Ars Legendi

Technical methodology guide

A comprehensive framework for building
text-anchored Latin learning environments

Grant Henry
granthenry34@icloud.com

Contents

1. Introduction and design philosophy

Ars Legendi is a mastery-based Latin reading and learning platform built around a single premise: grammar instruction should be anchored to the actual text students are reading, not abstracted into disconnected charts and paradigms. This guide documents the complete technical methodology I used to build the system and walks you through how to replicate the approach for other Latin texts.

The platform addresses a persistent problem in Classical language education. Students learn vocabulary from frequency lists and grammar from reference tables, but when they sit down with an actual page of Caesar or Cicero, they cannot bridge the gap between that abstract knowledge and real reading. I designed Ars Legendi to close this gap by:

• Parsing every sentence computationally to extract lemmas, morphological features, and syntactic structure

• Detecting grammatical constructions automatically using pattern matching over dependency trees

• Linking vocabulary and grammar instruction directly to specific passages where each item appears

• Tracking student mastery with an adaptive Elo-based rating system that surfaces weaknesses and guides practice

The core technical challenge

Traditional commentaries are produced line-by-line through painstaking human labor. A teacher or scholar reads each sentence, identifies noteworthy constructions, and writes explanatory notes. This process is slow and non-scalable. More critically, it produces static artifacts that cannot adapt to individual student needs.

I inverted this model. Instead of manually annotating text, I process Latin through a Natural Language Processing (NLP) pipeline that produces structured, computable annotations. These annotations then feed into automated detectors that identify constructions systematically. The result is a dynamic commentary that can be queried and personalized, and that can be extended to new texts by running them through the same pipeline.

Prerequisites: This guide assumes familiarity with basic programming concepts (variables, functions, JSON) and intermediate comfort with JavaScript and Python. You do not need prior NLP experience. I explain the relevant concepts from first principles.

Source code: The full codebase for Ars Legendi is available on GitHub. Throughout this guide, I reference specific files and modules. For the complete implementation, including edge cases and details not covered here, consult the repository directly.

What this guide covers

This document walks through every major technical component of the system:

1. **The NLP pipeline** : how raw Latin text is tokenized, lemmatized, and parsed into Universal Dependencies (UD)

2. **Construction detection** : how rule-based detectors identify ablative absolutes, indirect statements, purpose clauses, and nine other construction types

3. **Data architecture** : the JSON structures that store sentences, tokens, constructions, vocabulary, and glosses

4. **The Word Inspector** : how the interactive token analysis interface retrieves and displays morphological data

5. **Vocabulary system** : the three-phase learning model (recognition, reinforcement, production) and session construction

6. **Grammar practice** : how quiz configurations define multi-step exercises anchored to detected constructions

7. **Mastery tracking** : the Elo-based rating algorithm adapted for educational skill measurement

8. **Adaptive feedback** : pattern detection that identifies struggling students and triggers interventions

9. **Teacher analytics** : class-level insights that surface reteaching opportunities

10. **Replication** : a step-by-step guide to adapting the framework for a new text

2. System architecture overview

Before diving into individual components, it helps to understand how the pieces fit together. Ars Legendi follows a conventional client-server architecture, but with an unconventional data layer. Instead of a traditional database, the system stores all content and user data as structured JSON files. This simplifies deployment and makes the data portable: you can inspect and edit every piece of content directly.

High-level component map

The system has three layers. First, a **data pipeline** (run offline, in Python) that transforms raw Latin text into annotated JSON using NLP. Second, a **server** (Node/Express) that serves that data through a REST API and handles authentication and student event logging. Third, a **client** (React) that renders the reading interface, vocabulary trainer, grammar quizzes, mastery dashboard, and teacher analytics.

The data pipeline feeds into the server's static data directory:

Data pipeline (offline, Python)	Server (Node + Express)	Client (React)

`Raw text`
`↓`
`Stanza NLP processing`
`↓`
`Token annotations (UD)`
`↓`
`Construction detection`
`↓`
`JSON export`
	`Caesar routes  /api/caesar`
	`Auth routes  /api/auth`
	`Class routes  /api/classes`
	`Student data  /api/student`
	
	`JSON file storage`
	`/data/caesar/`
	`/data/pilot/`
		`Vocab Trainer (3-phase)`
		`Grammar Practice`
		`Reading Guide + Word Inspector`
		`Mastery Page (Elo)`
		`Teacher Dashboard`
		
		`Core libraries:`
		`attemptEvents, eloRating,`
		`adaptiveFeedback, storage`

Technology stack

Layer	Technology	Purpose
Frontend	React 18 + Vite	Single-page application with fast HMR
Styling	TailwindCSS	Utility-first CSS framework
Routing	React Router	Client-side navigation
Backend	Node.js + Express	REST API server
Storage	JSON files	Portable, version-controllable data
NLP	Stanza (Python)	Universal Dependencies parsing for Latin
Deployment	Vercel + Railway	Frontend (Vercel) and backend (Railway) hosting
Auth	Custom JWT	HMAC-SHA256 with 7-day token expiration

Key architectural decisions

Separation of content and logic. The learning engine (session management, mastery tracking, adaptive feedback) is entirely decoupled from the content it operates on. Vocabulary banks, sentence bundles, and construction tags are loaded from JSON files and passed into generic functions. This means adapting the system to a new text requires only swapping data files.

JSON file storage. Rather than using a database, I store all data as JSON files. This has three advantages. First, the data is human-readable and Git-trackable. Second, the system works offline or with simple file hosting. Third, there is no database schema to migrate when data structures evolve. The tradeoff is that JSON files do not support concurrent writes efficiently, but for an educational platform with modest user counts, this is acceptable.

Backend-ready abstraction. Although the current implementation stores mastery data in localStorage, all storage operations go through an abstraction layer (*lib/storage.js*) that can be pointed at server endpoints. This allows the system to migrate to server-synced persistence without rewriting UI code.

3. The NLP pipeline: from raw text to structured data

The foundation of Ars Legendi is a Natural Language Processing pipeline that transforms raw Latin prose into richly annotated, machine-readable data. This section explains what Universal Dependencies are, how to run a Latin parser, and exactly what data structures the system expects.

3.1 Universal Dependencies and Latin parsing

Universal Dependencies (UD) is a framework for consistent grammatical annotation across languages. It defines a standard set of part-of-speech tags (UPOS), morphological features, and dependency relations that work for Latin, Greek, English, and many other languages. By using UD conventions, I ensured that the annotations are interoperable with a large ecosystem of tools and corpora.

The UD annotation layers

A UD-annotated sentence contains four layers of information for each token:

- **Lemmatization:** the dictionary headword (e.g., *amavit* → *amo*)
- **Part-of-speech (UPOS):** a coarse tag like NOUN, VERB, ADJ, ADP, CCONJ
- **Morphological features:** fine-grained attributes like Case=Acc, Number=Sing, Tense=Perf, Voice=Act
- **Dependency parse:** the syntactic head of each token and the relationship (nsubj, obj, obl, advmod, etc.)

Together, these layers give us everything needed to identify grammatical constructions. For example, detecting an ablative absolute requires finding a participle and a noun both in the ablative case, with the noun serving as the subject of the participle. All of that information is encoded in UD annotations.

Example: a single token in UD format

```
{
  "id": 1,
  "text": "Gallia",
  "form": "Gallia",
  "lemma": "Gallia",
  "upos": "NOUN",
  "xpos": "n-s---fn-",
  "head": 2,
  "deprel": "nsubj",
  "feats": {
    "Case": "Nom",
    "Gender": "Fem",
    "Number": "Sing"
  },
  "start": 0,
  "end": 6
}
```

This token tells us: *Gallia* is the surface form, *Gallia* is also the lemma (it is a proper noun), it is tagged as NOUN, it is nominative singular feminine, and syntactically it serves as the subject (nsubj) of token 2.

The start/end fields give character offsets for highlighting in the UI.

Why UD matters for construction detection

The dependency parse is what makes automated construction detection possible. Consider an indirect statement (accusative subject + infinitive after a verb of saying or thinking). Without parsing, you would need to enumerate all possible word orders. With UD, you simply look for:

- A verb of saying/thinking (a known list of lemmas)
- A dependent with deprel *ccomp* or *xcomp* that is an infinitive (VerbForm=Inf)
- An accusative noun or pronoun linked to that infinitive as its subject

The parser handles word order for you. Whether the accusative comes first, last, or is embedded in a relative clause, the dependency arcs point to the same structural relationship.

3.2 Token-level annotation structure

The raw output of a UD parser is typically in CoNLL-U format (a tab-separated text format). For web applications, I convert this to JSON. Ars Legendi expects each sentence to be stored as a bundle containing the original text, an array of tokens, and metadata:

```
{
  "sid": "1.2.3",          // Sentence ID: Chapter.Section.Sentence
  "chapter": 1,
  "index": 5,              // Position within chapter
  "text": "His rebus cognitis Caesar legiones in Galliam duxit.",
  "tokens": [
    { "id": 1, "form": "His", "lemma": "hic", "upos": "DET",
      "feats": {"Case":"Abl","Number":"Plur","Gender":"Neut"},
      "head": 2, "deprel": "det", "start": 0, "end": 3 },
    { "id": 2, "form": "rebus", "lemma": "res", "upos": "NOUN",
      "feats": {"Case":"Abl","Number":"Plur","Gender":"Fem"},
      "head": 3, "deprel": "obl", "start": 4, "end": 9 },
    // ... remaining tokens
  ],
  "translation": "With these matters having been learned, Caesar led..."
}
```

Critical fields for downstream processing

Field	Type	Purpose
id	integer	1-indexed position in sentence (used for head references)
form	string	Surface form as it appears in the text
lemma	string	Dictionary headword for glossary lookup
upos	string	Universal POS tag (NOUN, VERB, ADJ, etc.)
feats	object	Morphological features (Case, Number, Tense, Mood, Voice, etc.)
head	integer	ID of syntactic parent (0 = root)
deprel	string	Dependency relation to head (nsubj, obj, obl, advmod, etc.)
start / end	integer	Character offsets in original string for span highlighting

Implementation note: The start and end offsets are not produced by all parsers. If your parser outputs only tokens without character positions, you must reconstruct them by walking through the original string and matching each token. This is non-trivial when the parser normalizes punctuation or whitespace; you may need careful string matching to handle edge cases.

3.3 Practical implementation with Stanza

Stanza is my recommended tool for Latin parsing. It is a Python library developed by Stanford NLP that provides state-of-the-art accuracy on UD treebanks, including the Latin ITTB, PROIEL, and Perseus corpora. It handles tokenization, lemmatization, POS tagging, morphological analysis, and dependency parsing in a single pipeline.

Installation and basic usage

```
# Install Stanza
pip install stanza

# Download the Latin model (one-time)
import stanza
stanza.download('la')

# Initialize the pipeline
nlp = stanza.Pipeline('la', processors='tokenize,mwt,pos,lemma,depparse')

# Process a sentence
doc = nlp("Gallia est omnis divisa in partes tres.")

# Access token annotations
for sentence in doc.sentences:
    for word in sentence.words:
        print(f"{word.text}: lemma={word.lemma}, upos={word.upos}, "
              f"head={word.head}, deprel={word.deprel}")
        print(f"  feats: {word.feats}")
```

Converting Stanza output to the expected JSON format

Stanza outputs Document objects with nested Sentence and Word objects. You need to convert these to the JSON structure described above. Here is the conversion function I use:

```
def stanza_to_json(doc, chapter, base_sid):
    """Convert a Stanza Document to Ars Legendi sentence bundles."""
    bundles = []

    for sent_idx, sentence in enumerate(doc.sentences):
        text = sentence.text

        tokens = []
        for word in sentence.words:
            # Parse feats string into dict (Stanza returns "Case=Nom|Number=Sing")
            feats = {}
            if word.feats:
                for pair in word.feats.split('|'):
                    if '=' in pair:
                        k, v = pair.split('=', 1)
                        feats[k] = v

            tokens.append({
                'id': word.id,
                'form': word.text,
                'lemma': word.lemma,
                'upos': word.upos,
                'xpos': word.xpos,
                'feats': feats,
                'head': word.head,
                'deprel': word.deprel,
                'start': word.start_char,
                'end': word.end_char
            })

        bundles.append({
            'sid': f"{chapter}.{base_sid + sent_idx}",
            'chapter': chapter,
            'index': sent_idx,
            'text': text,
            'tokens': tokens
        })

    return bundles
```

Handling multi-word tokens and enclitics

Latin has enclitics (-*que*, -*ne*, -*ve*) that attach to words but are separate syntactic units. Stanza's multi-word token processor splits these: *populusque* becomes [*populus*, *que*]. This is correct for syntax but complicates character offsets, since both subtokens share the same span in the original string. I handle this by assigning the full span to the first subtoken and marking subsequent subtokens as having zero width, though you could also store multi-word token spans separately.

Alternative: UDPipe. If you prefer a command-line tool or need faster batch processing, UDPipe is an excellent alternative. It produces CoNLL-U output directly, which you can convert to JSON with a simple script. The tradeoff is slightly lower accuracy on some Latin constructions.

4. Construction detection: automated grammar tagging

With UD-parsed sentences in hand, I wrote detectors that identify grammatical constructions. This is the heart of what makes Ars Legendi's grammar support systematic rather than ad-hoc. Each detector is a function that takes a sentence bundle and returns an array of construction spans.

4.1 Rule-based detection architecture

I use rule-based detectors rather than machine learning classifiers. This choice is deliberate: rules are transparent (you can explain exactly why a construction was tagged), debuggable (you can trace failures to specific conditions), and editable (teachers can refine rules based on their pedagogical priorities). The tradeoff is that rules require manual authoring and may miss edge cases that a trained classifier would catch.

The detector interface

Every detector receives a sentence bundle and returns an array of construction objects. A construction object specifies the type, the token span, confidence, and optionally a subtype or evidence:

```
{
  "type": "ablative_absolute",
  "subtype": null,
  "start": 0,                    // First token index (inclusive)
  "end": 2,                      // Last token index (inclusive)
  "highlight_spans": [[0, 2]],   // May be discontinuous
  "confidence": 0.95,
  "evidence": {                  // Optional debug info
    "participle_id": 3,
    "subject_id": 2
  }
}
```

The construction types

Ars Legendi recognizes 11 construction types. I chose this set to cover the major grammatical structures students encounter in Caesar while remaining tractable for rule-based detection:

Type	Description	Key detection signals
cum_clause	Temporal/causal clauses with cum	cum + subjunctive verb, clause boundary
abl_abs	Ablative absolute	Ablative noun + ablative participle, syntactic independence
indirect_statement	Accusative + infinitive	Verb of saying/thinking + infinitive + acc. subject
purpose_clause	ut/ne + subjunctive	ut/ne introducing subjunctive, no result-clause signal
result_clause	ut + subjunctive with signal	tam/ita/tantus in main clause + ut + subjunctive
relative_clause	qui/quae/quod clauses	Relative pronoun with dependent clause
subjunctive_rel	Relative with subjunctive	Relative pronoun + subjunctive verb

Type	Description	Key detection signals
conditional	If-then structures	si/nisi + verb, protasis/apodosis pairing
gerund	Verbal noun	Gerund form (VerbForm=Ger)
gerundive	Verbal adj. of necessity	Gerundive form (VerbForm=Gdv)
gerund_flip	Gerundive replacing gerund + obj	Gerundive agreeing with former object

4.2 Implementing core detectors

Below are conceptual implementations of three representative detectors: ablative absolute, indirect statement, and purpose clause. These illustrate the general pattern of querying token features and dependency relations to identify constructions.

Ablative absolute detector

An ablative absolute consists of a noun (or pronoun) in the ablative case serving as the subject of a participle, also in the ablative. The key insight is that this construction is syntactically detached from the main clause. The participle typically has a dependency relation like *advcl* to the main verb, not a core argument relation.

```
def detect_ablative_absolute(sentence):
    constructions = []
    tokens = sentence['tokens']
    token_map = {t['id']: t for t in tokens}

    for token in tokens:
        # Look for participles in the ablative
        if token['upos'] != 'VERB':
            continue
        if token['feats'].get('VerbForm') != 'Part':
            continue
        if token['feats'].get('Case') != 'Abl':
            continue

        participle = token

        # Find the subject of this participle (should also be ablative)
        subject = None
        for t in tokens:
            if t['head'] == participle['id'] and t['deprel'] == 'nsubj':
                if t['feats'].get('Case') == 'Abl':
                    subject = t
                    break

        if not subject:
            continue

        # Verify syntactic independence
        if participle['deprel'] in ['nsubj', 'obj', 'iobj']:
            continue

        # Calculate span (min to max token index)
        span_tokens = [participle['id'], subject['id']]
        for t in tokens:
            if t['head'] == participle['id']:
                span_tokens.append(t['id'])

        start = min(span_tokens) - 1
        end = max(span_tokens) - 1

        constructions.append({
            'type': 'abl_abs',
            'start': start,
            'end': end,
            'confidence': 0.9,
            'highlight_spans': [[start, end]],
            'evidence': {
                'participle_id': participle['id'],
                'subject_id': subject['id']
            }
        })

    return constructions
```

Indirect statement detector

Indirect statement in Latin uses an accusative subject with an infinitive after verbs of saying, thinking, or perceiving. The detector must identify head verbs from a known list, find an infinitive dependent, and locate an accusative that serves as the subject of that infinitive.

```
HEAD_VERBS = {
    'dico', 'puto', 'credo', 'scio', 'video', 'audio', 'sentio',
    'intellego', 'cognosco', 'existimo', 'arbitror', 'iubeo',
    'nuntio', 'respondeo', 'nego', 'aio', 'fateor', 'confiteor'
}

def detect_indirect_statement(sentence):
    constructions = []
    tokens = sentence['tokens']

    for token in tokens:
        if token['lemma'] not in HEAD_VERBS:
            continue

        head_verb = token

        for t in tokens:
            if t['head'] != head_verb['id']:
                continue
            if t['deprel'] not in ['ccomp', 'xcomp']:
                continue
            if t['feats'].get('VerbForm') != 'Inf':
                continue

            infinitive = t

            # Find accusative subject of the infinitive
            acc_subject = None
            for subj in tokens:
                if subj['head'] == infinitive['id'] and subj['deprel'] == 'nsubj':
                    if subj['feats'].get('Case') == 'Acc':
                        acc_subject = subj
                        break

            if not acc_subject:
                for subj in tokens:
                    if subj['head'] == head_verb['id'] and subj['deprel'] == 'obj':
                        if subj['feats'].get('Case') == 'Acc':
                            acc_subject = subj
                            break

            span_ids = [head_verb['id'], infinitive['id']]
            if acc_subject:
                span_ids.append(acc_subject['id'])

            start = min(span_ids) - 1
            end = max(span_ids) - 1
            constructions.append({
                'type': 'indirect_statement',
                'start': start,
                'end': end,
                'confidence': 0.85 if acc_subject else 0.6,
                'highlight_spans': [[start, end]]
            })

    return constructions
```

Purpose clause detector

Purpose clauses are introduced by *ut* (positive) or *ne* (negative) with a subjunctive verb. The challenge is distinguishing purpose from result clauses, which also use *ut* + subjunctive. The key signal is the presence of a degree word (*tam*, *ita*, *tantus*, etc.) in the main clause for result.

```
RESULT_SIGNALS = {'tam', 'ita', 'tantus', 'talis', 'tot', 'adeo', 'sic'}

def detect_purpose_clause(sentence):
    constructions = []
    tokens = sentence['tokens']
    has_result_signal = any(t['lemma'] in RESULT_SIGNALS for t in tokens)

    for token in tokens:
        if token['lemma'] not in ['ut', 'ne']:
            continue

        subordinator = token

        subj_verb = None
        for t in tokens:
            if t['head'] == subordinator['id'] or subordinator['head'] == t['id']:
                if t['upos'] == 'VERB' and t['feats'].get('Mood') == 'Sub':
                    subj_verb = t
                    break

        if not subj_verb:
            continue

        if has_result_signal and subordinator['lemma'] == 'ut':
            continue

        clause_tokens = [subordinator['id'], subj_verb['id']]
        for t in tokens:
            if t['head'] == subj_verb['id']:
                clause_tokens.append(t['id'])

        start = min(clause_tokens) - 1
        end = max(clause_tokens) - 1

        constructions.append({
            'type': 'purpose_clause',
            'start': start,
            'end': end,
            'confidence': 0.85,
            'highlight_spans': [[start, end]]
        })

    return constructions
```

A note on conditionals. Conditional sentences are among the hardest constructions to detect automatically. Latin conditionals span a wide range of types (simple fact, future-more-vivid, future-less-vivid, present and past contrary-to-fact), and the parser must correctly pair protasis and apodosis, which may be separated by considerable intervening material. My detector handles the common cases, but edge cases involving nested conditions, mixed types, or implied *si*-clauses required significant tuning. If you are implementing conditionals for a new text, I recommend consulting the original detection code on the GitHub repository, where the full logic and fallback heuristics are documented.

4.3 Span calculation and highlighting

Constructions must be highlighted in the UI. This requires converting token indices to character offsets. Some constructions are discontinuous: a relative clause may have its antecedent in one position and the clause itself separated by intervening words. The *highlight_spans* field handles this by storing an array of [start, end] pairs rather than a single span:

```
def token_span_to_char_span(sentence, token_start, token_end):
    tokens = sentence['tokens']
    char_start = tokens[token_start]['start']
    char_end = tokens[token_end]['end']
    return [char_start, char_end]

# For discontinuous constructions, return multiple spans:
# highlight_spans: [[0, 3], [7, 10]]
```

Debugging tip: When a detector produces unexpected results, visualize the dependency tree. Many UD parsers have visualization tools, or you can render trees with the displaCy library. This makes it much easier to understand why a rule matched or failed.

5. Data architecture and file structures

Ars Legendi stores all content as JSON files in the */server/data/* directory. This section documents the structure of each file type so you can create equivalent files for a new text.

5.1 The sentence bundle format

The core data file is the sentence bundle file (e.g., *dbg1_sentences.json*). Each entry contains the text, tokens, detected constructions, and a reference to the translation:

```
{
  "sid": "1.2.3",
  "chapter": 1,
  "index": 5,
  "text": "His rebus cognitis Caesar legiones in Galliam duxit.",
  "tokens": [
    { "id": 1, "form": "His", "lemma": "hic", ... },
    { "id": 2, "form": "rebus", "lemma": "res", ... },
    ...
  ],
  "constructions": [
    {
      "type": "abl_abs",
      "start": 0,
      "end": 2,
      "highlight_spans": [[0, 2]],
      "confidence": 0.9
    }
  ],
  "translation": "With these matters having been learned, Caesar led..."
}
```

File organization by purpose

File	Size	Purpose
dbg1_ud.json	~2.8 MB	Full UD token data for all sentences
dbg1_sentences.json	~61 KB	Sentence text and metadata (without full tokens)
dbg1_translations.json	~77 KB	English translations keyed by sentence ID
dbg1_constructions.json	~309 KB	Construction tags extracted for grammar practice
dbg1_chapter_vocab_ok.json	~297 KB	Target vocabulary assigned to chapters
caesar_lemma_glosses_MASTER.json	~521 KB	Dictionary definitions for all lemmas
dbg1_lemma_index.json	~973 KB	Form → Lemma lookup (for Word Inspector)
dbg1_form_index.json	~1.0 MB	Lemma → Forms lookup (for vocabulary examples)

5.2 Vocabulary data structures

Vocabulary is stored in a chapter-indexed format that tracks each word's first appearance, frequency, part of speech, and an example context:

```
{
  "lemma": "duco",
  "chapter": 1,
  "firstChapter": 1,
  "upos": "VERB",
  "count": 23,
  "gloss_short": "lead, consider",
  "dictionary_entry": "duco, ducere, duxi, ductus: lead, conduct; consider",
  "example": {
    "sid": "1.2.3",
    "token_index": 7,
    "form": "duxit"
  }
}
```

The vocabulary file serves four purposes: building practice sessions by chapter, displaying word counts in the UI, linking to example sentences, and generating distractor options for multiple-choice quizzes.

Generating distractors

Multiple-choice vocabulary quizzes need plausible wrong answers. I generate distractors by selecting words that share the same part of speech and approximate frequency but have different meanings. Distractors are computed during session construction, not stored in the vocabulary file:

```
def generate_distractors(target_lemma, vocab_bank, n=3):
    target = vocab_bank[target_lemma]

    candidates = [
        v for v in vocab_bank.values()
        if v['upos'] == target['upos']
        and v['lemma'] != target_lemma
        and v['gloss_short'] != target['gloss_short']
    ]

    candidates.sort(key=lambda v: abs(v['count'] - target['count']))
    return [c['gloss_short'] for c in candidates[:n]]
```

5.3 The glossary and lemma index

The glossary file (*caesar_lemma_glosses_MASTER.json*) provides dictionary definitions for every lemma in the text. Each entry includes the full dictionary form, short gloss, and optionally a longer definition or usage notes:

```
{
  "lemma": "duco",
  "dictionary_entry": "duco, ducere, duxi, ductus",
  "gloss_short": "lead, conduct; consider, think",
  "gloss_long": "To lead, bring, conduct; to draw (a sword); to consider, regard",
  "semantic_field": "motion, cognition"
}
```

Building a working glossary

One of the more labor-intensive steps is producing the glossary itself. I built mine by combining several sources. First, I extracted every unique lemma from the UD-parsed text. Then I looked up definitions using Whitaker's Words (a freely available Latin dictionary program that you can run locally or query through various online interfaces). For lemmas that Whitaker's missed or glossed poorly, I consulted the Perseus Digital Library's Lewis and Short entries. Finally, I reviewed the glosses by hand to ensure consistency and pedagogical usefulness, shortening long definitions and choosing the sense most relevant to Caesar's usage.

If you are building a glossary for a new text, the practical workflow is: (1) extract lemmas, (2) batch-query Whitaker's Words or a similar tool, (3) fill in gaps from Perseus or Logeion, and (4) do a manual review pass. This last step matters because automated dictionaries sometimes return overly broad or misleading glosses for words that have a specific meaning in context.

The lemma index: form-to-lemma lookup

When a student clicks on a word in the reading interface, the system needs to look up its lemma from the inflected form. The lemma index maps every attested form to its lemma(s). Note that some forms are ambiguous (e.g., *esse* could be the infinitive of *sum* or *edo*):

```
{
  "duxit": ["duco"],
  "esse": ["sum", "edo"],
  "rei": ["res"],
  "Galliam": ["Gallia"],
  ...
}
```

I generated this index by walking through all tokens in the parsed text and collecting unique form-lemma pairs. The Word Inspector uses it to retrieve definitions when a student clicks on a word.

6. The Word Inspector: interactive token analysis

The Word Inspector is the popup interface that appears when a student clicks on any word in the reading view. It displays the word's lemma, part of speech, morphological features, glossary definition, and syntactic role. Here is how the data flows from click to display.

Data flow on word click

When a user clicks a word, five things happen in sequence. The component retrieves the token object from the sentence data. It extracts the lemma. It calls the server's glossary endpoint. It formats the morphological features into a readable string. And it renders all of this in a popup.

Step	Action	Data
1	Retrieve token from sentence	token = sentence.tokens[7]
2	Extract lemma	lemma = token.lemma // "duco"
3	Fetch glossary definition	GET /api/caesar/glossary?lemma=duco
4	Format morphology for display	"3rd singular perfect active indicative"
5	Render Word Inspector popup	Assembled token + gloss + morphology

Formatting morphological features

The raw morphological features from UD parsing are stored as key-value pairs (Case=Nom, Number=Sing). For student-facing display, I convert these to readable descriptions. The formatter function handles this conversion based on the part of speech:

```
function formatMorphology(token) {
  const { upos, feats } = token;

  if (upos === 'NOUN' || upos === 'ADJ' || upos === 'PRON') {
    const case_ = feats.Case || '?';
    const number = feats.Number === 'Sing' ? 'singular' : 'plural';
    const gender = feats.Gender ? feats.Gender.toLowerCase() : '';
    return `${case_.toLowerCase()} ${number} ${gender}`.trim();
  }

  if (upos === 'VERB') {
    const person = feats.Person || '';
    const number = feats.Number === 'Sing' ? 'singular' : 'plural';
    const tense = feats.Tense || '';
    const voice = feats.Voice === 'Act' ? 'active' :
                  feats.Voice === 'Pass' ? 'passive' : '';
    const mood = feats.Mood === 'Ind' ? 'indicative' :
                 feats.Mood === 'Sub' ? 'subjunctive' :
                 feats.Mood === 'Imp' ? 'imperative' : '';

    if (feats.VerbForm === 'Part') {
      return `${tense.toLowerCase()} ${voice} participle, ` +
             `${feats.Case?.toLowerCase()} ${number}`;
    }
    if (feats.VerbForm === 'Inf') {
      return `${tense.toLowerCase()} ${voice} infinitive`;
    }

    return `${person}${ordinalSuffix(person)} ${number} ` +
           `${tense.toLowerCase()} ${voice} ${mood}`;
  }

  return upos.toLowerCase();
}
```

Component structure

The Word Inspector is implemented as a React component (*WordInspector.jsx*) that receives the clicked token and renders a popup. It makes an API call to fetch the glossary definition and formats the morphology client-side:

```
function WordInspector({ token, sentence, onClose }) {
  const [gloss, setGloss] = useState(null);

  useEffect(() => {
    fetch(`/api/caesar/glossary?lemma=${encodeURIComponent(token.lemma)}`)
      .then(res => res.json())
      .then(data => setGloss(data));
  }, [token.lemma]);

  const morphDescription = formatMorphology(token);
  const syntaxRole = describeDependency(token.deprel);

  return (
    <div className="word-inspector-popup">
      <div className="inspector-header">
        <span className="form">{token.form}</span>
        <span className="lemma">({token.lemma})</span>
      </div>
      <div className="inspector-morphology">{morphDescription}</div>
      <div className="inspector-syntax">Role: {syntaxRole}</div>
      {gloss && (
        <div className="inspector-definition">
          <strong>{gloss.dictionary_entry}</strong>
          <p>{gloss.gloss_long || gloss.gloss_short}</p>
        </div>
      )}
    </div>
  );
}
```

7. Vocabulary learning system

The vocabulary trainer is the most fully developed learning engine in Ars Legendi. It implements a three-phase learning model with mastery tracking and session construction that balances new material with review.

7.1 Three-phase learning model

Each vocabulary session progresses through three phases, each targeting a different level of learning:

Phase	Activity	Cognitive target	Completion criterion
1	Multiple choice	Recognition (passive)	All words attempted once
2	Retry missed items	Error correction	Each miss answered correctly
3	Typed recall from English	Production (active)	All words typed correctly

This progression mirrors how I would want a teacher to scaffold vocabulary instruction: introduce words, address mistakes immediately, then challenge the student to produce from memory. The three-phase structure ensures students cannot complete a session by luck; they must demonstrate understanding at multiple levels.

Phase transition logic

```
// Transition from Phase 1 to Phase 2
function endPhase1() {
  const misses = [];

  Object.entries(phaseLogs[1]).forEach(([lemma, attempts]) => {
    const lastAttempt = attempts[attempts.length - 1];
    if (lastAttempt === false) {
      misses.push(lemma);
    }
  });

  const phase2Queue = shuffle(misses);
  setPhase2Queue(phase2Queue);
  setScreen('phase1_report');
}

// Transition from Phase 2 to Phase 3
function endPhase2() {
  // Phase 2 ends when all misses are corrected
  setPhase(3);
  setPhase3Index(0);
  setScreen('phase3');
}
```

7.2 Session construction algorithm

When a student starts a session, the system decides which words to include. The session builder balances several goals: cover new material from selected chapters, optionally include previously seen words for review, and keep the set large enough to be meaningful without being overwhelming.

```
function buildSession(mode, selectedCategories) {
  // mode: 'new' | 'review' | 'mixed'
  const wordSet = {};
  const order = [];
  const breakdown = {};

  for (const category of selectedCategories) {
    const categoryWords = vocabBank.filter(w =>
      w.chapter === category || w.upos === category
    );

    const masteryMap = loadMasteryMap();
    let filtered = categoryWords;

    if (mode === 'new') {
      filtered = categoryWords.filter(w => !masteryMap[w.lemma]);
    } else if (mode === 'review') {
      filtered = categoryWords.filter(w => masteryMap[w.lemma]);
    }

    for (const word of filtered) {
      const distractors = generateDistractors(word.lemma, vocabBank);
      wordSet[word.lemma] = {
        english: word.gloss_short,
        distractors,
        entry: word.dictionary_entry,
        chapter: word.chapter,
        upos: word.upos
      };
      order.push(word.lemma);
    }
    breakdown[category] = filtered.length;
  }

  return { wordSet, order: shuffle(order), breakdown };
}
```

7.3 Performance logging and state management

Every attempt is logged to enable mastery tracking and adaptive feedback. I maintain two log structures: a global session log and phase-specific logs that track performance at finer granularity.

```
const [logs, setLogs] = useState({});
const [phaseLogs, setPhaseLogs] = useState({ 1: {}, 2: {}, 3: {} });

function handleAnswer(lemma, isCorrect) {
  const nextLogs = {
    ...logs,
    [lemma]: [...(logs[lemma] || []), isCorrect]
  };
  setLogs(nextLogs);

  const currentPhaseLog = phaseLogs[phase] || {};
  currentPhaseLog[lemma] = [...(currentPhaseLog[lemma] || []), isCorrect];
  setPhaseLogs({ ...phaseLogs, [phase]: currentPhaseLog });

  logAttemptEvent({
    skill: `VOCAB_${wordSet[lemma].upos}`,
    subskill: phase === 3 ? 'RECALL' : 'RECOGNIZE',
    itemId: lemma,
    correct: isCorrect,
    latency: responseTime
  });
}
```

Mastery persistence

At the end of a session, words answered correctly in Phase 3 are marked as mastered and stored in localStorage. The mastery map tracks how many times each word has been mastered, so the system can prioritize words that have never been mastered or were mastered long ago.

```
const MASTERY_KEY = 'vt_mastery_v1';

function awardMastery(lemma, english, entry) {
  const map = loadMasteryMap();
  const current = map[lemma] || { count: 0, english, entry, lastAt: 0 };

  current.count += 1;
  current.lastAt = Date.now();
  map[lemma] = current;
  saveMasteryMap(map);
}

function endSession() {
  Object.entries(phaseLogs[3]).forEach(([lemma, attempts]) => {
    if (attempts.every(a => a === true)) {
      awardMastery(lemma, wordSet[lemma].english, wordSet[lemma].entry);
    }
  });
}
```

8. Grammar practice system

The grammar practice system complements vocabulary training by drilling students on construction identification. It uses the construction tags generated by the detectors (Section 4) to create interactive exercises anchored to sentences from the text.

8.1 Quiz configuration architecture

Each grammar quiz is defined by a configuration object that specifies the construction types to practice, the exercise steps, and the mastery criteria. This declarative approach makes it easy to add new quiz types without modifying the quiz engine.

```
{
  "quizId": "cum_clause_quiz",
  "skillId": "cum_clause",
  "displayName": "Cum Clauses",
  "description": "Practice identifying temporal and causal cum clauses",
  "constructionTypes": ["cum_clause"],

  "steps": [
    {
      "type": "identify",
      "subskillId": "identify",
      "prompt": "Click on the cum clause in this sentence"
    },
    {
      "type": "self_check",
      "subskillId": "classify",
      "prompt": "What type of cum clause is this?",
      "options": [
        { "value": "temporal", "label": "Temporal (when)" },
        { "value": "causal", "label": "Causal (since/because)" },
        { "value": "concessive", "label": "Concessive (although)" }
      ]
    }
  ],

  "sentenceCount": 4,
  "passingThreshold": 0.6
}
```

Exercise step types

Step type	Student action	Feedback
identify	Click on tokens forming the construction	Highlight correct span if wrong
self_check	Select from multiple choice options	Reveal correct answer with explanation
classify	Categorize the construction subtype	Show explanation of classification
produce	Type the translation or transformation	Compare to expected answer

8.2 Construction-based exercise design

The quiz engine fetches sentences containing the target construction type, then renders each step. For identify steps, students click on words to select the construction span; the system compares their selection to the stored *highlight_spans*.

```
async function startGrammarQuiz(config) {
  const response = await fetch(
    `/api/caesar/examples?types=${config.constructionTypes.join(',')}`
  );
  const sentences = await response.json();
  const selected = shuffle(sentences).slice(0, config.sentenceCount);

  setQuizSentences(selected);
  setCurrentSentenceIndex(0);
  setCurrentStepIndex(0);
}

function handleIdentifySubmit(selectedTokens) {
  const sentence = quizSentences[currentSentenceIndex];
  const construction = sentence.constructions.find(
    c => config.constructionTypes.includes(c.type)
  );

  const expected = new Set();
  for (const [start, end] of construction.highlight_spans) {
    for (let i = start; i <= end; i++) expected.add(i);
  }

  const correct = setsEqual(expected, new Set(selectedTokens));

  logAttemptEvent({
    skill: `grammar:${config.skillId}`,
    subskill: 'identify',
    itemId: sentence.sid,
    correct
  });

  showFeedback(correct, construction);
}
```

9. The mastery and Elo rating system

Ars Legendi tracks student mastery using an Elo-based rating system. Originally designed for chess, Elo ratings provide a principled way to measure skill from performance data. I adapted the algorithm for educational skill measurement.

9.1 Elo algorithm adaptation for education

In chess, a player's rating predicts their probability of beating opponents. In education, I reframe this: a student's rating predicts their probability of answering items correctly. Items have their own difficulty ratings, analogous to opponent ratings.

The core formula

```
// Expected probability of correct answer
expected = 1 / (1 + 10^((itemDifficulty - studentRating) / 400))

// Update rating based on actual outcome (1 = correct, 0 = wrong)
newRating = currentRating + K * (actual - expected)

// K-factor: higher for new students (more volatile), lower for experienced
```

Implementation constants

```
const INITIAL_RATING = 1200;
const K_FACTOR_NEW = 40;       // Students with <20 attempts
const K_FACTOR_STABLE = 24;    // Students with 20+ attempts
const NEW_THRESHOLD = 20;
const MIN_RATING = 400;
const MAX_RATING = 2400;

function calculateNewRating(currentRating, itemDifficulty, correct, attemptCount) {
  const k = attemptCount < NEW_THRESHOLD ? K_FACTOR_NEW : K_FACTOR_STABLE;
  const expected = 1 / (1 + Math.pow(10, (itemDifficulty - currentRating) / 400));
  const actual = correct ? 1 : 0;
  let newRating = currentRating + k * (actual - expected);

  return Math.max(MIN_RATING, Math.min(MAX_RATING, newRating));
}
```

9.2 Skill levels and progression

I convert Elo ratings to human-readable skill levels for display. The thresholds are calibrated so that a student answering roughly 50% correctly at baseline will be rated as "learning," and a student consistently above 80% will reach "mastered."

Rating range	Level	Description
< 1200	Novice	Still learning fundamentals
1200 to 1399	Learning	Developing competence

Rating range	Level	Description
1400 to 1599	Proficient	Solid understanding
1600+	Mastered	Consistent accuracy

9.3 Item difficulty calibration

Items (vocabulary words, construction instances) also have difficulty ratings. Initially all items start at 1200. As students attempt them, I recalculate difficulties based on aggregate performance. An item that 70% of students answer correctly is easier than one only 20% answer correctly.

```
function calibrateItemDifficulty(itemId, allAttempts) {
  const itemAttempts = allAttempts.filter(a => a.itemId === itemId);

  if (itemAttempts.length < 5) return 1200;

  const correct = itemAttempts.filter(a => a.correct).length;
  const accuracy = correct / itemAttempts.length;

  // accuracy 0.5 -> difficulty 1200
  // accuracy 0.7 -> difficulty ~1000
  // accuracy 0.3 -> difficulty ~1400
  const difficulty = 1200 + 400 * Math.log10((1 - accuracy) / accuracy);

  return Math.max(MIN_RATING, Math.min(MAX_RATING, difficulty));
}
```

10. Adaptive feedback engine

The adaptive feedback engine monitors student behavior and triggers interventions when it detects problematic patterns. This is what makes the system behave like a teacher: it notices when a student is struggling, guessing, or plateauing, and responds with appropriate support.

10.1 Behavioral pattern detection

The engine analyzes recent attempts to detect seven behavioral patterns:

Pattern	Detection signal	Interpretation
highLatency	Average response time >15 seconds	Student is struggling, needs support
hintDependency	>50% of attempts use hints	Over-reliant on scaffolding
guessing	Quick (<3s) wrong answers	Not reading carefully, rushing
inconsistent	Alternating correct/wrong on same items	Unstable knowledge
stagnating	Accuracy flat despite 20+ attempts	Needs a different approach
momentum	5+ correct in a row, improving speed	On a roll, can push harder
fatigued	Accuracy declining this session	May need a break

Detection implementation

```
function detectPatterns(recentAttempts, sessionAttempts) {
  const patterns = [];

  // High latency
  const avgLatency = mean(recentAttempts.map(a => a.latency));
  if (avgLatency > 15000) {
    patterns.push({
      type: 'highLatency',
      severity: avgLatency > 20000 ? 'high' : 'medium',
      data: { avgLatency }
    });
  }

  // Guessing
  const quickWrong = recentAttempts.filter(a => !a.correct && a.latency < 3000);
  if (quickWrong.length > recentAttempts.length * 0.3) {
    patterns.push({ type: 'guessing', severity: 'high', data: { count: quickWrong.length } });
  }

  // Momentum
  const lastFive = recentAttempts.slice(-5);
  if (lastFive.length === 5 && lastFive.every(a => a.correct)) {
    const latencies = lastFive.map(a => a.latency);
    const improving = latencies.every((l, i) => i === 0 || l <= latencies[i-1] * 1.1);
    if (improving) {
      patterns.push({ type: 'momentum', severity: 'positive', data: { streak: 5 } });
    }
  }

  // ... additional pattern detectors
  return patterns;
}
```

10.2 Intervention logic and coach triggers

When patterns are detected, the coach trigger system selects an appropriate intervention. Interventions range from gentle encouragement to restructuring the practice session.

Pattern	Recommended action	UI response
highLatency	SLOW_DOWN_SET	Shorter sets, encourage deliberation
hintDependency	NO_HINTS_CHALLENGE	Disable hints for next 5 items
guessing	SLOW_DOWN_SET	Add confirmation step before submit
stagnating	REVIEW_MISSES	Show detailed feedback on weak items
momentum	MOMENTUM_SET	Congratulate, optionally increase difficulty
fatigued	SUGGEST_BREAK	Prompt to take a break

11. Teacher analytics and class insights

Teachers have access to a dashboard that aggregates student data into actionable insights. The analytics surface class-wide patterns, identify students who need support, and generate specific reteaching recommendations with pedagogical guidance.

Class-level aggregation

```
function generateClassInsights(studentDataArray, skills) {
  const insights = {
    skillSummaries: {},
    reteachDecisions: [],
    misconceptions: [],
    studentAlerts: []
  };

  for (const skill of skills) {
    const ratings = studentDataArray
      .map(s => s.skills[skill.id]?.eloRating || null)
      .filter(r => r !== null);

    if (ratings.length === 0) continue;

    const avgRating = mean(ratings);
    const belowProficient = ratings.filter(r => r < 1400).length;
    const pctBelow = belowProficient / ratings.length;

    insights.skillSummaries[skill.id] = {
      avgRating,
      distribution: {
        novice: ratings.filter(r => r < 1200).length,
        learning: ratings.filter(r => r >= 1200 && r < 1400).length,
        proficient: ratings.filter(r => r >= 1400 && r < 1600).length,
        mastered: ratings.filter(r => r >= 1600).length
      }
    };

    // Flag for reteaching if >40% of class is below proficient
    if (pctBelow > 0.4) {
      insights.reteachDecisions.push({
        skillId: skill.id,
        name: skill.displayName,
        reason: `${Math.round(pctBelow * 100)}% of class below proficiency`,
        suggestion: RETEACH_TEMPLATES[skill.id]?.suggestion,
        keyPoints: RETEACH_TEMPLATES[skill.id]?.keyPoints
      });
    }
  }
  return insights;
}
```

Reteaching templates

I included built-in pedagogical templates for each skill that provide teachers with specific reteaching strategies and key points to emphasize:

```
const RETEACH_TEMPLATES = {
  'grammar:purpose_clause': {
    name: 'Purpose Clause',
    suggestion: 'Use paired examples showing purpose vs. result side-by-side',
    keyPoints: [
      'Purpose = intention (answers "why?")',
      'Result = outcome (answers "what effect?")',
      'No signal word (tam/ita) for purpose',
      'Both use ut + subjunctive, but context differs'
    ]
  },
  'grammar:abl_abs': {
    name: 'Ablative Absolute',
    suggestion: 'Practice identifying the two-part structure (noun + participle)',
    keyPoints: [
      'Always contains a noun AND a participle in ablative',
      'Grammatically independent from main clause',
      'Common with perfect passive participles'
    ]
  },
  // ... templates for all 11 construction types
};
```

12. Replication guide: adapting for new texts

This section provides a step-by-step walkthrough for adapting Ars Legendi to a new Latin text. The framework is designed to be text-agnostic: you swap the content data, and the learning engine remains unchanged.

12.1 Scope selection and text preparation

Start by choosing a manageable scope. A single book or a few chapters is ideal for a first adaptation.

Scope selection:

- Choose an author and work (e.g., Cicero, *In Catilinam* I)
- Define chapter or section boundaries
- Obtain a clean digital text (Perseus Digital Library, The Latin Library, etc.)
- Verify text quality: no OCR errors, consistent formatting
- Obtain or create translations with 1:1 sentence alignment

Text preparation:

- Split the text into a sentence-per-line format
- Assign stable sentence IDs (e.g., Cat1.1.1, Cat1.1.2)
- Create a chapter metadata file (title, sentence count, etc.)
- Review sentence boundaries carefully: fix run-on sentences and split compound sentences where appropriate

Prose vs. poetry

Ars Legendi was designed for prose (Caesar). Poetry is significantly harder to process for several reasons: word order is freer, which breaks parser assumptions; constructions may be discontinuous across line boundaries; ellipsis is common, leaving implicit elements; and meter influences word choice, complicating vocabulary selection.

Recommendation: Start with prose. Once you have experience with the pipeline, poetry adaptations are possible but require additional detector tuning and UI work for discontinuous spans.

12.2 Data pipeline execution

Once your text is prepared, work through the following pipeline steps in order.

1. NLP processing

- Install Stanza: `pip install stanza`
- Download the Latin model: `stanza.download('la')`
- Run the pipeline on all sentences
- Export to JSON with tokens, lemmas, feats, heads, and deprels

- Add character offsets (start/end) to each token
- Handle multi-word tokens and enclitics

2. Construction detection

- Run all 11 detectors on the parsed sentences
- Review a sample of the output for accuracy
- Tune detector thresholds if needed
- Export constructions to `[text]_constructions.json`

3. Vocabulary extraction

- Extract unique lemmas with occurrence counts
- Assign lemmas to chapters based on first appearance
- Obtain glossary definitions (Whitaker's Words, Perseus, Logeion)
- Generate form-to-lemma and lemma-to-form indices
- Create distractor pools grouped by part of speech

4. Translation alignment

- Match English translations to sentence IDs
- Verify 1:1 alignment (every sentence has exactly one translation)
- Export to `[text]_translations.json`

5. File assembly

- Create a `/server/data/[yourtext]/` directory
- Place all JSON files with consistent naming
- Update server routes to point to the new data directory

12.3 Frontend and backend adaptation

Once the data files are prepared, update the application to serve them.

Server updates

```
// Option A: Update existing routes (quick)
// In /server/routes/caesar.mjs, change:
const DATA_DIR = path.join(__dirname, '../data/caesar');
// To:
const DATA_DIR = path.join(__dirname, '../data/[yourtext]');

// Option B: Create a new route module (cleaner)
// Copy caesar.mjs to [yourtext].mjs and update paths
// In server/index.js, add:
import yourTextRoutes from './routes/[yourtext].mjs';
app.use('/api/[yourtext]', yourTextRoutes);
```

Client updates

```
// 1. Create new page component
//    Copy /client/src/pages/CaesarDBG1.jsx to [YourText].jsx
//    Update API calls from /api/caesar/* to /api/[yourtext]/*

// 2. Add route in App.jsx
<Route path="/[yourtext]" element={<YourText />} />

// 3. Update navigation in Navbar.jsx
<Link to="/[yourtext]">[Your Text Name]</Link>

// 4. Update grammar lessons if construction types differ
//    Edit /client/src/data/grammarLessons.js

// 5. Update quiz configurations if needed
//    Edit /client/src/data/grammarQuizConfigs.js
```

Testing checklist

Before deploying, verify each of these:

- Server starts without errors
- `GET /api/[yourtext]/chapters` returns the chapter list
- `GET /api/[yourtext]/sentenceBundle?sid=1.1` returns valid data
- `GET /api/[yourtext]/glossary?lemma=X` returns a definition
- Vocab trainer loads words for selected chapters
- Grammar practice fetches sentences with constructions
- Word Inspector displays morphology and definitions
- Mastery tracking persists across sessions
- Teacher analytics show class data correctly

Appendix A

Complete data file reference

File name	Description
[text]_ud.json	Full Universal Dependencies token annotations
[text]_sentences.json	Sentence bundles with text, tokens, constructions
[text]_translations.json	English translations keyed by sentence ID
[text]_constructions.json	Extracted construction tags for grammar practice
[text]_chapter_vocab.json	Vocabulary targets assigned to chapters
[text]_lemma_glosses.json	Dictionary definitions for all lemmas
[text]_lemma_index.json	Form → Lemma lookup for Word Inspector
[text]_form_index.json	Lemma → Forms lookup for vocabulary examples

Appendix B

Construction type catalog

Type key	Display name	Subtype examples
cum_clause	Cum clause	temporal, causal, concessive
abl_abs	Ablative absolute	—
indirect_statement	Indirect statement	—
purpose_clause	Purpose clause	ut, ne, relative
result_clause	Result clause	—
relative_clause	Relative clause	indicative
subjunctive_relative_clause	Subjunctive relative	purpose, characteristic
conditional	Conditional	simple, future-less-vivid, contrary-to-fact
gerund	Gerund	—
gerundive	Gerundive	passive periphrastic, attraction
gerund_gerundive_flip	Gerund/gerundive flip	—