as `lit inscribe --fons <source>` (HISTORY §8 *Source flag*). English alias `--from`; short form `-f`. Greek register intentionally unset — flag names are connective tissue and don't all need three registers.
exemplar (sent copy)	`exemplar`	TBD (candidates, unvetted: ἀντίγραφον *antigraphon* — the copy made or sent; παράδειγμα *paradeigma*)	The archive that carries a version's raw files to another hand (SPEC §9.7): documents at their manifest paths plus the three-line `.exemplar` metadata file (marker, *basis*, *segmentatio*). Latin *exemplar*: in manuscript culture, precisely the copy one transcribes from. Produced by `lit exemplar <source>`; the recipient's return trip is `lit exemplar` with no source in the unpacked directory. A convenience form, never identity-bearing. (HISTORY §10.)

English	Latin	Greek	Description
collate (verb) / CLI	`confero` (imperative `confer`) — unvetted	TBD (candidate, unvetted: ἀντιβάλλω *antiballō*; noun ἀντιβολή *antibolē* — the technical verb for collating a copy against its exemplar)	Ingest a returned exemplar (`lit collate <file>`): verify the `.exemplar` metadata, guard the working tree (§8.9), write the archive's documents with exscribe's document-set fidelity, and report the *basis* for review. English *collate* from Latin *collatum* (supine of *conferre*), the textual critic's own term for comparing a witness against its exemplar. (SPEC §9.7.)

A.3 Maintaining this table

The source of truth for this table is `guide-tex/terminology.tex` (rendered both as this appendix and as the standalone `TERMINOLOGY.md`; regenerate both after editing).

— A term gets decided → fill in the cell, remove `TBD`, and update the matching entry in `HISTORY.md` §11.

— A candidate is proposed but not yet final → keep `TBD` in the cell, append the candidate in parentheses with reasoning.

— A new concept comes up → add a row to the appropriate section. If it spans both (e.g. a git concept we're now extending), put it in §A.1 and note the extension in the description.

— When in doubt about literary fit, prefer the term with **classical / philological precedent** over the term with **Latinate-sounding novelty**.