Docbook futon4

Joe Corneli

2026-01-05

Contents

1 Quickstart

- Source: dev/bootstrap2.el

1.1 Context

- Start Futon1 locally (‘`http://localhost:8080`‘).
- ‘M-x load-file‘ ‘arxana/dev/bootstrap.el‘
- ‘M-x arxana-build‘ (use ‘C-u‘ to force tangling).
- ‘(setq futon4-base-url “`http://localhost:8080/api/alpha`”)‘
- ‘(setq futon4-enable-sync t)‘
- Register buffers: ‘(make-current-buffer-into-article “Demo”)‘
- Capture scholia: ‘(scholium “demo/link” “Note” ’((“Demo”)))‘

2 Overview

2.1 Context

The classic Arxana client now talks to Futon1 through a small set of Elisp helpers. This document captures the supported operations, the Elisp entry points you should call, and the matching HTTP routes + curl snippets you can run to verify what Emacs just did.

All commands assume the Futon API is reachable at ‘futon4-base-url‘ (default ‘`http://localhost:8080/api/alpha`‘). Set ‘X-Profile‘ headers if you work inside multiple plexus profiles.

3 Logical model

3.1 Context

Arxana manages a single logical hypergraph. The core primitives are:

Nema Universal node record with stable id, labels, optional endpoints, and payload.

Article A nema labeled :label/article, holding text and core metadata.

Metadata scholium A nema paired 11 with an article, caching backlinks, labels, and indexes.

Event (hyperedge) A nema representing a relation or annotation, with :hx/type and N endpoints.

Plexus A nema that names a working set or profile and carries configuration.

Classic Arxana entities, relations, and scholia embed cleanly:

- Entities article nemas with derived identifiers.
- Relations 2-end events (:hx/*) between nemas.
- Scholia events and/or metadata nemas attached to articles.
- Inclusion / clusion / provenance multi-end events with appropriate :role values.

3.1.1 Storage substrates

All storage backends implement the same logical model.

1. In-memory storage of one form or another
   - Historically, nemas, articles, events, and plexuses were held in hash tables / HONEY-style networks.
   - This is the reference representation; other substrates hydrate/dehydrate it.
2. Document-based substrates
   - Literate sources (e.g., LaTeX, Org, Markdown, TEI) could be parsed into nemas and events.
   - Arxana can in principle and, increasingly, in practice both read **and write** these formats, keeping ids and links stable across round-trips.
3. Historical: Relational substrate
   - Common Lisp provided a bridge to SQL that encode relations in tables.

- Semantics are those of a structured triple store: subjects, predicates, objects, plus spans and provenance.
- Suitable for durability, multi-user access, and heavy queries.

4. Possible future: Triple-store substrate

   - Direct RDF/quad representation of the same model not implemented but it could be if we want it.
   - Again, we could in principle read and write external knowledge graphs (and integrate SPARQL tooling).

5. Current reality: EAV in Clojure (Datascript + XTDB)

   Instead of going the triple route, all storage backends in the current beta generation converge on an open **entityattributevalue (EAV)** model. Each recordwhether an article, event, or plexusis represented as a set of attributevalue pairs attached to a unique entity id. This design keeps the schema **open**, so new attributes or link types can appear dynamically without migration.

   The canonical implementation uses:

   - **Datascript** for in-memory, client-side operation.
     - Pure Clojure; lightweight and immutable.
     - Supports live queries, undo/redo, and fast incremental updates.
     - Ideal for interactive use inside Emacs or JVM clients.
   - **XTDB** for durable, time-traveling storage.
     - Schemaless documents mirror the same EAV keys verbatim.
     - Transactions are append-only; every version of the graph is queryable.
     - Perfect for provenance, journaling, and long-term archives.

   Together these give Arxana an **open-schema substrate**: a graph database where nemas, events, and plexuses are all entities, and attributes such as ‘:hx/type‘, ‘:article/text‘, or ‘:plexus/members‘ are just first-class keys. Queries can run over the generic EAV view, or use higher-level helpers that expose structured projections (e.g., articles, hyperedges, plexuses).

   > **Design posture:** Start open (EAV); specialize only when scale or semantics demand it. > Structured viewsrelational tables, triple

stores, materialized indexescan be layered on > demand without altering the underlying data model.

This approach keeps computational properties stable: query complexity and transaction semantics stay identical whether data are viewed as triples, hash maps, or SQL rows. It also makes cross-substrate replication trivial: each backend stores the same ‘(entity attribute value)‘ facts, differing only in query language and persistence model.

3.1.2 Emacs client (Arxana classic)

The Emacs Lisp implementation in this repository provides:

- Interactive creation and editing of articles and scholia.
- Browsing modes over the in-memory graph.
- Operations for inclusion, derivation, and provenance as events.
- Import/export for specific document formats (historically LaTeX/Org).

Internally, it maintains the logical model in memory and (optionally) syncs with external substrates (files, SQL backend). nil

3.2 Emacs client (Arxana classic)

3.3 Context

The Emacs Lisp implementation in this repository provides:

- Interactive creation and editing of articles and scholia.
- Browsing modes over the in-memory graph.
- Operations for inclusion, derivation, and provenance as events.
- Import/export for specific document formats (historically LaTeX/Org).

Internally, it maintains the logical model in memory and (optionally) syncs with external substrates (files, SQL backend). nil

3.4 Storage substrates

3.5 Context

All storage backends implement the same logical model.

1. In-memory storage of one form or another

   - Historically, nemas, articles, events, and plexuses were held in hash tables / HONEY-style networks.
   - This is the reference representation; other substrates hydrate/dehydrate it.

2. Document-based substrates

   - Literate sources (e.g., LaTeX, Org, Markdown, TEI) could be parsed into nemas and events.
   - Arxana can in principle and, increasingly, in practice both read **and write** these formats, keeping ids and links stable across round-trips.

3. Historical: Relational substrate

   - Common Lisp provided a bridge to SQL that encode relations in tables.
   - Semantics are those of a structured triple store: subjects, predicates, objects, plus spans and provenance.
   - Suitable for durability, multi-user access, and heavy queries.

4. Possible future: Triple-store substrate

   - Direct RDF/quad representation of the same model not implemented but it could be if we want it.
   - Again, we could in principle read and write external knowledge graphs (and integrate SPARQL tooling).

5. Current reality: EAV in Clojure (Datascript + XTDB)

   Instead of going the triple route, all storage backends in the current beta generation converge on an open **entityattributevalue (EAV)** model. Each recordwhether an article, event, or plexusis represented as a set of attributevalue pairs attached to a unique entity id. This

design keeps the schema **open**, so new attributes or link types can appear dynamically without migration.

The canonical implementation uses:

- **Datascript** for in-memory, client-side operation.
  - Pure Clojure; lightweight and immutable.
  - Supports live queries, undo/redo, and fast incremental updates.
  - Ideal for interactive use inside Emacs or JVM clients.
- **XTDB** for durable, time-traveling storage.
  - Schemaless documents mirror the same EAV keys verbatim.
  - Transactions are append-only; every version of the graph is queryable.
  - Perfect for provenance, journaling, and long-term archives.

Together these give Arxana an **open-schema substrate**: a graph database where nemas, events, and plexuses are all entities, and attributes such as ‘:hx/type‘, ‘:article/text‘, or ‘:plexus/members‘ are just first-class keys. Queries can run over the generic EAV view, or use higher-level helpers that expose structured projections (e.g., articles, hyperedges, plexuses).

> **Design posture:** Start open (EAV); specialize only when scale or semantics demand it. > Structured viewsrelational tables, triple stores, materialized indexescan be layered on > demand without altering the underlying data model.

This approach keeps computational properties stable: query complexity and transaction semantics stay identical whether data are viewed as triples, hash maps, or SQL rows. It also makes cross-substrate replication trivial: each backend stores the same ‘(entity attribute value)‘ facts, differing only in query language and persistence model.

3.5.1 Current reality: EAV in Clojure (Datascript + XTDB)

3.6 Context

Instead of going the triple route, all storage backends in the current beta generation converge on an open **entityattributevalue (EAV)** model. Each recordwhether an article, event, or plexusis represented as a set of attribute-value pairs attached to a unique entity id. This design keeps the schema

open, so new attributes or link types can appear dynamically without migration.

The canonical implementation uses:

- **Datascript** for in-memory, client-side operation.
  - Pure Clojure; lightweight and immutable.
  - Supports live queries, undo/redo, and fast incremental updates.
  - Ideal for interactive use inside Emacs or JVM clients.
- **XTDB** for durable, time-traveling storage.
  - Schemaless documents mirror the same EAV keys verbatim.
  - Transactions are append-only; every version of the graph is queryable.
  - Perfect for provenance, journaling, and long-term archives.

Together these give Arxana an **open-schema substrate**: a graph database where nemas, events, and plexuses are all entities, and attributes such as ‘:hx/type‘, ‘:article/text‘, or ‘:plexus/members‘ are just first-class keys. Queries can run over the generic EAV view, or use higher-level helpers that expose structured projections (e.g., articles, hyperedges, plexuses).

> **Design posture:** Start open (EAV); specialize only when scale or semantics demand it. > Structured viewsrelational tables, triple stores, materialized indexescan be layered on > demand without altering the underlying data model.

This approach keeps computational properties stable: query complexity and transaction semantics stay identical whether data are viewed as triples, hash maps, or SQL rows. It also makes cross-substrate replication trivial: each backend stores the same ‘(entity attribute value)‘ facts, differing only in query language and persistence model.

3.6.1 Document-based substrates

3.7 Context

- Literate sources (e.g., LaTeX, Org, Markdown, TEI) could be parsed into nemas and events.
- Arxana can in principle and, increasingly, in practice both read **and write** these formats, keeping ids and links stable across round-trips.

3.7.1 Historical: Relational substrate

3.8 Context

- Common Lisp provided a bridge to SQL that encode relations in tables.
- Semantics are those of a structured triple store: subjects, predicates, objects, plus spans and provenance.
- Suitable for durability, multi-user access, and heavy queries.

3.8.1 In-memory storage of one form or another

3.9 Context

- Historically, nemas, articles, events, and plexuses were held in hash tables / HONEY-style networks.
- This is the reference representation; other substrates hydrate/dehydrate it.

3.9.1 Possible future: Triple-store substrate

3.10 Context

- Direct RDF/quad representation of the same model not implemented but it could be if we want it.
- Again, we could in principle read and write external knowledge graphs (and integrate SPARQL tooling).

4 Article lifecycle

4.1 Context

This module defines the in-memory article lifecycle for the Emacs client. The core state is ‘arxana-article–table‘, a hash table keyed by article name. Each entry is a plist that minimally carries ‘:name‘ and ‘:text‘. This table is the local working store that other subsystems (scholia, relations, sync) build on. Think of it as the authoritative buffer-backed cache before anything is pushed to Futon/XTDB.

‘make-current-buffer-into-article‘ is the registration step. It captures the current buffer text, stores it under the given name in the table, and sets ‘name-of-current-article‘ buffer-locally. That local name is how downstream

code knows which article a buffer represents without having to re-parse headers.

The path cache ‘arxana-article-path-cache‘ exists to normalize and remember canonical filesystem paths for articles. ‘arxana-article–canonical-path‘ calls into ‘futon4–canonical-path‘ when available, so every lookup uses the same normalized path. This avoids subtle mismatches when the same article is referenced via different path spellings.

This file uses ‘unless (fboundp ...)‘ wrappers so it can provide fallback definitions in environments where the classic Arxana functions are missing. In full Futon builds, these names may already be defined elsewhere; the guard lets this module fill gaps without overriding canonical implementations.

‘get-article‘ returns the stored plist, and ‘sch-plain-text‘ pulls ‘:text‘ from that plist. The pairing is deliberate: higher-level scholium routines can operate on article objects without knowing their internal structure, and the plain-text accessor stays small and easy to replace if the article structure changes.

Finally, ‘arxana-article–metadata-envelope‘ builds a metadata wrapper around a doc if ‘metadata-article‘ and ‘scholium-text‘ are available. It collects the scholium metadata and optional labels into an alist under the ‘metadata‘ key. This envelope is used when emitting structured payloads for storage or export, so callers do not need to stitch metadata together themselves.

5 Browsing & relations

- (no entry content yet)

5.1 Catalog & label menus

- Source: dev/arxana-browse.el

5.2 Context

The Part VI menus still work via modern commands:

5.3 Commands & context

- Source: dev/arxana-relations.el

5.4 Context

Finally we wire up interactive commands (‘arxana-relations-show-ego‘, etc.), store the last Futon context, and expose refresh/provenance/copy-id helpers.

5.5 Formatting helpers

- Source: dev/arxana-relations.el

5.6 Context

Before we render anything we normalize type labels, build friendly names, and provide helpers for inserting sections into the buffer.

5.7 Legacy navigation commands

5.8 Context

5.9 Link-following & history

5.10 Context

5.11 Context

Arxana links focus on preserving intent rather than storing every computed edge. Strategies describe how to find links; anchors and scholia capture the human signal around them. When strategies are persisted, links can be recomputed as documents evolve without losing the provenance of the annotations that motivated them.

5.12 Link tiers

- Tier 0 (Ephemeral): computed links only, no persistence.
- Tier 1 (Strategy): persist finder rules; links can be recomputed.
- Tier 2 (Anchored): persist anchored regions for immutable or historic text.

5.13 Core workflows

5.13.1 Check status

5.13.2 Create or load a strategy

The demo uses the futon4 repo and a definition-based finder. It is safe to run multiple times because the strategy id is deterministic.

5.13.3 Create a resilient scholium

1. Select a region in a buffer.
2. Run M-x arxana-links-demo-create-scholium.
3. Enter the annotation text.

Anchors use content hashes and surrounding context so they survive minor edits.

5.13.4 Capture a surface form

1. Select a phrase.
2. Run M-x arxana-links-capture-surface-form.
3. Provide the concept id.

Surface forms inform finders and auto-linking later on.

5.13.5 Verify anchors

This runs the three-step re-finding algorithm (offset check, context search, fuzzy context).

5.13.6 Embedding neighbors for patterns

Embedding neighbors require a cached embedding space persisted as an embedding cache entity.

5.14 Persistence entities

- Strategy: arxana/link-strategy
- Voiced Link: arxana/voiced-link
- Surface Form: arxana/surface-form
- Scholium: arxana/scholium
- Embedding Cache: arxana/embedding-cache

Use arxana-links-make-* + arxana-links-persist-* to create and save these records. Use the arxana-links-load-* helpers to retrieve them. Payloads are stored in Futon entities with the full record serialized into the source field so they round-trip through the existing API.

5.15 Testing

5.15.1 Unit tests (Emacs only)

5.15.2 Integration tests (real Futon server)

The links test suite includes integration tests tagged :futon that run only when a Futon server responds to /types.

Set the base URL with either futon4-base-url or $\text{FUTON4}_{\text{BASEURL}}$ if the server is not on `http://localhost:8080`.

When the server is reachable, the tests exercise persistence for strategies, voiced links, surface forms, scholia, embedding caches, and demo strategy creation.

5.16 Relation buffers in context

- Source: dev/arxana-relations.el

5.17 Context

The relation helpers render Futon adjacency inside Emacs so we no longer need to shell out to curl. Each buffer is a minor mode with refresh, provenance, and copy-id bindings.

5.18 Rendering Futon responses

- Source: dev/arxana-relations.el

5.19 Context

Each renderer translates a Futon response (‘/ego‘, ‘/cooccur‘, ‘/tail‘) into a human-readable set of lines.

5.20 Tabulated UI & commands

- Source: dev/arxana-xtdb-browse.el

5.21 Context

The major mode uses ‘tabulated-list-mode‘ with shortcuts for refreshing, changing limits, inspecting raw payloads, and opening relation buffers.

5.22 XTDB browsing shims

- Source: dev/arxana-xtdb-browse.el

5.23 Context

The XTDB browser exposes ‘/tail‘ data via a tabulated list and lets contributors inspect source/target entities, jump into relation buffers, or tweak the limit.

5.24 XTDB helpers

- Source: dev/arxana-xtdb-browse.el

5.25 Context

Normalize Futon rows into tabulated entries and provide entity inspectors.

6 Compatibility & test support

- Source: dev/arxana-compat.el

6.1 Context

- The compatibility shim ensures modern ‘get-article‘ lookups hit the in-memory table before falling back to the legacy tangle.
- Test suites load the tangled code by way of the shared helper so they work whether invoked from the repo root or inside ‘arxana/‘.

7 Contributor guide (embedded)

- Source: docs/contributor-guide.md

7.1 Context

Phase-aware workflow

Prototype 1 keeps the fast-moving enhancements inside ‘dev/‘, while canonical docs live in XTDB. Filesystem doc snapshots are optional and should be treated as temporary unless explicitly promoted.

1. Load the harness (‘M-x arxana-load‘ or ‘./dev/run-tests.sh‘) so the dev

modules share the same Emacs session as your edits.

1. Make iterative changes inside ‘dev/*.el‘ or ‘test/*.el‘.

2. Record the change in the living plan (‘docs/reanimation-plan.org‘) and

the relevant XTDB doc entry whenever a milestone flips state.

1. If you create a filesystem doc snapshot, tag it as temporary or

archival and note the XTDB doc id it mirrors.

Dev code XTDB docs

To keep code and docs aligned:

- Keep module header comments accurate and point to the XTDB doc id (or

the temporary snapshot) that describes the behavior.

- When adding a new entry point, add a short paragraph to the relevant

XTDB doc entry explaining how it fits the plan.

- Track drift in ‘dev/docs-backlog.org‘ so missing doc updates are

easy to spot.

Pattern library contributions

The pattern-sync section in the README describes the day-to-day loop. For contributor clarity:

1. ‘M-x arxana-patterns-open‘ enter the slug (e.g., ‘mojo/center‘).

2. Edit the Org buffer the command opens; each stanza includes the Futon

component ids.

1. ‘C-c C-s‘ (‘arxana-patterns-save‘) to push text edits back to Futon.

2. Run ‘dev/run-tests.sh‘ (or the single ‘arxana-patterns‘ test when it

arrives) before opening a PR.

This guarantees design patterns round-trip through XTDB and that reviewers can diff both the Org edits and the corresponding graph changes.

QA / regression expectations

- Always run ‘dev/run-tests.sh‘. It loads the dev modules and executes

every suite in ‘test/‘.

- If you change inclusion/integration flows, also run the scripted Org

walkthrough (see README Scripted verification) so the Scholia display and relation buffers stay consistent with Futon.

- For storage changes, use the ‘docs/storage-bridge.org‘ curl snippets to

validate against a real Futon instance.

Open limitations (next steps)

- Generalized pattern-language editing (treating an entire pattern set as a

single editable language) is still experimental. Todays tooling focuses on individual pattern slugs; the multi-pattern workflow will ship in a later milestone.

- Orggraph semantic mirroring is still one-way. We can import/export Org

trees, but live edits inside Org buffers are not yet reflected into the EAV stores automatically.

- In terminal Emacs, ‘ffplay‘ can still receive keypresses from the TTY, so

media hotkeys (like ‘m‘ for mute/volume) can affect playback while the browser is focused. Restart Emacs after changing playback flags, or use a GUI session to avoid this limitation.

Track both of these gaps in README/plan updates so contributors know where help is welcome.

8 Downtime handling

8.1 Context

- When ‘futon4-enable-sync‘ is nil or Futon is unreachable, the helpers refuse to send traffic and stash a diagnostic plist in ‘arxana-store-last-error‘. Inspect it with ‘(message “%S” arxana-store-last-error)‘.
- Calls never signal errors back to users; they log with ‘message‘ and return ‘nil‘ so interactive commands can degrade gracefully.

9 Entry points

9.1 Context

YYYY Elisp helper Description Futon route Verification

`arxana-store-ensure-article` Ensure an entity exists for an article name/path. `POST /entity` `curl -s -X POST "$BASE/entity" -H 'Content-Type: application/json' -d '{"id":"...","name":"...","type":"arxana/article"}'`

`arxana-store-upsert-scholium` Mirror a scholium-style relation between two articles. `POST /relation` `curl -s -X POST "$BASE/relation" -H 'Content-Type: application/json' -d '{"type":"arxana/scholium","src":"...","dst":"..."}'`

`arxana-store-post-hyperedge` Publish multi-end inclusion / derivation hyperedges with spans + provenance. `POST /hyperedge` `curl -s -X POST "$BASE/hyperedge" -H 'Content-Type: application/json' -d '{"type":"arxana/transclusion","hx/type":":hx/transclusion","hx/endpoints":[{"role":"...)","scholium/type":"derives-from"}}'`

`arxana-store-save-snapshot` Request an XTDB snapshot (`all` history vs `latest` revisions). `POST /snapshot` `curl -s -X POST "$BASE/snapshot" -H 'Content-Type: application/json' -d '{"scope":"all","label":"nightly"}'`

`arxana-store-restore-snapshot` Rehydrate Futon/XTDB from a recorded snapshot id. `POST /snapshot/restore` `curl -s -X POST "$BASE/snapshot/restore" -H 'Content-Type: application/json' -d '{"action":"restore","scope":"all","snapshot/id":"xtdb:snapshot/2024-11-14"}'`

`arxana-export-org-directory` Export the current article table as `.org` files + manifest. (local) `M-x arxana-export-org-directory` pick destination & scope.

`arxana-patterns-open` Fetch Futon pattern library entries (and edit them) inside Emacs. (local) `M-x arxana-patterns-open` enter pattern slug, edit headings, `C-c C-s` to sync.

`MANIFEST.org` now captures a sortable article table, per-article hyperedge summaries (inclusion/transclusion passages), and a label index so the iPod-style browser can jump through categories offline. Those sections live alongside the snapshot metadata, so bundles carry both provenance and enough navigation hints to explore the export without Emacs.

YYYY `arxana-store-fetch-entity` Fetch the latest copy of a Futon entity. `GET /entity/:id` =curl -s "$BASE/entity/<url-encoded-id>" jq ''=

`arxana-store-entity-history` Inspect prior versions of an entity. `GET /entities/history/:id?limit=n` =curl -s "$BASE/entities/history/<id>?limit=5" jq '.versions'=

`arxana-store-ego` List relations touching a given article (wrapper around Futon's `/ego`). `GET /ego/:name` =curl -s "$BASE/ego/Demo%20Article" jq '.ego.links'=

`arxana-store-cooccur` Fetch entities that co-occur with the named article. `GET /cooccur/:name` =curl -s "$BASE/cooccur/Demo%20Article" jq '.rows'=

`arxana-store-tail` Show the latest Futon relations (recent tail). `GET /tail?limit=n` =curl -s "$BASE/tail?limit=5" jq '.relations'=

`arxana-store-sync-current-article` Force-sync the current buffer's metadata and status. `POST /entity` =curl -s "$BASE/entity/<id>" jq '.entity.props'= to confirm the new `metadata` / `status` values.

Replace `$BASE` with `http://localhost:8080/api/alpha` (or your chosen host).

10 Inclusion / derivation UX

10.1 Context

10.2 Context

This module is a small guardrail for include and transclude workflows. The problem it solves is silent failure: if you call 'include-article' or 'transclude-article' with an article name that has not been registered, classic behavior can end up inserting 'nil' or doing nothing without explanation. The guards force a clear, actionable error instead of a confusing empty insert.

The core helper is ‘arxana-inclusion–require-article‘. It calls a lookup function (typically ‘get-article‘) and raises a ‘user-error‘ if the article is missing. The error message tells you exactly what to do: go to the source buffer and run ‘make-current-buffer-into-article‘ to register it. This is the main user-facing improvement.

‘include-article‘ and ‘transclude-article‘ are both defined here as fallbacks when the canonical definitions are not available. They are similar but not identical. ‘include-article‘ inserts the plain text of the named article into the current buffer. ‘transclude-article‘ also verifies the **target** article (the buffer you are in) has been registered, so cross-article inclusion remains consistent and traceable.

When ‘arxana-tangled‘ is loaded, the module uses advice-like wrapping: it captures the existing function definition and rebinds it to a guarded wrapper. This lets the guard apply even when a different implementation is already present, without changing that implementation. The result is a smoother UX: missing registration is caught early, with a message that tells you how to fix it.

11 Known limitations

11.1 Context

- Pattern-language wide editing (future milestone).
- Orggraph semantic mirroring pending.

12 Org imports / exports & snapshots

- (no entry content yet)

12.1 Directory importer

- Source: dev/arxana-import.el

12.2 Context

12.3 Export helpers

12.4 Context

12.5 Context

Export helpers produce readable manuals from the docbook store. The preferred targets are:

- Org (canonical): a single merged Org file for a book.
- PDF (compiled): generated from the Org export via LaTeX.

Markdown/HTML exports are not wired yet; use Org + PDF for releases.

12.6 Export commands (Emacs)

From a docbook contents view:

- ‘M-x arxana-browser-docbook-export-org‘
- ‘M-x arxana-browser-docbook-export-pdf‘

From Lisp (batch or interactive):

- ‘arxana-docbook-export-org-book‘
- ‘arxana-docbook-export-pdf-book‘

Exports default to ‘docs/docbook/<book>/<book>.org‘ (and ‘.pdf‘).

12.7 Prerequisites

- For PDF export, a LaTeX toolchain is required (‘ox-latex‘ uses the system LaTeX engine).
- For remote-sourced exports, enable sync and point ‘futon4-base-url‘ at the Futon1 API.

12.8 Single-file importer

- Source: dev/arxana-import.el

12.9 Context

13 Pattern workflows

- Source: dev/arxana-patterns.el

13.1 Context

The reanimation plan calls out pattern tooling as the last literate piece: we need both the pattern editing helpers and the ingest shims (see docs/reanimation-plan.org) inside the new spine so contributors change Org first and then tangle.

13.1.1 Editing Futon pattern entries

These commands let Emacs users work with Futon patterns without leaving Org buffers.

- ‘M-x arxana-patterns-open‘ fetch a Futon pattern (e.g., ‘mojo/center‘) into an Org buffer, complete with the component identifiers.
- Edit the buffer like any other Org document; the Futon ids stay in the headings so provenance survives.
- ‘g‘ (‘arxana-patterns-refresh-buffer‘) refetches the same slug; ‘C-c C-s‘ (‘arxana-patterns-save‘) pushes the updated text back to Futon.
- Run ‘dev/run-tests.sh‘ before publishing to confirm the pattern helpers and storage shims still behave.

13.1.2 Importing pattern libraries

Pattern data usually starts in the flexiarg collections from Futon3. The ingest shim converts every ‘.flexiarg‘ entry into Futon entities, links them to pattern languages, and registers each language inside the shared catalog so ‘/ego‘ lookups stay consistent.

- ‘M-x arxana-patterns-list-languages‘ lists languages plus the number of linked patterns.
- ‘M-x arxana-patterns-ingest-directory‘ ingests an entire directory and optionally wires the results into a named language.

13.2 Editing Futon pattern entries

- Source: dev/arxana-patterns.el

13.3 Context

These commands let Emacs users work with Futon patterns without leaving Org buffers.

- ‘M-x arxana-patterns-open‘ fetch a Futon pattern (e.g., ‘mojo/center‘) into an Org buffer, complete with the component identifiers.
- Edit the buffer like any other Org document; the Futon ids stay in the headings so provenance survives.
- ‘g‘ (‘arxana-patterns-refresh-buffer‘) refetches the same slug; ‘C-c C-s‘ (‘arxana-patterns-save‘) pushes the updated text back to Futon.
- Run ‘dev/run-tests.sh‘ before publishing to confirm the pattern helpers and storage shims still behave.

13.4 Importing pattern libraries

- Source: dev/arxana-patterns-ingest.el

13.5 Context

Pattern data usually starts in the flexiarg collections from Futon3. The ingest shim converts every ‘.flexiarg‘ entry into Futon entities, links them to pattern languages, and registers each language inside the shared catalog so ‘/ego‘ lookups stay consistent.

- ‘M-x arxana-patterns-list-languages‘ lists languages plus the number of linked patterns.
- ‘M-x arxana-patterns-ingest-directory‘ ingests an entire directory and optionally wires the results into a named language.

14 QA checklist

14.1 Context

15 Recent changes (futon4, pilot)

- (no entry content yet)

16 Relation browsing inside Emacs

16.1 Context

- ‘M-x arxana-relations-show-ego‘ renders the ‘/ego/:name‘ response in the ‘*Arxana Relations*‘ buffer, splitting outgoing vs incoming links.
- ‘M-x arxana-relations-show-cooccur‘ formats ‘/cooccur/:name‘ so you can spot frequently paired entities without leaving Emacs.
- ‘M-x arxana-relations-show-tail‘ displays the ‘/tail‘ feed (most recent relations) with confidence + timestamp notes.

Each of these commands relies on the ‘arxana-store-*‘ helpers above, so they honour the same failure handling and record diagnostics in ‘arxana-store-last-error‘ when Futon is unreachable.

The hyperedge helper is exercised via the scholium adjacency tests (‘arxana/test/arxana-adjacency-test.el‘), which stub the HTTP layer and verify that inclusion / transclusion scholia emit source, target, passage, and provenance endpoints before hitting ‘/hyperedge‘. Snapshot save/restore flows are covered by ‘arxana/test/arxana-saving-test.el‘, ensuring the Emacs commands call ‘/snapshot‘ and ‘/snapshot/restore‘ with the selected scope. The Org importer/exporter are covered by ‘arxana/test/arxana-import-test.el‘ and ‘arxana/test/arxana-export-test.el‘, guaranteeing that ‘.org‘ bundles can round-trip through XTDB snapshots.

17 Storage bridge

17.1 Context

17.2 Context

This section documents developer-facing storage concepts and the persistence contract used by the Futon4 client when it synchronizes with Fu-

ton1/XTDB.

17.3 Arxana links scope

Arxana links target the Xanadu-style semantics that matter for authoring: transclusion, versioned anchors, provenance, and scholia. The system persists strategies and anchor evidence rather than raw computed links so that links can be recomputed as documents evolve while preserving author intent.

17.4 Data structures

- Strategy (arxana/link-strategy): finder rules + source metadata.
- Scholium (arxana/scholium): author annotations tied to anchors.
- Voiced link (arxana/voiced-link): declared link assertions tied to a strategy.
- Surface form (arxana/surface-form): phrase-to-concept mappings for link hints.
- Embedding cache (arxana/embedding-cache): cached neighbors for pattern lookups.

Each record has a stable id derived from content or deterministic inputs so it can be regenerated and round-tripped reliably.

17.5 Persistence model

The persistence model is XTDB-only. Arxana wraps each record as a Futon entity with ‘:id‘, ‘:name‘, and ‘:type‘, storing the full record JSON in ‘:source‘. This keeps the Futon API contract stable while allowing richer payloads to be serialized by the client.

Primary helpers in ‘dev/arxana-links.el‘:

- ‘arxana-links-persist-*‘ (write)
- ‘arxana-links-load-*‘ (read)
- ‘arxana-links–wrap-entity-payload‘ / ‘arxana-links–unwrap-entity‘ (encoding)

17.6 Authoring workflow

Authoring is scholium-first with region selection:

1. Select text in an article buffer.
2. Capture a scholium (annotation) or surface form.
3. Persist the record to Futon1; strategies recompute links as needed.

Anchors are resilient: a content hash plus contextual windows let Arxana re-find regions after minor edits. Surface forms and embeddings guide link suggestions and future automation.

18 Testing hooks

18.1 Context

The ERT suite in ‘arxana/test/arxana-store-test.el‘ provides fake responses for ‘/entity‘ and ‘/ego‘, so we can validate the glue without a running Futon server. Extend those tests whenever you add a new helper.

19 Troubleshooting relations

19.1 Context

1. Verify ‘futon4-enable-sync‘ is non-nil and rerun the command that failed.
2. Inspect ‘arxana-store-last-error‘ to see the HTTP method, reason, and any context when a request returns ‘nil‘.
3. Use the relation browser commands above to confirm Futon is returning the data you expect; the buffer will show No data returned from Futon when the server was unreachable.
4. Run the adjacency unit tests to prove scholium links are mirrored correctly:

   “‘bash HOME=$PWD/.home-tmp EMACS$_{\text{NATIVECOMPILATION}}$=0 emacs –batch \ -l arxana/dev/bootstrap.el \ -l arxana/test/arxana-adjacency-test.el \ -f ert-run-tests-batch-and-exit “‘

These tests stub the network layer and assert that ‘futon4–sync-about-links‘ calls the new store helpers with derived ids, so failures often indicate an unexpected change in the hook wiring.

20 Workflow notes

20.1 Context

1. Regenerate / load with ‘M-x arxana-build‘.
2. Ensure your article buffer has been turned into an article (‘M-x make-current-buffer-into-article‘).
3. Run the relevant ‘arxana-store-*‘ helper from Emacs; if it reports an error check ‘arxana-store-last-error‘ for the diagnostic.
4. Run the matching curl command to confirm the Futon server contains the expected data.

21 Dev docs

Top-level heading for developer documentation entries generated from dev/*.el.

21.1 dev/arxana-article.el

21.2 Context

- arxana-article-path-cache: Defines a hash-table cache keyed by article name to store canonical filesystem paths, reducing repeated normalization work. It acts as a shared mutable store across article lifecycle operations, supporting fast lookups when bridging Arxana article identities to Futon-backed storage paths.
- arxana-article–canonical-path: Normalizes a given path through Futons canonicalization hook when available, gracefully falling back when Futon is not loaded. This helper centralizes path hygiene so higher-level article operations can assume stable, comparable paths regardless of caller or storage backend.
- arxana-article– (article lookup helper): Provides name-to-path resolution for articles, consulting the path cache and Futon storage helpers to locate canonical article roots. It forms the primary integration

point between Arxanas logical article identifiers and the underlying filesystem representation.

- arxana-article– (metadata or scholia hook): Coordinates article-level metadata or scholia updates with storage changes, ensuring side effects are synchronized with Futon operations. This definition encapsulates lifecycle hooks so classic Arxana behaviors remain consistent as storage evolves.

- arxana-article– (deletion or cleanup helper): Implements safe article removal by reconciling Arxanas deletion semantics with Futons storage model, updating caches and invoking storage helpers as needed. It represents the teardown counterpart to article creation and lookup within the lifecycle bridge.

21.3 dev/arxana-browse.el

21.4 Context

- arxana-browse: Customization group that namespaces all Part VI browsing helpers under the broader Arxana configuration. It provides a user-facing entry point in Emacs customization system, signaling that the commands in this file are about navigation and discovery rather than content editing, and integrates legacy browsing workflows into the modern arxana-* command family.

- arxana-browse–ensure: Internal guard function that validates the presence of a legacy browsing function before invocation. It checks whether a given symbol is fboundp and raises a clear user-error if not, guiding contributors to load development sources where tangling is enabled. This centralizes error handling for all wrapper commands.

- arxana-browse-open-catalog: Interactive autoloaded command that opens the article catalog using legacy Part VI menus. With a prefix argument, it lists all articles; otherwise it shows the default subset. It delegates to article-menu-listing or article-menu-list-all-articles after validation, providing a stable modern entry point without exposing older function names.

- arxana-browse-follow-link: Interactive wrapper that follows a reference or scholium at point in the current buffer. It ensures the legacy follow-reference-or-scholium function is available, then delegates navigation.

This command bridges in-buffer navigation semantics from the old system into the arxana-browse namespace for consistent keybinding and discovery.

- arxana-browse-history-back: Interactive navigation command that moves backward through the temporal article browsing history. It validates and calls the legacy sb-back function, preserving established history semantics while presenting them as part of the modern browsing API. This supports linear exploration workflows alongside catalog- and link-based navigation.

21.5 dev/arxana-compat.el

21.6 Context

- arxana-compat–orig-get-article: Holds the original ‘get-article‘ function object before any compatibility shimming occurs. This variable is used to preserve legacy behavior and enable safe delegation when the modern lookup path does not yield a result, ensuring reversibility and minimal disruption when interposing the new accessor logic.

- arxana-compat–hash-article: Looks up an article by name in the modern ‘article-table‘ hash table when it exists and is valid. Returns a ‘(NAME . VALUE)‘ cons only when a non-nil value is found, providing a normalized result shape compatible with legacy expectations while cleanly short-circuiting if prerequisites are missing.

- arxana-compat–get-article: Acts as the compatibility accessor that prefers the modern hash-table-backed article store and transparently falls back to the original ‘get-article‘ implementation. This function centralizes the precedence logic and ensures callers do not need to know whether data is sourced from modern or legacy storage.

- with-eval-after-load 'arxana-tangled: Installs the compatibility shim once the core system is loaded by capturing the existing ‘get-article‘ symbol-function and redefining it to point at ‘arxana-compat–get-article‘. This integration point ensures the override happens safely, only once, and without breaking load order assumptions.

- provide 'arxana-compat: Declares the feature for Emacs require system, allowing other components to depend on the compatibility layer explicitly. This signals that legacy accessor interposition is available

and active, enabling modular integration within the wider Arxana codebase.

21.7 dev/arxana-derivation.el

21.8 Context

- arxana-derivation–build-items: Walks rendered scholia to extract ‘derives-from‘ metadata and construct normalized preview plists. Each item captures source/target names, buffer ranges, snippet text, character length, derivative kind (inclusion/transclusion/identification), stable preview keys, and optional Futon article IDs. This function defines the core data shape consumed by both preview rendering and highlight overlay installation.

- arxana-derivation–collect-previews: Entry point run after markup completes; gathers derivation preview items for the current Scholium Display render. It resolves the rendering target buffer, builds preview items from ‘raw-scholia‘, resets cached render state, and immediately applies highlights. This function bridges upstream scholia processing with downstream UI features and maintains buffer-local preview caches.

- arxana-derivation–apply-highlights: Installs and refreshes overlay-based highlights for inclusion, transclusion, and identification passages in a target buffer. It clears prior overlays, consults user customization flags, bounds ranges safely to the buffer, assigns faces and priorities, and tracks overlays buffer-locally. This is the primary integration point between derivation metadata and visible inline emphasis.

- arxana-derivation-preview-refresh: Interactive command that re-renders the entire derivation preview block in the current buffer. It safely handles read-only/view-mode buffers, preserves point with markers, and delegates to the renderer to rebuild collapsible preview content. Used after toggling preview state to keep the display synchronized with stored expansion preferences.

- arxana-derivation–render-hook: Hook function attached to ‘scholia-display-extras-hook‘ that injects the derivation preview block into the Scholium Display buffer. It ensures previews appear as an auxiliary section without modifying the main rendering pipeline, completing the integration that surfaces derivation audits inline for readers.

21.9 dev/arxana-docbook.el

21.10 Context

- arxana-docbook-browse: Interactive entry point that selects a doc book (defaulting to futon4), initializes a tabulated-list browser buffer, sets the active book, probes available data sources, and populates entries. Integrates with filesystem discovery under dev/logs/books, optional remote API usage, and configures header-line status. Serves as the primary UI for browsing docbook entries inside Emacs.

- arxana-docbook–refresh: Core data refresh routine for the browser mode that probes remote storage and filesystem availability, selects the effective data source, fetches recent entries accordingly, normalizes them into tabulated-list rows, and updates mode-line and header-line state. Central integration point between probing logic, data acquisition, and the tabulated-list UI.

- arxana-docbook–probe-storage: Probes the Futon remote docbook API for a given book, issuing an HTTP request to fetch contents metadata and returning a status plist describing reachability, sync state, and heading counts. Encapsulates remote availability logic, error handling, and status classification used to decide between storage, filesystem, or ephemeral state sources.

- arxana-docbook-open-book: Builds a compiled, read-only Org buffer that merges TOC structure with available entries for an entire book. Renders headings, base entries, lab-draft additions, and fallback descendant content, preserving separation from source artifacts while inserting navigable source links. Acts as a higher-level compiled view distinct from the tabulated browser.

- arxana-docbook–render-entry: Renders a single docbook entry into a dedicated Org buffer, resolving related entries by doc-id, selecting the latest non-lab version, and generating a contextual view. Integrates navigation state, TOC ordering, two-pane layout with optional source file buffer, and function-jump support to bridge documentation and underlying code.

21.11 dev/arxana-export.el

21.12 Context

- arxana-article–labels-for: Declares an external helper that returns label metadata for a named article, used by exporters to annotate Org output with stable identifiers or tags. This declaration signals a dependency on the article model layer, allowing export code to enrich serialized Org content without directly coupling to article storage internals.

- get-article: Declares access to the canonical article lookup in the tangled Arxana data layer. Export helpers rely on this to resolve article names into full article structures before extracting fields, ensuring exported Org files reflect the authoritative article state used by XTDB snapshots.

- scholium-text: Declares a function that extracts the primary textual body of a scholium from an article record. Exporters use this accessor to populate Org sections with human-readable content, separating narrative text from metadata like names or relationships during bundle generation.

- arxana-store-save-snapshot: Declares the entry point for persisting a snapshot of the current store, optionally scoped and labeled. The exporter integrates with this function to produce shareable XTDB-backed bundles, coordinating Org file writes with snapshot creation for portability and reproducibility.

- arxana-store-sync-enabled-p: Declares a predicate indicating whether store synchronization is active. Export logic can consult this to adjust behavior when writing Org files or snapshots, avoiding assumptions about remote sync availability and supporting both offline and synchronized export workflows.

21.13 dev/arxana-flexiarg-normalize.el

21.14 Context

- arxana-flexiarg–clause-regexp: Defines the canonical pattern for recognizing legacy simplified clause headings in flexiarg bodies, matching uppercase labels followed by a colon and inline text. This constant

anchors the parsers ability to detect IF/HOWEVER/THEN-style sections and is central to determining whether a buffer can be structurally rewritten into the canonical ! conclusion plus + CLAUSE format.

- arxana-flexiarg–extract-clauses: Parses a flexiarg body string into an ordered list of (LABEL . TEXT) pairs by scanning for simplified clause headers and aggregating following lines. It normalizes labels to uppercase, preserves relative ordering, trims trailing whitespace, and ignores unrelated preamble content, providing a structured intermediate representation consumed by the canonical formatter.

- arxana-flexiarg–build-body: Constructs the full canonical flexiarg body text from extracted clauses. It selects a summary for the ! conclusion section using THEN, CONCLUSION, or the first clause as fallback, inserts [pending] placeholders when content is empty, formats each clause with proper indentation, and returns a normalized string suitable for direct buffer replacement.

- arxana-flexiarg–normalize-buffer: Orchestrates in-buffer normalization by splitting header and body, detecting already-canonical content, rewriting legacy alias headings, extracting simplified clauses, and rebuilding the canonical template. It performs destructive buffer edits only when applicable, signals user errors when no recognizable clauses exist, and reports whether any modifications were made.

- arxana-flexiarg-normalize-file: Autoloaded entry point that normalizes a single ‘.flexiarg‘ file on disk using a temporary buffer and ‘arxana-flexiarg–normalize-buffer‘. It supports a dry-run mode for reporting prospective changes without writing, integrates with interactive use, and returns status information used by higher-level batch operations.

21.15 dev/arxana-import.el

21.16 Context

- arxana-import–read-file-contents: Reads an entire file from disk into a single string using a temporary buffer. This is the lowest-level I/O helper for the import pipeline, abstracting file access so higher-level functions can operate purely on strings. It assumes the file exists and is readable, and does not perform decoding, validation, or Org-specific parsing itself.

- arxana-import–org-title-from-string: Scans a string containing Org-mode text and extracts the value of a #+TITLE keyword when present. The search is case-insensitive and only considers the first matching title directive. It returns a trimmed string or nil, enabling higher-level logic to prefer explicit Org metadata over filenames when naming imported articles.

- arxana-import–derive-name: Determines a stable article name by combining file metadata and content inspection. It prefers an Org #+TITLE extracted from the file text, falling back to the files base name or final path component. This function centralizes naming policy so imports produce consistent identifiers even when Org metadata is missing or incomplete.

- arxana-import-org-file: Imports a single Org file into Arxanas article graph and, via store hooks, into XTDB. It reads file contents, derives an article name, creates a scholium node with Org type metadata, and optionally ensures a backing store record with a canonicalized path. It is interactive, reports progress, and returns the article name.

- arxana-import-org-directory: Batch-imports all Org files in a directory, optionally recursing into subdirectories. It discovers candidate files, sequentially imports each via arxana-import-org-file, counts successful imports, and reports a summary message. This function defines the main entry point for bulk ingestion workflows and directory-based knowledge synchronization.

21.17 dev/arxana-inclusion.el

21.18 Context

- arxana-inclusion–require-article: Validates that an article NAME is registered by calling a provided LOOKUP function, returning the resolved article object or signaling a user-error. Errors include the CALLER name and actionable guidance to register the article first, preventing silent nil insertions and making legacy include/transclude failures explicit and debuggable.

- include-article: Rebound at runtime (after arxana-tangled loads) to a wrapped version that enforces article existence checks before executing the original behavior. The wrapper integrates arxana-inclusions guard

logic transparently, preserving the original commands interface while ensuring missing or unregistered articles raise clear, user-facing errors.

- transclude-article: Similarly rebound after arxana-tangled loads to a guarded wrapper that validates referenced articles before transclusion. This ensures legacy transclude workflows fail loudly and consistently when articles are unknown, aligning behavior with inclusion safeguards and avoiding silent insertion of empty or nil content.

- with-eval-after-load arxana-tangled: Establishes deferred integration so that wrapping only occurs once arxana-tangled is available and the target commands are defined. This avoids load-order issues, conditionally applies guards based on function availability, and keeps arxana-inclusion decoupled from the core implementation until integration time.

- provide arxana-inclusion: Publishes the feature symbol for this module, enabling other parts of the system to require it reliably. This marks the inclusion-guard layer as a distinct, loadable unit responsible for enforcing registration checks around legacy include and transclude commands.

21.19 dev/arxana-lab.el

21.20 Context

- arxana-lab–locate-root: Resolves the lab notebook root directory by honoring the customizable arxana-lab-root or auto-detecting a parent directory containing a lab folder. It derives a base path from the current file, buffer, or default directory, integrates with Emacs locate-dominating-file, and normalizes the result to the concrete lab/ path used throughout the browser.

- arxana-lab-entries: Collects and returns structured lab entry objects derived from JSON files under lab/raw. It locates the root, enumerates raw JSON files, parses each into a plist containing session metadata and related paths, drops unreadable entries, and sorts the resulting list by descending start timestamp, forming the primary index used by higher-level UI commands.

- arxana-lab-file-items: Produces lightweight file item descriptors for browsing non-entry artifacts such as raw logs, stubs, and drafts. Given

a kind symbol, it scans the corresponding subdirectory, filters regular files, and returns plists with labels, paths, modification times, and kind tags, enabling generic file pickers and unified open behavior across artifact types.

- arxana-lab-open-stub-viewer: Opens a human-readable Org-mode view of a stub file that may embed summaries inside JSON or fenced org blocks. It applies multiple extraction strategiesJSON parsing, regex fallbacks, escape decoding, and raw org detectionto recover meaningful content, preserves org headers when present, reports failures diagnostically, and presents the result in a read-only viewer buffer.

- arxana-lab-open-raw-viewer: Renders a raw lab session JSON file as a structured Org-mode notebook view. It parses session metadata, touched files, and user/assistant messages, inserts a summary section, and builds a chronological message timeline with assistant messages visually highlighted. The function integrates JSON decoding, org presentation, and overlay styling to support interactive inspection.

21.21 dev/arxana-media.el

21.22 Context

- arxana-media–entries: Loads and caches the Zoom media catalog from a JSON index, auto-detecting the path and reloading only when the file modification time changes. It normalizes each entry via arxana-media–normalize-entry and exposes a shared in-memory list used by all browsing, filtering, playback, tagging, and publication features, while emitting warnings when the catalog is missing or unreadable.

- arxana-media-play-at-point: Interactive entry point for playback integration with the Futon browser. It resolves the media item at point, derives a readable title and playable file path, initializes a playback queue based on the current browser context, manages a playback token to avoid races, and spawns an external player process with optional autoplay chaining.

- arxana-media–track-items: Core adapter that turns catalog entries into browser-visible media-track items. It filters entries by status or project, sorts them by recording time descending, and emits plists containing type, labels, project metadata, status, and SHA identifiers.

This function bridges the raw catalog data model with the arxana-patterns browser UI.

- arxana-media-publish-marked: High-level workflow for publication export. It gathers marked catalog entries, derives a slugged publication name, copies readable audio files into a publication directory, writes optional JSON metadata, tags entries via $\text{zoom}_{\text{sync.py}}$ using a configurable prefix, invalidates the catalog cache, and refreshes the browser to reflect publication state.

- arxana-media–retitle-track: Internal integration point for mutating catalog metadata through $\text{zoom}_{\text{sync.py}}$. It validates entry status, constructs command-line arguments keyed by SHA256, optionally scopes updates to a recorder root, executes the external script, and reports failures. This function enforces that only hold items are retitled and keeps Emacs as a thin orchestrator.

21.23 dev/arxana-patterns-ingest.el

21.24 Context

- arxana-patterns-ingest–parse-flexiarg: Reads a single ‘.flexiarg‘ file and converts it into a structured plist with pattern name, optional title, summary, ordered components, and raw metadata. It recognizes ‘@‘ metadata directives and section headers, trims empty lines, slugifies section labels, derives a fallback name from the path if needed, and validates required identifiers.

- arxana-patterns-ingest–ingest-file: Orchestrates ingestion of one parsed pattern into Futon by ensuring the pattern entity exists, discovering existing ‘:pattern/includes‘ links, creating or updating component entities, and wiring ordered component relations. Returns a minimal plist with the patterns canonical name and resolved entity id for downstream language assembly.

- arxana-patterns-ingest–ensure-language: Ensures a pattern-language entity exists, links it to an ordered list of pattern entities with order metadata, and applies classification tags for source and status. It also guarantees inclusion in a central language catalog, deduplicates existing relations via ‘/ego‘ lookups, and derives defaults from directory context when explicit metadata is absent.

- arxana-patterns-ingest-directory: High-level interactive and programmatic entry point that ingests all ‘.flexiarg‘ files in a directory, sorted by modification time. It validates Futon sync, processes each file, optionally creates or updates a pattern-language wrapper, supports overriding language title and status, and reports ingestion results and language identifiers to the user.

- arxana-patterns-list-languages: Interactive UI command that queries the catalog and ‘/ego‘ metadata to present a read-only summary buffer of known pattern languages. It displays language name, source classification, status classification, and pattern count, providing a lightweight browsing and validation surface for previously ingested libraries.

21.25 dev/arxana-patterns.el

21.26 Context

- arxana-patterns-open: Interactive entry point that fetches a Futon pattern by name via ego queries, resolves its entity, summary, and component graph, and renders the result into a dedicated Org buffer. The buffer includes header metadata, a bounded summary block, and hierarchical component sections, then enables Org mode and a minor view mode for syncing edits back to Futon.

- arxana-patterns–fetch-pattern-data: Core data assembly function that ensures Futon sync, queries ego for a pattern entity, fetches canonical source text, discovers component links, resolves component entities (including parent relationships), computes nesting levels, and sorts components by declared order. Returns a normalized plist with pattern id, name, title, summary text, and a leveled, ordered component list for rendering.

- arxana-flexiarg-collection-mode: Major mode derived from flexiarg-mode that supports editing multiple .flexiarg files in a single buffer with structural tracking. It guards edits to stay within source-file boundaries, manages overlays for hidden metadata and context blocks, integrates outline folding, provides save/revert commands to write back to original files, and enables cycling views focused on conclusions.

- arxana-patterns-edit-collection: Opens an editor buffer for a filesystem collection of .flexiarg files, either selected from the browser or

chosen interactively. It sorts files by declared @order metadata, prepares a combined buffer with per-file headers and tracked regions, activates arxana-flexiarg-collection-mode, and allows batch editing with safe persistence back to individual source files.

- arxana-patterns-browse: Launches the main tabulated-list browser UI for Arxana patterns and related assets. It initializes navigation state and renders a hierarchical menu covering pattern languages, filesystem collections, docbooks, media, and lab data. The browser supports drilling into entries, importing collections into Futon, reordering patterns, editing collections, and opening rendered patterns.

21.27 dev/arxana-relations.el

21.28 Context

- arxana-relations-buffer-mode: Minor mode enabling relation browsing buffers with dedicated keybindings for refresh, provenance, and ID copying. It installs a custom revert-buffer-function so standard buffer refresh re-renders Futon relation data instead of reverting file contents, and cleans up buffer-local state when disabled. Serves as the primary integration point between rendered relation views and standard Emacs buffer workflows.

- arxana-relations-context: Buffer-local plist storing metadata about the most recent Futon relation render, such as entity identifiers, relation type, and request parameters. This context is reused by refresh, provenance lookup, and copy commands to avoid recomputation and ensure commands operate on the exact dataset currently displayed, making relation buffers stateful and self-describing.

- arxana-relations–render-ego: Renders Futon /ego relation responses into a structured, human-readable buffer view. It extracts entity names, types, and directional relationships from the response body, formats them into labeled sections, and inserts them using shared helpers. This function defines the canonical presentation for egocentric relationship exploration inside Emacs.

- arxana-relations–render-cooccur: Handles rendering of Futon /cooccur relation data, focusing on entities that frequently appear together. It transforms raw response bodies into ordered line items grouped under descriptive section headers, emphasizing counts or strengths where

available. This renderer supports comparative analysis of related entities without requiring external tools or manual JSON inspection.

- arxana-relations–render-tail: Renders Futon /tail relation results, typically representing downstream or trailing relationships from a source entity. It formats sparse or asymmetric data defensively, ensuring empty sections are explicit, and integrates with shared insertion helpers for consistent layout. Acts as a specialized renderer for less common but structurally distinct relation queries.

21.29 dev/arxana-saving.el

21.30 Context

- arxana-saving–fallback-scopes: Constant listing default snapshot scopes ("all" and "latest") used when the snapshot store helpers are unavailable. It provides a minimal, predictable completion set for prompting users, ensuring save/restore commands remain usable even if the newer arxana-store APIs are missing or not yet loaded.

- arxana-saving–prompt-scope: Reads a snapshot scope with a prompt and default, abstracting over interactive and noninteractive use. In batch or programmatic contexts it silently returns the default. When available, it delegates to arxana-stores richer scope prompt; otherwise it falls back to completing-read over predefined scopes, maintaining compatibility across configurations.

- arxana-saving–snapshot-enabled-p: Predicate that decides whether save operations should be redirected to snapshot-based behavior. It checks that Futon sync is enabled via futon4-enable-sync and that the snapshot save function is defined. This central gate cleanly separates legacy file-based saving from snapshot-backed saving at runtime.

- arxana-saving–wrap-save-all: Around-advice wrapper for save-all-scholia that transparently switches saving to Futon snapshots when enabled. It prompts for scope and optional label, invokes arxana-store-save-snapshot, extracts a snapshot ID if possible, and reports status to the user. When snapshots are disabled, it delegates unchanged to the original save-all implementation.

- arxana-saving–wrap-read-scholia: Around-advice wrapper for read-scholia-file that treats the filepath argument as a snapshot ID when sync is

enabled. It prompts for a restore scope and calls arxana-store-restore-snapshot instead of reading from disk. If snapshot restore is unavailable, it cleanly falls back to the legacy file-based reader.

21.31 dev/arxana-scholium.el

21.32 Context

- defgroup arxana-scholium: Defines the customization group that houses all scholium authoring helpers, integrating them into the broader ‘arxana‘ customization hierarchy. This group provides a stable namespace and user-facing entry point for configuring scholium-related commands and variables, signaling that the files purpose is interactive authoring support rather than core publishing or rendering logic.

- defvar new-scholium-name: Declares the global variable used to store the current or last-used scholium name during authoring. It acts as shared state between interactive prompts and underlying scholium creation commands, enabling name reuse across invocations and compatibility with the historical ‘new-scholium-mode‘ workflow expected by existing scholium tooling.

- defvar new-scholium-about: Holds the normalized about metadata for a scholium in the shape expected by legacy scholium machinery. This variable is populated by helper functions before invoking ‘make-scholium‘, serving as the primary integration point where modern wrappers adapt user intent into the data structure consumed by the classic scholium publishing pipeline.

- defun arxana-scholium–compose-about: Internal helper that transforms a TARGET into the canonical scholium about format and immediately triggers scholium creation. By wrapping TARGET in the nested list structure required by legacy code and calling ‘make-scholium‘, it bridges modern command usage with historical expectations, centralizing data normalization in a single, reusable function.

- defun arxana-scholium-compose: Autoloaded interactive command that opens a scholium buffer using ‘new-scholium-mode‘, optionally prompting for a name. It ensures required support functions exist, sets the scholium name, invokes ‘make-scholium‘, and guides the user through the publish flow. This function is the primary user-facing entry point for scholium authoring.

21.33 dev/arxana-store.el

21.34 Context

- arxana-store–request: Core HTTP bridge to Futon, handling sync enablement, URL construction, headers, JSON encoding/decoding, timeouts, and error capture. Accepts method, path, optional payload, and query string, returning parsed JSON alists. Records last response and structured errors, normalizes protocol failures, and integrates with legacy futon4 settings like base URL and canonical paths.

- arxana-store-ensure-entity: High-level POST helper for Futons /entity endpoint that ensures an entity exists with the given name and optional metadata. Accepts keyword arguments mirroring Futons payload (type, ids, source, counts, pinning, timestamps), normalizes keyword types, validates required name, and delegates network behavior to arxana-store–request.

- arxana-store-upsert-scholium: Interactive and programmatic helper to create or update a scholium relation between two articles by name. Resolves article names to Futon IDs via futon4 helpers, validates sync and inputs, posts a relation with an optional label, reports interactive success, and returns a plist describing the stored relation endpoints.

- arxana-store-fetch-entity: Fetches a single entity by ID from Futon with optional version or as-of timestamp parameters. Builds encoded query strings, validates required identifiers, supports interactive prompting, and returns parsed JSON describing the entity state. Serves as the primary read path for entity inspection and historical lookup integration.

- arxana-store-ego: Queries Futons /ego endpoint to retrieve an ego network centered on a named entity, optionally limited in size. Supports interactive defaults from the current article context, validates presence of a target name, encodes query parameters, and returns structured JSON suitable for graph or relationship analysis.

21.35 dev/arxana-test-support.el

21.36 Context

- arxana-test–locate-tangled: Computes a best-effort filesystem path to the generated arxana-tangled.el by checking multiple candidate loca-

tions relative to either an explicit arxana-root-directory or the current default-directory. Designed for test harness flexibility, it abstracts over repo-root versus subtree execution and returns the first existing path or nil if none are found.

- arxana-test–ensure-tangled-loaded: Ensures the tangled Arxana sources are loaded into the Emacs session for tests. It checks for the arxana-tangled feature, locates the file via arxana-test–locate-tangled, and load-files it if necessary. Fails fast with a clear error when tangling is unavailable, guiding developers toward proper setup.

- provide (arxana-test-support): Publishes the arxana-test-support feature for require-based inclusion in ERT suites and other dev tooling. This formalizes the module boundary so test files can depend on shared loading and path-resolution helpers without duplicating logic or hard-coding repository layouts.

- require (cl-lib): Declares a dependency on cl-lib to support Common Lispstyle utilities, notably cl-loop, used in path candidate selection. This ensures consistent iteration and early-return semantics across Emacs versions, making the test-support helpers portable and reliable within diverse development environments.

- featurep check for arxana-tangled: Integrates with Emacs feature system to avoid redundant loading of the tangled sources. By keying off the arxana-tangled feature symbol, the test support code cooperates with the wider Arxana load lifecycle and respects prior initialization performed by other dev or test entry points.

21.37 dev/arxana-xtdb-browse.el

21.38 Context

- arxana-xtdb-browse-default-limit: User-customizable default number of relations fetched from XTDB when browsing. This value seeds buffer-local state and is used by interactive commands unless overridden. It integrates with the ‘/tail‘ API via ‘arxana-store-tail‘, shaping the scale and performance of the browser UI and providing a consistent default across sessions.

- arxana-xtdb–tail-rows: Normalizes heterogeneous ‘/tail‘ responses into a clean list of relation alists. It tolerates multiple payload shapes

(‘:tail‘, ‘:relations‘, ‘:links‘, ‘:items‘) and filters out non-relation data using a structural predicate. This function is the main compatibility layer insulating the UI from backend response variations.

- arxana-xtdb–rows->entries: Transforms normalized relation rows into ‘tabulated-list‘ entries suitable for display. It extracts source, target, relation type, and last-seen metadata, formats human-readable entity labels, and preserves the original row as entry data. This bridges raw XTDB relations with Emacs tabular UI machinery.

- arxana-xtdb-refresh: Core command that fetches fresh XTDB data and updates the browser buffer. It calls ‘arxana-store-tail‘, caches the raw response for inspection, updates buffer-local limits, rebuilds tabulated entries, and refreshes the display. It also handles empty results gracefully and provides user feedback, serving as the main integration point with the datastore.

- arxana-xtdb-browse: Autoloaded entry point that creates or reuses the XTDB browser buffer, enables the dedicated major mode, and triggers an initial refresh. It supports an optional interactive limit argument and manages buffer presentation. This function ties together mode setup, data retrieval, and user workflow into a single command.

21.39 dev/bootstrap.el

21.40 Context

- arxana-root-directory: Computes and exposes the absolute root directory of the Arxana checkout, resolving relative to the loaded bootstrap file, default-directory, or fallback context. This path anchors discovery of spine2.org, tangled output, and dev modules, and is also used to extend load-path for development. It centralizes filesystem assumptions so other bootstrap functions can remain location-agnostic.

- arxana-allow-tangle: User-facing customization flag that gates whether tangling and rebuilding from Org sources is permitted. By default it is nil, enforcing a safer workflow where developers load dev/ sources directly. arxana-build checks this variable and signals a user-error unless explicitly enabled, making tangling an intentional, opt-in operation during development or CI.

- arxana–ensure-version: Internal guard that enforces a minimum supported Emacs version before any substantive build or load occurs. It

compares emacs-version against arxana-minimum-emacs-version and signals a hard error on mismatch. This provides an early, explicit failure mode that avoids subtle runtime incompatibilities later in the bootstrap or tangled code.

- arxana-load-spine: Loads spine2.org via org-babel so that helper functions and file lists defined in the spine become available. It validates the spines existence, ensures all required Org, Babel, and networking dependencies are present, and suppresses confirmation prompts for evaluation. This function is the bridge between Org sources and executable Elisp.
- arxana-build: Primary entry point that ensures a current tangled Elisp file exists and is loaded. It conditionally regenerates arxana-tangled.el based on timestamps or a force argument, loads the result, prefers dev/staging implementations, and opportunistically requires many Arxana subsystems. It integrates version checks, tangling policy, spine expansion, and module loading into one workflow.

21.41 dev/bootstrap2.el

21.42 Context

- bootstrap2.el: Acts as a minimal entrypoint that loads dev/bootstrap.el relative to the current default-directory, ensuring developer bootstrap logic is executed without duplicating configuration; it relies on Emacs load semantics to resolve and evaluate the target file at startup or during interactive development.
- bootstrap2.el: Provides an indirection layer for tooling and editors that expect a stable bootstrap filename, allowing dev/bootstrap.el to evolve while consumers reference this wrapper; integration hinges on expand-file-name to compute an absolute path before loading.
- bootstrap2.el: Encapsulates environment setup by delegating all behavior to dev/bootstrap.el, making this file intentionally side-effectful and definition-free; its role is orchestration rather than declaring functions or variables.
- bootstrap2.el: Enables reproducible developer setup by standardizing how the bootstrap code is invoked across machines and sessions, assuming default-directory points at the project root or a known subdirectory.

- bootstrap2.el: Serves as a compatibility shim in development workflows, letting scripts or Emacs commands load a predictable file while centralizing actual bootstrap logic elsewhere, reducing duplication and maintenance overhead.

21.43 dev/check-parens.el

21.44 Context

- arxana-check-parens–pos->linecol: Utility that converts a buffer position into a 1-based (LINE . COL) pair. It temporarily moves point to the given position and queries Emacss line and column functions. This normalization is used for user-facing diagnostics so batch output can be consumed by editors, CI logs, or other tooling expecting conventional line/column locations.

- arxana-check-parens–check-buffer: Core validator that loads a file into a temporary buffer, enables Emacs Lisp syntax rules, and runs ‘check-parens‘. On success it returns nil; on failure it catches the error and returns a plist containing file path, raw position, derived line/column, and a human-readable message. This isolates parsing, error capture, and reporting into one reusable unit.

- arxana-check-parens–file-list: Determines which files to validate in batch mode. It parses ‘command-line-args-left‘, honoring ‘–‘ as an option separator and ignoring flags, and falls back to discovering all ‘.el‘ files under ‘dev/‘ and ‘test/‘ when no explicit paths are given. This provides flexible CLI integration while supporting sensible defaults for CI runs.

- arxana-check-parens-run: Orchestrates validation across a list of files, either explicitly provided or discovered via the file-list helper. It processes files sequentially and short-circuits on the first detected problem, returning either nil for success or the error plist from the failing file. This function is the programmatic entry point suitable for reuse beyond the CLI.

- arxana-check-parens-cli: Batch-oriented entry point intended to be invoked via ‘emacs –batch‘. It runs the checker, prints a concise single-line status or error report with file, line, column, and message, and exits Emacs with status code 0 on success or 1 on failure. This definition integrates the checker cleanly into scripts and CI pipelines.

21.45 dev/debug-fold-output.el

21.46 Context

- add-to-list: Extends Emacs load-path to include the local dev directory, ensuring that in-repo development files are discoverable without installation. This establishes the execution context for debugging by prioritizing local sources over installed packages, which is critical when iterating on flexiarg or related Arxana components.

- load: Explicitly loads an external flexiarg.el from a contrib path, bypassing autoloads and package management. This guarantees the exact implementation under test is active, enabling reproducible debugging of folding behavior and buffer transformations tied to flexiarg, regardless of what versions might otherwise be present.

- provide: Advertises the feature symbol flexiarg after loading, satisfying require dependencies elsewhere. This makes the debug script behave like a proper feature provider, allowing other code to rely on feature resolution and ensuring integration consistency during interactive or batch evaluation.

- require: Pulls in arxana-patterns as a dependency, indicating that the debug scenario depends on pattern definitions or utilities from Arxana. This establishes the integration surface being exercised and ensures that flexiarg behavior is tested in a realistic environment with its pattern-matching support loaded.

- with-temp-buffer: Encapsulates a complete, side-effect-free test workflow that initializes flexiarg collection mode, prepares a buffer from a .flexiarg document, invokes conclusion folding, and prints before/after snippets. This form defines the scripts core purpose: a minimal, inspectable harness for debugging fold output behavior.

21.47 dev/debug-fold-props.el

21.48 Context

- provide 'flexiarg: Declares the flexiarg feature after loading the contrib implementation, making this debug script act as a minimal integration shim so other code can ‘(require 'flexiarg)‘ during development without installing the package formally.

- arxana-flexiarg-collection-mode: Activates the core Arxana flexiarg collection minor/major mode inside a temporary buffer, establishing syntax, overlays, and analysis hooks needed to parse and visualize flexiarg documents during debugging.

- arxana-flexiarg–prepare-buffer: Initializes the temp buffer with a specific ‘.flexiarg‘ document path, triggering parsing and metadata/context overlay construction; this is the primary entry point for loading real documents into the analysis pipeline.

- arxana-flexiarg-show-conclusions: Renders computed conclusions from the prepared flexiarg document, exercising the presentation layer that turns internal argument structures into user-visible overlays.

- overlay-get (metadata/context overlays): Inspects the ‘display‘ property of the first metadata and context overlays, logging their rendered forms to messages; this validates fold/display behavior and is the core diagnostic check performed by this script.

21.49 dev/docbook-toc-export.el

21.50 Context

- arxana-docbook-export-toc: Batch-oriented entry point that parses spine2.org into an Org AST, extracts all headlines, and emits a JSON array of TOC nodes. It resolves project roots, defaults book and paths, computes per-heading metadata (IDs, titles, outline paths, levels), ensures output directories exist, writes dev/logs/books/<book>/toc.json, and logs a summary message for tooling consumption.

- arxana-docbook–heading-outline: Walks upward from a headline element through its parents to collect the full hierarchical outline. It returns an ordered list of raw heading titles without text properties, forming the canonical outline path used for display strings, hierarchical context, and stable ID derivation across exports.

- arxana-docbook–doc-id: Generates a stable, deterministic document identifier per heading by hashing the book name and outline path. It accepts either a list or string path, normalizes it, computes a SHA1 digest, and returns a concise book-prefixed ID suitable for browser linking and cache-friendly TOC synchronization.

- arxana-docbook–slug: Normalizes arbitrary text into a URL-friendly slug by lowercasing, replacing non-alphanumeric runs with hyphens, collapsing duplicates, and trimming edges. Although not currently wired into the exporter, it provides a reusable utility for future TOC URLs, anchors, or human-readable identifiers.

- provide (docbook-toc-export): Declares the feature symbol for this module, enabling reliable require-based integration in batch and interactive contexts. This supports reuse by other tooling, ensures load ordering, and complements the noninteractive auto-run behavior so the exporter can function as a standalone script or a library.

21.51 dev/spine2-export.el

- Source: header

21.52 Context

- arxana-spine2-root-directory: Caches the repository root directory containing spine2.org to avoid repeated filesystem searches. It is populated lazily by helper functions and reused across exports, ensuring consistent path resolution for inputs and outputs. This variable underpins all location-sensitive operations, including file discovery, relative links, and export artifact placement.

- arxana-spine2–resolve-src-target: Determines whether an Org src block should be replaced by a code link by inspecting its :tangle header arguments. It normalizes boolean and string values, ignores non-tangled or explicitly disabled blocks, and computes default tangle targets when appropriate, integrating Org Babel metadata with export-time decisions.

- arxana-spine2–code-link-src-block: Custom LaTeX translator for Org src blocks that replaces tangled blocks with a quoted code> marker linking to the generated file. It suppresses large dev include expansions, falls back to standard LaTeX rendering when no tangle target exists, and serves as the core behavior enabling concise, link-based exports.

- arxana-latex-code-links (derived backend): Defines a custom Org export backend derived from LaTeX that overrides src-block translation

to use code-link placeholders. This backend integrates with Orgs export pipeline, allowing selective behavior changes without altering the standard LaTeX backend, and is activated explicitly during specialized exports.

- arxana-spine2–export: Orchestrates exporting spine2.org with a chosen backend to a specified output file. It ensures required Org and export dependencies are loaded, configures export options, injects preparsing hooks for code-link handling when needed, and performs a synchronous file-based export suitable for repeatable, scripted use.

21.53 dev/test-flexiarg-cycle.el

21.54 Context

- with-temp-buffer test harness: Executes an isolated, non-interactive integration test that initializes flexiarg collection mode, prepares a buffer from a sample .flexiarg document, and drives the cycle state machine. It prints diagnostic output to standard output/messages, making the file suitable for manual or batch inspection rather than automated ERT-style assertions.

- arxana-flexiarg–prepare-buffer invocation: Loads a real flexiarg document into the temporary buffer and initializes internal overlays and state. This exercises the document parsing and overlay setup paths as they would run in normal usage, ensuring metadata and context regions are created before any cycling occurs.

- arxana-flexiarg-cycle-buffer sequencing: Calls the cycle command three times, logging arxana-flexiarg–cycle-state after each transition. This validates that cycling advances deterministically through expected stages and that repeated invocations update visibility and overlay properties consistently.

- Overlay inspection logic: Locates representative metadata and context overlays, then inspects before-string/display placeholders, intangible flags, and text invisibility properties. This directly verifies how flexiarg encodes hidden or placeholder content using overlays and text properties, which is critical for correct editor interaction.

- Outline visibility assertions: Searches for key marker strings in the buffer and checks outline-invisible-p at each cycle stage. This confirms

that high-level structural sections become visible or hidden as intended across cycle states, integrating flexiarg behavior with Emacs outline invisibility mechanism.

21.55 dev/test-metadata-flag.el

21.56 Context

- provide (feature 'flexiarg): Declares the flexiarg feature as provided after loading an external contrib file, allowing other modules to ‘(require 'flexiarg)‘ without direct path knowledge. This acts as a lightweight integration shim, ensuring downstream code sees the feature as satisfied even though the implementation is loaded from a local contrib path during testing.

- arxana-flexiarg-collection-mode: Invoked inside a temporary buffer to initialize the major/minor mode responsible for flexiarg collection behavior. This sets up buffer-local state, hooks, and overlays expected by the Arxana flexiarg pipeline, ensuring subsequent preparation logic runs in a realistic mode context during this test harness.

- arxana-flexiarg–prepare-buffer: Called with a list of flexiarg document paths to populate and analyze the temporary buffer. This function is responsible for loading content, parsing metadata, and constructing internal data structures such as overlays, making it the core entry point under test for metadata extraction behavior.

- arxana-flexiarg–metadata-overlays: A dynamically populated internal variable inspected after buffer preparation to validate metadata handling. The test reports its length, using overlay count as a proxy for successful parsing and annotation, which helps verify that metadata flags in the flexiarg document are being recognized and materialized.

- with-temp-buffer: Provides an isolated, ephemeral buffer environment for running the flexiarg mode and preparation logic without side effects. This ensures the test does not pollute user buffers or global state, making it suitable for automated or ad hoc validation of metadata-related behavior in the Arxana flexiarg workflow.

21.57 dev/test-overlay-outline.el

21.58 Context

- require: Loads the built-in outline library to ensure outline-mode and related functions are available. This establishes the dependency under test and signals that the file is validating interactions with core outline functionality rather than defining new APIs. It frames the file as a minimal reproduction or regression check for outline behavior with overlays.

- with-temp-buffer: Creates an isolated temporary buffer for the test scenario, avoiding side effects on user buffers. The body sets up sample outline content, activates modes, and prints results. This structure clarifies the tests scope as self-contained, deterministic, and intended for non-interactive evaluation or batch execution.

- outline-mode: Enables outline minor mode in the temporary buffer, activating heading recognition and visibility controls. This is the primary integration point being exercised, as subsequent operations depend on outlines text properties and hiding logic interacting correctly with overlays and after-strings.

- make-overlay: Constructs an overlay spanning the first line (the heading) and attaches an after-string payload. This models real-world usage where overlays add virtual text. The test specifically probes how such overlays behave when outline visibility changes, highlighting potential edge cases in display or buffer-string output.

- outline-hide-sublevels: Invoked to collapse outline content below a given depth, triggering outlines hiding mechanics. Its interaction with the previously created overlay is the core behavior under test, validating whether overlay after-strings remain visible, move, or are suppressed when headings are folded.