TEACHING PACK · CLINCODER LEARNER SERIES · VOLUME 3

Define-XML: the Data's Promise

The machine-readable catalog every submission carries · ItemDefs, codelists, the stylesheet, and why the namespace is the contract. Learn by interrogating a real tool with AI · predict, ask, verify, break, rebuild. No prior Python and no prior CDISC required.

Prepared for	Bhanoji Duppada & team
Grounded in	NCI-EVS / CDISC sources · the ClinCoder CT layer on VPS2
Lessons	12 lessons in 3 parts, one AI-tutor loop each
Basis commits	sdtm_mapper 3f312c22a281c525e86f521fd46506891e01a887
sdtm_review	518eb50afdc224885fb3050cb9fded4ce021f5b1
Built	03 August 2026 · synthetic data only

How to use this book

This is not a reference manual — it is a **COURSE**. It teaches one thing: define.xml as a promise about the data — how the promise is written, why the namespace line is the contract, and how to check the promise against the data instead of trusting the writer that made it.

Every lesson runs the same loop, borrowed from the Learner's Companion: **PREDICT** (guess before reading), **Ask** (read, or let Claude teach it with the tutor prompt in Part 3), **Verify** (run the snippet — all standalone, stdlib only), **Break** (do the break-it exercise), and **Rebuild** (write the toy from memory).

THE ONE RULE THAT CARRIES THE WHOLE SUBJECT: validate against the published schema, never against your own writer. Part 2 tells the true story of two define-writing arms in our own codebase — one correct, one welded to a namespace that does not exist in the standard — and the reconciliation module that trusted the wrong one.

Tool examples come from the real ClinCoder define pipeline on VPS2 at the basis commits on the cover; standards facts come from CDISC, FDA and PharmaSUG sources cited inline. All data is synthetic.

Part 1 — The concepts

Before Part 2 walks you into a real define writer running on this machine, you need three ideas: what a define.xml actually is and why a regulator opens it first, which version and namespace strings make one readable to the world's tools, and how the rendered page you see in a browser relates to the record you actually submit — plus the mismatches validators catch every day. Everything here uses synthetic examples and facts taken directly from the published standard, the FDA's conformance guide, and two industry papers, so nothing in Part 2 will look mysterious.

Lesson 1 — The catalog the reviewer opens first

PREDICT BEFORE YOU READ: A REGULATORY REVIEWER RECEIVES A SUBMISSION CONTAINING DOZENS OF DATASET FILES, AN ANNOTATED CRF, REVIEWER'S GUIDES, AND ONE FILE CALLED DEFINE.XML. WHICH FILE DO THEY OPEN FIRST — AND WHY IS IT NOT A DATASET? COMMIT TO AN ANSWER; CHECK YOURSELF AT THE END.

The store catalog analogy

Imagine walking into a vast warehouse store with no signage: thousands of shelves, no map, no labels, no index. Everything you need is in there somewhere, but finding it means opening every box. Now imagine the same store handing you a catalog at the door: every aisle listed, every product named, described, and priced, with a note on where it came from and how it was made.

A clinical trial submission is that warehouse. The datasets are the shelves — rows and rows of values with terse eight-character column names like `AESEV` and `LBORRES`. And define.xml is the catalog: a machine-readable document that describes every dataset, every variable, every codelist of allowed values, every derivation, and every supporting document in the submission. The reviewer reads the catalog before touching the shelves, because without it a column called `AESEV` is just letters — the catalog says what it means, what values it may take, and where it came from.

This is not a metaphor the industry would find fanciful; it is how the standard describes itself. CDISC — the standards body for clinical data — says Define-XML "transmits metadata that describes tabular dataset structure" and "is required by the United States Food and Drug Administration (FDA) and the Japanese Pharmaceuticals and Medical Devices Agency (PMDA) for every study in each electronic submission" (cdisc.org). A 2025 industry paper puts it in plainer terms: the define.xml "provides essential metadata about the submitted datasets, allowing regulatory reviewers to easily access, interpret, and validate the clinical data," and is "a mandatory file required by regulatory authorities such as the US FDA and the Japan PMDA" (pharmasug.org).

And here is the sentence that should recalibrate how seriously you take a "metadata file." The FDA's Study Data Technical Conformance Guide (TCG, June 2026 edition) says the data definition file "is considered arguably the most important part of the electronic dataset submission for regulatory review" (fda.gov). Not the data — the description of the data. The same section warns that "an insufficiently documented data definition file is a common deficiency that reviewers have noted," and tells sponsors to provide "complete detail in this file, especially for the specifications pertaining to derived variables," with the codelist and origin for each variable "clearly and easily accessible."

The operational rules, also from the TCG: one separate data definition file per dataset type per study — SDTM, SEND, and ADaM each get their own define — submitted "in XML format, i.e., a properly functioning define.xml," with a printable define.pdf provided if the XML cannot be printed. The reviewer's guide (ADRG) gives reviewers "a single point of entry" alongside the define, but the TCG is

explicit that a reviewer's guide "does not eliminate the requirement to submit a complete and informative define.xml file."

One honest footnote before we go on. You will hear teachers (including this one) say "define first, data second." That slogan is a teaching phrase, not TCG language; what the sourced documents actually establish is the underlying idea — reviewers rely on the define.xml to access, interpret, and validate the data, and the FDA calls it arguably the most important part of the dataset submission. The phrase is a mnemonic; the citations are the facts.

The anatomy, at postcard level

Open a define.xml in a text editor and you will find one consistent skeleton, verified against a published walkthrough of the CDISC example file (pharmasug.org). At the top sits `ODM` — the root element, because Define-XML is built as an extension of CDISC's Operational Data Model. Inside it, one `Study` element with the study's name and protocol. Inside that, one `MetaDataVersion` — and that is the catalog proper. Everything interesting lives inside MetaDataVersion, and each element type answers one question:

- **ItemGroupDef** — defines one dataset (domain): its name, description, structure, class, keys, and the list of variables it contains. One per dataset.
- **ItemDef** — defines one variable: its label, data type, length, origin, and links to its codelist or derivation method.
- **CodeList** — the controlled terminology for a variable: the permitted coded values and their human-readable decodes. (If you read Volume 2 of this series, this is where CT gets recorded in a submission.)
- **MethodDef** — documents the computational algorithm for a derived variable, so a reviewer can see how it was produced. This is exactly where the TCG demands "complete detail."
- **def:ValueListDef** — value-level metadata: the same variable described differently depending on context — a lab result column, for instance, described per lab test.
- **def:WhereClauseDef** — the condition (for example, lab test code equals "GLUC") that says when a particular value-level definition applies.
- **def:CommentDef** — reusable comment text that datasets and variables can point to.
- **def:leaf** — a link (an href) to a physical file in the submission: a dataset file or a referenced PDF.
- **def:AnnotatedCRF / def:SupplementalDoc** — pointers to the annotated CRF and other supporting documents, so they are reachable from the catalog. The aCRF itself, per the TCG, is a PDF that maps the data collection fields on the forms to the SDTM variables they populate.

Notice the two prefixes. Plain names like `ItemGroupDef` come from the ODM base standard; names starting `def:` are Define-XML's own additions layered on top. That layering becomes the whole plot of Lesson 2.

Ten lines of stdlib: the skeleton as a tree

You do not need Python fluency here — read it as "build the chain, then print it indented."

```
import xml.etree.ElementTree as ET

ODM = "http://www.cdisc.org/ns/odm/v1.3"

root  = ET.Element(f"{{{ODM}}}ODM", ODMVersion="1.3.2", FileType="Snapshot")
study = ET.SubElement(root, f"{{{ODM}}}Study", OID="STD.TEACH")
mdv   = ET.SubElement(study, f"{{{ODM}}}MetaDataVersion", OID="MDV.1", Name="Teaching catalog")

ET.SubElement(mdv, f"{{{ODM}}}ItemGroupDef", OID="IG.AE", Name="AE")
ET.SubElement(mdv, f"{{{ODM}}}ItemDef", OID="IT.AE.AESEV", Name="AESEV", DataType="text")

def show(el, depth=0):
    print("  " * depth + el.tag.split("}")[1])   # tag without its namespace
    for child in el:
        show(child, depth + 1)

show(root)
```

Run it and five lines come back — `ODM`, then `Study` indented once, then `MetaDataVersion`, then `ItemGroupDef` and `ItemDef` side by side inside it. That is the entire chain of command of every define.xml ever submitted: ODM → Study → MetaDataVersion → the catalog entries. A real one has hundreds of entries; the skeleton never changes.

Break it on purpose: the catalog that lies

The catalog's authority comes from one property: it matches the shelves. So break that. Suppose the submission contains three datasets, but someone deleted the AE dataset's ItemGroupDef from the define — perhaps a regeneration went wrong, perhaps the dataset was added late and nobody updated the catalog:

```
datasets_in_submission = ["DM", "AE", "LB"]
itemgroups_in_define   = ["DM", "LB"]        # AE's ItemGroupDef is gone

for ds in datasets_in_submission:
    if ds in itemgroups_in_define:
        print(f"{ds}: cataloged")
    else:
        print(f"{ds}: IN SUBMISSION BUT NOT IN CATALOG — the define now lies")
```

The AE file still sits in the folder, perfectly intact. But the catalog now claims the submission contains two datasets when it contains three. A reviewer navigating by the define will never be routed to AE; a reviewer who stumbles on the file will find a dataset with no declared variables, no codelists, no origins — undocumented data, the exact "common deficiency" the TCG warns about. A wrong catalog is worse than a missing one, because tools and people trust catalogs. Keeping the catalog and the shelves in agreement is, as you will see in Lesson 3 and throughout Part 2, most of the actual work.

Quiz 1

1. The FDA calls define.xml "arguably the most important part of the electronic dataset submission." The datasets contain the actual patient data. Reconcile those two statements in one sentence.
2. A submission contains SDTM datasets and ADaM datasets for one study. How many define files does the TCG expect, and why doesn't a thorough reviewer's guide remove the need for any of them?
3. Which element defines a dataset, which defines a variable, and which element family (`def:leaf`, `def:AnnotatedCRF`, `def:SupplementalDoc`) makes the define a hub rather than an island?

TAKEAWAY: define.xml is the machine-readable catalog of a submission — datasets, variables, codelists, derivations, and documents — required by FDA and PMDA per dataset type per study, and its entire value rests on one property: it must match the data it describes.

Sources: https://www.cdisc.org/standards/data-exchange/define-xml ; https://pharmasug.org/proceedings/2025/MM/PharmaSUG-2025-MM-277.pdf ; https://www.fda.gov/media/153632/download

---

Lesson 2 — Versions and namespaces: the contract line

PREDICT BEFORE YOU READ: DEFINE-XML VERSION 2.1 WAS PUBLISHED IN 2019. YOU OPEN A V2.1 FILE AND LOOK AT THE ROOT ELEMENT FOR ITS VERSION NUMBER. WHAT NUMBER DO YOU EXPECT TO SEE? HINT: IT IS NOT 2.1.

A short version history

The official CDISC page lists the whole lineage (cdisc.org): Define-XML v1.0 arrived on 5 February 2005, v2.0 on 24 April 2014, and v2.1 was originally published on 15 May 2019. Since then v2.1 has accumulated point releases — v2.1.1 (26 March 2021) through v2.1.10 (29 October 2025), with the current listed release **V2.1.11, DATED 6 APRIL 2026**, an update reflecting CDISC Controlled Terminology Package 61. Per the same pages, v2.1 contains all the material of the v2.0 specification plus a documented set of changes: an updated approach to declaring a variable's origin (`def:Origin`), identification of the standards and controlled-terminology versions being referenced, sub-class support, better SEND support, and fixes to v2.0 errata.

So which version do you use? Not the newest by reflex. A 2025 industry paper summarizes the acceptance rule: "According to the latest data standards catalog published by each agency, the current acceptable versions of define.xml for submitting metadata for CDISC SDTM and ADaM datasets are define.xml v2.0 and define.xml v2.1, with the specific version depending on the start date of the clinical study" (pharmasug.org). Both FDA and PMDA publish those catalogs (fda.gov; pmda.go.jp). The FDA's TCG adds a preference: "It should be noted that Define-XML version 2.0 or later is strongly preferred" (fda.gov). Two honest notes: CDISC's own page does **NOT** label v2.0 (or even v1.0) as deprecated — older versions simply sit lower in the version history — and the exact support-end dates by study start date are something you read from the "Support Ends" column of the current FDA Data Standards Catalog spreadsheet, not something to trust from any book, including this one.

And the future? As of this writing, **NO DEFINE-XML V3.0 EXISTS.** The current release is v2.1.11 and near-term activity is v2.1.x maintenance. CDISC's ODM v2.0 exists, with XML and JSON serializations, and is positioned as the base for next-generation extensions — but no ODM v2.0-based Define standard has been released (cdisc.org; lexjansen.com). Meanwhile Dataset-JSON v1.1, CDISC's JSON exchange format for the tabular data itself, links optionally to Define-XML: per its official spec, "Each Dataset-JSON dataset can optionally reference a Define-XML document containing more complete metadata for the dataset," with the dataset's `itemGroupOID` doubling as a foreign key to an `ItemGroupDef` OID, each column's `itemOID` pointing to an `ItemDef` OID, and optional `metaDataRef`, `studyOID`, and `metaDataVersionOID` fields completing the linkage (cdisc-org.github.io). The pointers are all optional — Dataset-JSON never mandates a define — but notice the design: the catalog from Lesson 1 is becoming the join key that other formats reference.

The answer to the predict question: two layers, two numbers

Here is the fact this whole lesson orbits. Every Define-XML 2.x document — v2.0 and v2.1 alike — is built as an **EXTENSION OF ODM VERSION 1.3.2**, CDISC's Operational Data Model. The CDISC pages for every

2.x release list ODM-XML v1.3.2 as its related standard (cdisc.org). So the root element of a Define-XML v2.1 file declares `ODMVersion="1.3.2"` — the version of the base layer, not of the define layer riding on top.

The layering shows up most precisely in the namespace declarations. From the published CDISC example file reproduced in the 2025 paper (pharmasug.org, Figure 2), a define.xml root carries:

- `xmlns="http://www.cdisc.org/ns/odm/v1.3"` — the ODM base vocabulary: `ODM`, `Study`, `MetaDataVersion`, `ItemGroupDef`, `ItemDef`, `CodeList`, `MethodDef` all live here;
- `xmlns:def="http://www.cdisc.org/ns/def/v2.0"` for a v2.0 file, or `http://www.cdisc.org/ns/def/v2.1` for v2.1 — Define-XML's own additions: `def:ValueListDef`, `def:WhereClauseDef`, `def:leaf`, and friends (cdisc.org);
- `xmlns:xlink="http://www.w3.org/1999/xlink"` — the W3C linking vocabulary that `def:leaf` hrefs use.

Read those three lines as the document's passport. A namespace looks like a URL, but no software ever fetches it — it is compared as a string, character for character, and that comparison is the first thing any schema-aware tool does when it opens the file. Match, and the tool recognizes every element inside. Mismatch by one character, and the same tool sees a well-formed file full of elements it has never heard of. That is why the namespace line is best understood as the **contract line**: it is the sentence in which your file promises to be speaking a published, schema-backed dialect. Part 2 of this book visits a real codebase where two define writers disagreed about exactly this string — and what that did to the checker downstream — so hold the thought.

Even the standard has bugs

Lest you conclude that published standards are infallible and only implementers err: Define-XML v2.1 shipped with conformance rule #266, which required an `ItemGroupDef`'s description and an `ItemDef`'s description to exactly match the standard's dataset and variable labels. That rule conflicts with SDTMIG v3.3 and later, where labels are informative rather than normative — and CDISC's own guidance now says validation engines "should not implement Define-XML v2.1 conformance rule #266" for SDTMIG v3.3, v3.4, and newer (cdisc.org). A rule published in the standard, later walked back by the standard's own maintainers. The lesson generalizes: read errata, not just specifications, and when a validator and a standard disagree, check whether the rule itself has a known defect before you "fix" your file.

The contract check, in twelve lines

The cheapest schema-shaped check you will ever write: does the root element's namespace equal the expected constant?

```
import xml.etree.ElementTree as ET

EXPECTED_ODM = "http://www.cdisc.org/ns/odm/v1.3"

define_text = """<ODM xmlns="http://www.cdisc.org/ns/odm/v1.3"
     xmlns:def="http://www.cdisc.org/ns/def/v2.1"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     ODMVersion="1.3.2" FileType="Snapshot"/>"""

root = ET.fromstring(define_text)
actual = root.tag[1:].split("}")[0] if root.tag.startswith("{") else "(none)"
print("root namespace:", actual)
assert actual == EXPECTED_ODM, f"WRONG NAMESPACE: {actual!r}"
print("contract line holds — a v1.3-aware tool will recognize this file")
```

ElementTree spells a namespaced tag as `{namespace}Tag`, so slicing the root's tag between `{` and `}` recovers the declared namespace, and one `assert` compares it to the published constant. In a real pipeline `define_text` would be a file read from disk; nothing else changes. (One safety habit to file away now: the string above is our own literal, so plain ElementTree is fine — but a define.xml that arrives from outside, from a sponsor or a vendor, should be parsed with the `defusedxml` package instead, because stdlib XML parsers accept external-entity tricks by default. Part 2 shows a real module doing exactly that.)

Break it on purpose

In the snippet, change one character of the namespace inside `define_text` — say `odm/v1.3` to `odm/v1.4`. The XML is still perfectly well-formed; a human skimming it would swear nothing meaningful changed; `ODMVersion` still says 1.3.2. But rerun the code and the assert fires, printing the exact impostor string. That single character moved the file out of the published dialect and into a dialect **no schema on Earth defines** — every element inside now carries a family name no validator recognizes. The check that catches it is one string comparison. Part 2, Lesson 5 shows what happens in a real system when nobody writes that one comparison; here, just internalize the asymmetry: a one-character edit, an entire toolchain blinded, a five-line guard.

Quiz 2

1. A colleague opens a Define-XML v2.1 file, sees `ODMVersion="1.3.2"`, and files a bug: "version attribute is wrong, should say 2.1." Explain, in one sentence, why the file is correct.
2. Your company wants to standardize on v2.1 for everything. What determines whether v2.0 or v2.1 is the acceptable version for a given study, and where do you look it up?
3. Conformance rule #266 required define descriptions to exactly match standard labels, and CDISC later said engines should not implement it for SDTMIG v3.3+. What general habit does this episode teach about standards documents?

TAKEAWAY: Define-XML 2.x is a layer riding on ODM 1.3.2 — the root says `ODMVersion="1.3.2"` and `xmlns=...odm/v1.3` no matter which 2.x you use — the acceptable version is set by the agency catalogs against your study start date, no v3 exists yet, and the namespace line is a byte-for-byte contract that one string comparison can verify.

Sources: https://www.cdisc.org/standards/data-exchange/define-xml ; https://www.cdisc.org/standards/data-exchange/define-xml/define-xml-v2-1 ; https://www.cdisc.org/standards/data-exchange/define-xml/define-xml-v2-1-3 ; https://pharmasug.org/proceedings/2025/MM/PharmaSUG-2025-MM-277.pdf ; https://www.fda.gov/media/153632/download ; https://www.fda.gov/regulatory-information/search-fda-guidance-documents/data-standards-catalog ; https://

www.pmda.go.jp/english/review-services/reviews/0002.html ; https://www.cdisc.org/standards/data-exchange/odm-xml/odm-v2-0 ; https://www.cdisc.org/standards/data-exchange/dataset-json/dataset-json-v1-1 ; https://cdisc-org.github.io/DataExchange-DatasetJson/doc/dataset-json1-1.html ; https://www.lexjansen.com/phuse-us/2025/ds/PAP_DS05.pdf ; https://docs.openclinica.com/oc4/how-and-when-to-use-apis/cdisc-odm-xml/

Lesson 3 — The stylesheet, and the pitfalls you will actually meet

PREDICT BEFORE YOU READ: YOU DOUBLE-CLICK A DEFINE.XML AND YOUR BROWSER SHOWS A HANDSOME, NAVIGABLE PAGE — TABLES OF DATASETS, CLICKABLE VARIABLES, FORMATTED CODELISTS. YOU OPEN THE SAME FILE IN A TEXT EDITOR AND SEE NOTHING BUT ANGLE BRACKETS. WHICH ONE IS THE SUBMISSION — THE PRETTY PAGE OR THE BRACKETS?

One file, two faces

The pretty page is a rendering. A define.xml can reference an XSL stylesheet — a transform that turns the raw XML into an HTML-like view; as one industry guide puts it, “An XSL Stylesheet can be referenced by the define.xml file to allow the XML file to be rendered in a more [readable form]” (quanticate.com). The reference is a one-line processing instruction near the top of the file — in standard practice, `<?xml-stylesheet type="text/xsl" href="define2-1.xsl"?>` — telling the browser: before you show this, transform it with that.

Where does the stylesheet come from? The original `define2-1.xsl` was published by CDISC in May 2019 as part of the official Define-XML v2.1 package (downloadable from cdisc.org, sign-in required). A maintained, MIT-licensed open copy — plus localized variants and conversion scripts — lives in Lex Jansen’s GitHub repository for the 2.1 stylesheets, with a counterpart repo for 2.0 (github.com). And the FDA cares where it sits: the TCG tells sponsors to reference the stylesheet listed in the Catalog “and place the corresponding style sheet in the same submission folder as the define.xml file” (fda.gov) — a broken href means a reviewer double-clicks and gets brackets.

Two cautions from the same guide, and they matter more than the mechanics. First, there is no single CDISC-governed stylesheet that must be used: “The CDISC Define-XML Standard does not dictate how a stylesheet should display Define-XML v2,” and the shipped file says outright that it may be “altered to satisfy alternate visualization needs” (quanticate.com). Second, stylesheets embed display assumptions — for example, rendering decisions for ISO 8601 values based on data type or on variable names ending in `--DUR` — so the rendered view can genuinely differ from the raw XML. (The lineage is its own trap: an early v1.0-era stylesheet was confusingly named `define2-0-0.xsl`.) Put those together and you have your predict answer: **the XML is the record; the render is a view.** Two different stylesheets can show the same file two different ways; validators, reviewers’ tools, and this book all read the brackets.

The pitfall parade

Because the define is a promise about the data, the recurring disease is drift: the data changes after the define is written. A 2025 paper lists the drivers — source data refreshes that change lengths, significant digits, and CT values; datasets and variables added by protocol amendments or SAP updates; derivation logic changes — each requiring a corresponding define.xml update (pharmasug.org). Validators exist to catch what slipped. Here are six real Pinnacle 21 Enterprise findings, rule IDs as documented in a 2022 Merck paper (pharmasug.org):

SD1231 — DECLARED LENGTH VS. ACTUAL LENGTH. The define says a variable's length is one number; the longest value in the data says another. The fix is updating the length in both variable-level and value-level metadata. This is drift in its purest form: the measurement was true once.

DD0074 — ORIGIN MISMATCH ACROSS LEVELS. A variable-level origin (where the value comes from) conflicts with the value-level origins beneath it. When one variable has multiple value-level origins, the fix is to omit the variable-level origin and let the specific ones speak.

DD0068 — LENGTH WHERE LENGTH IS MEANINGLESS. A length populated on a datetime variable (`--DTC`, which holds ISO 8601 text). Length metadata must be missing when the data type is not Integer, Float, or Text — a reminder that each attribute has rules about when it may exist, not just what value it holds.

OD0071 — MISSING SIGNIFICANT DIGITS. Float variables need significant-digits metadata; leave it off and the reviewer's tools cannot know how much precision the numbers claim.

SD0037 — CODELIST VS. DATA. The validator "will cross-check the codelist terms in the specifications with the corresponding values in [the data]": a codelist in the define promising values the datasets don't honor — or datasets containing values the codelist never mentions — is the classic define/data drift, flagged directly.

CT2001 / CT2002 — CODELIST VS. THE PUBLISHED CT. Terms are also cross-checked against the CDISC Controlled Terminology file itself; adding a term to a non-extensible codelist is an error. (Volume 2 of this series is entirely about why.)

Read the list again and notice they are all one disease: **TWO ARTIFACTS THAT SHOULD AGREE, DON'T** — define vs. data, level vs. level, define vs. published CT. The workflow answer, per the 2025 paper's final step, is mechanical: validate, correct, revalidate "until the report is free of errors and warnings" before submission. One honesty note: you will also hear of rules for duplicate OIDs and wrong/missing `def` namespaces; the public rule list was unreachable during this book's research, so no rule IDs are asserted for those — but the underlying requirement is real, because OIDs are the keys everything joins on (Lesson 2 showed Dataset-JSON using them as foreign keys), so a duplicate OID breaks referencing whatever any rule list says.

Where define sits on the spine

Volume 2 introduced the seven-stage spine every serious data tool walks: ingest, inventory, knowledge, propose, verify, review, deliver + evidence. Place define.xml on it and the placement explains this whole lesson. The define is a **STAGE 7** artifact — deliverable and evidence: it ships with the outputs and records what the submission contains, against which codelists, derived how. And reconciliation — checking the define against the produced data, which is what SD1231 and SD0037 do from outside and what Part 2's recon module does from inside — is **stage 5 verify, applied to your own promise**: deterministic re-derivation of whether each claim in the catalog is actually true of the shelves. A define nothing re-checks is a promise nobody audits; the pitfall parade above is what auditors find.

Exercise (no code)

Take any define.xml you can find — the CDISC example file, or one from Part 2's codebase — and open it twice: once in a browser (rendered by its stylesheet) and once in a text editor. List three things visible in the brackets that the rendered page hides or transforms. Candidates to look for: the namespace declarations from Lesson 2's contract line, the `xml-stylesheet` processing instruction itself, OID attributes that link elements together, and how a date-like value is displayed versus stored. Keep your list; in Part 2 you will watch a checker fail precisely on something only the brackets show.

Quiz 3

1. An agency reviewer and your browser can show the same define.xml differently. What two facts about stylesheets from this lesson explain how, and which representation is authoritative?
2. SD1231, SD0037, and DD0074 catch three different mismatches. Name the pair of artifacts each compares, and the one-word disease all three are symptoms of.
3. On the seven-stage spine, define.xml is a stage-7 artifact — yet this lesson claims stage 5 is what keeps it honest. Explain in one sentence.

TAKEAWAY: the stylesheet makes define.xml browsable, but the XML is the record and the render is only a view — and because the define is a promise about data that keeps changing, the essential habit is stage-5 verification of your own stage-7 artifact: validate, correct, revalidate until define and data agree.

Sources: https://www.quanticate.com/hubfs/Resources-2022/PDFs/CDISC%20Define.xml%20Guide.pdf ; https://github.com/lexjansen/define-xml-2.1-stylesheets ; https://github.com/lexjansen/define-xml-2.0-stylesheets ; https://www.cdisc.org/standards/data-exchange/define-xml/define-xml-v2-1 ; https://www.fda.gov/media/153632/download ; https://pharmasug.org/proceedings/2022/SS/PharmaSUG-2022-SS-023.pdf ; https://pharmasug.org/proceedings/2025/MM/PharmaSUG-2025-MM-277.pdf

---

Answers to Part 1 quizzes

Lesson 1

1. The datasets hold the data, but without the define a reviewer cannot reliably interpret any of it — which variable means what, which values are legal, how derived numbers were produced — which is why the TCG calls the file that unlocks the data "arguably the most important part of the electronic dataset submission for regulatory review."
2. **Two** — one data definition file per dataset type per study, so the SDTM datasets get one define and the ADaM datasets another. The reviewer's guide helps navigation ("a single point of entry"), but the TCG states it "does not eliminate the requirement to submit a complete and informative define.xml file."
3. `ItemGroupDef` defines a dataset; `ItemDef` defines a variable. The `def:leaf` / `def:AnnotatedCRF` / `def:SupplementalDoc` family holds links (hrefs) out to physical files — datasets, the annotated CRF, other supporting documents — which is what makes the define a hub the whole submission hangs off, rather than a self-contained list.

Lesson 2

1. Define-XML 2.x is an extension layered on ODM version 1.3.2, so `ODMVersion="1.3.2"` names the base layer's version — the define layer announces itself separately, in the `def` namespace (`.../ns/def/v2.1`); the two numbers were never supposed to match.
2. The study's **start date**, looked up against the agency's data standards catalog — FDA's Data Standards Catalog and PMDA's equivalent each list the acceptable Define-XML versions per study start date, with the FDA TCG adding that "version 2.0 or later is strongly preferred." Company preference doesn't override the catalog.
3. That standards documents themselves carry defects and corrections — so read the errata and current guidance, not just the specification, and when a validator flags your file against a rule,

check whether the rule itself has been walked back (as #266 was for SDTMIG v3.3+) before "fixing" a correct file.

Lesson 3

1. First, no single stylesheet is mandated — stylesheets may be swapped or customized, so two viewers can legitimately render one file differently. Second, stylesheets embed display assumptions (such as ISO 8601 rendering keyed to data types or `--DUR` names), so even one stylesheet can show something other than the literal stored value. The **XML itself** is authoritative; the render is a view.
2. SD1231 compares the define's declared length against the actual data; SD0037 compares the define's codelist terms against the values in the data; DD0074 compares variable-level origin against value-level origins within the define. The disease: **drift** (two artifacts that should agree, drifting apart).
3. Because a deliverable is only trustworthy if something re-derives its claims: stage-5 verification — recon against the produced data, plus validator rules like SD1231/SD0037 — is the mechanism that catches the drift that would otherwise turn the stage-7 catalog into a lie that ships.

Part 2 — Inside a real define writer

In Part 1 you learned what a define.xml is: the metadata document that travels with a submission and describes every dataset, every variable, every codelist — the table of contents a reviewer opens before they open a single row of data. Now we go where most tutorials never take you: inside a real, working define writer that runs on this machine, following one real (synthetic) variable from a mapping decision all the way into the finished XML.

And then, because this is a teaching book and not a brochure, we visit the scar. This codebase does not contain one define writer. It contains **TWO**, and they disagree — about a single string, sitting in the very first line of the file, that decides whether any standard tool on Earth can read the document at all. One arm gets it right. One arm gets it wrong. And the module whose whole job is checking defines is welded to the wrong one. That story is Lesson 5, and it teaches the most unforgiving idea in standards work.

Our running example is the same synthetic study as always: the Cardiology therapeutic area, adverse events domain, no real person anywhere. One variable carries the whole plot: **AETERM**, the reported term for the adverse event — the actual words describing what happened to the patient, like "PALPITATIONS ON EXERTION".

Lesson 4 — One variable's paper trail: AETERM from plan to define

PREDICT BEFORE YOU READ: THE FINISHED DEFINE.XML SAYS AETERM HAS `LENGTH="12"`. WHO DECIDED TWELVE — A PERSON FILLING IN A FORM, THE CDISC STANDARD, OR SOMETHING ELSE ENTIRELY?

Hold your answer. Let's follow the variable.

Three documents, one fact

When this system maps the raw Cardiology AE file, it decides that the raw column `event_term` feeds the SDTM variable AETERM. That single decision then appears in **three** different artifacts, each written for a different reader, and the entire product is that all three say the same thing.

ARTIFACT ONE: THE MAPPING PLAN — for humans in a review meeting. Open `/root/sdtm_review/data/mapping_plans/Cardiology/plan.md` and on line 81 (verified on disk this session) you find:

```
| **AETERM** | Req | Reported Term for the Adverse Event | `event_term` | `R2:alias` | a reviewer recorded this alias |
```

Read it as a sentence: AETERM is Required, its label is "Reported Term for the Adverse Event", it comes from raw column `event_term`, and rule `R2:alias` made the call because a reviewer once recorded that alias. Who decided, and on what grounds — that's provenance, and it rides along everywhere.

ARTIFACT TWO: THE DDS JSON — for machines. `/root/sdtm_review/services/dds_export.py` converts the plan into a "Data Definition Spec": define.xml, but as JSON, before it becomes XML. In `/root/sdtm_review/data/mapping_plans/Cardiology/dds.json` (verified this session), the same variable is now:

```
{
 "OID": "IT.AE.AETERM",
 "name": "AETERM",
 "description": "Reported Term for the Adverse Event",
 "mandatory": true,
 "role": "Topic",
 "dataType": "text",
 "length": 12,
 "origin": {"type": "Collected"}
}
```

Same label. `Req` has become `"mandatory": true`. And there is our mysterious number: `"length": 12`.

ARTIFACT THREE: DEFINE.XML ITSELF — for regulators and their software. `/root/sdtm_review/services/define_pipeline.py` hands the DDS JSON to CDISC's own generator (more on that in Lesson 5), and inside `/root/sdtm_review/data/mapping_plans/Cardiology/define.xml` (verified this session) the variable has reached its final form:

```
<ItemRef ItemOID="IT.AE.AETERM" Mandatory="Yes" OrderNumber="6" Role="Topic"/>
...
<ItemDef OID="IT.AE.AETERM" Name="AETERM" DataType="text" SASFieldName="AETERM" Length="12" ...>
```

Notice the split into two elements, because define.xml separates two questions. The **ITEMDEF** answers "what is AETERM?" — a text variable, twelve characters, this label. The **ItemGroupDef** for the AE dataset holds an **ItemRef** answering "how is AETERM used here?" — it's the sixth column, it's mandatory, its role is Topic. Definition versus usage. Keep that split in mind; a check in the real code depends on it.

So who decided twelve?

Nobody. Twelve was **MEASURED**. The anatomy book session traced it: the define pipeline reads the produced AE dataset and records the longest value actually present in `event_term` — twelve characters — falling back to a default only when nothing was observed. This session I verified the number itself on disk, agreeing across all three artifacts: plan, `dds.json` (`"length": 12`), and `define.xml` (`Length="12"`).

That is the difference between this define and the spec workbooks you may have filled in by hand. A hand-authored spec declares an intent ("AETERM shall be at most 200 characters"). This file reports a measurement of the data that actually exists. Which leads to the one sentence I want you to carry out of this lesson:

THE DEFINE IS A PROMISE ABOUT THE DATA. RECON CHECKS THE PROMISE.

Every attribute in that ItemDef is a claim a reviewer's software will take literally. `Length="12"` promises no value is longer than twelve. `Mandatory="Yes"` promises no record leaves it blank. `DataType="text"` promises what kind of thing lives there. A define.xml that makes false promises is worse than none at all — it tells the reviewer's tools to trust something untrustworthy. So this codebase has a module whose only job is checking the promises: `/root/sdtm_review/services/define_recon.py`, which parses one define.xml and one produced dataset and runs six cross-checks, DR001 through DR006 (plus DR000, "one of my inputs is unusable — I refuse to pretend"). Here is the length check, quoted from the real file:

```
#| skip-check  (verbatim excerpt from services/define_recon.py, reconcile)
        # DR003: value longer than declared Length
        if meta["length"] > 0:
            over = {v for v in values if len(v) > meta["length"]}
            if over:
                findings.append(_finding(
                    "DR003", "Error", var,
                    f"Values exceed define.xml Length={meta['length']}.",
```

One set comprehension. That's the whole idea: hold the promise in one hand, the data in the other, and compare.

Try it yourself — write a tiny define with the standard library

Nothing but stdlib `xml.etree.ElementTree`. This builds the AETERM skeleton — one ItemGroupDef with its ItemRef (usage), one ItemDef (definition) — in the correct namespace, which Lesson 5 will make dramatic:

```
import xml.etree.ElementTree as ET

ODM = "http://www.cdisc.org/ns/odm/v1.3"     # ODM 1.3.2 — remember this string
DEF = "http://www.cdisc.org/ns/def/v2.1"
ET.register_namespace("", ODM)
ET.register_namespace("def", DEF)

root  = ET.Element(f"{{{ODM}}}ODM", ODMVersion="1.3.2", FileType="Snapshot")
study = ET.SubElement(root, f"{{{ODM}}}Study", OID="STD.DEMO")
mdv   = ET.SubElement(study, f"{{{ODM}}}MetaDataVersion", OID="MDV.1", Name="Demo")

ig = ET.SubElement(mdv, f"{{{ODM}}}ItemGroupDef", OID="IG.AE", Name="AE",
                   Repeating="Yes")
ET.SubElement(ig, f"{{{ODM}}}ItemRef", ItemOID="IT.AE.AETERM", Mandatory="Yes")

ET.SubElement(mdv, f"{{{ODM}}}ItemDef", OID="IT.AE.AETERM", Name="AETERM",
              DataType="text", Length="12")

print(ET.tostring(root, encoding="unicode"))
```

Run it and read the output slowly. Every element name is written as `{namespace}Tag` — the curly-brace form is how ElementTree spells "this element belongs to that family." The printed XML shows `xmlns="http://www.cdisc.org/ns/odm/v1.3"` on the root, which quietly stamps that family name onto every element inside. Twelve lines of real code and you have the same skeleton the 86-kilobyte Cardiology define is built from.

Break it — make the define lie, then catch it

Now the fun part. Pretend the mapping changed — a longer event term arrived in the data — but nobody regenerated the define. The data moved; the promise didn't. Here is the whole drama in one screen, ending with the six-line recon check that catches it:

```
# the data (one AE variable, synthetic) — longest value is 24 characters
data   = {"AETERM": ["HEADACHE", "DIZZINESS",
                     "PALPITATIONS ON EXERTION"]}

# the define's promise — stale, still says 12
define = {"AETERM": {"length": 12}}

# the recon check: six lines, DR003 in miniature
for var, meta in define.items():
    longest = max(len(v) for v in data[var])
    if longest > meta["length"]:
        print(f"DR003 {var}: value of length {longest} "
              f"exceeds declared Length={meta['length']}")
        print("   offending value:", max(data[var], key=len))
```

Run it: `PALPITATIONS ON EXERTION` is 24 characters against a promised 12, and DR003 fires. Now un-break it — change `"length": 12` to `"length": 24` — and the check falls silent. Notice what you just did not need: no XML parsing, no schema, no framework. Reconciliation is conceptually a `max(len(...))` and a comparison. Everything else in `define_recon.py` — five more checks, defused XML parsing, capped findings — is careful engineering wrapped around comparisons this small.

One honest note: the real module compares against a real CSV and a real define.xml, and it caps how many offending values it quotes (ten) so a wholly-wrong column can't produce a megabyte of findings. Small mercies matter in tools people actually read the output of.

Quiz 4 (answers at the end of Part 2)

1. The same fact about AETERM appears in `plan.md`, `dds.json`, and `define.xml`. Why is it a feature that there are three artifacts rather than one — and what new obligation does having three create?
2. `Length="12"` in this pipeline is a measurement, not a declaration. Name one concrete way a measured length can still end up wrong by the time a reviewer reads it (hint: the break-it exercise).
3. In define.xml, `Mandatory="Yes"` lives on the ItemRef while `DataType` and `Length` live on the ItemDef. In one sentence, what is the logic of that split?

TAKEAWAY: A DEFINE.XML IS NOT PAPERWORK — IT IS A MACHINE-READABLE PROMISE ABOUT THE DATA, KEPT HONEST ONLY BECAUSE A SEPARATE, SIMPLER PROGRAM RE-MEASURES THE DATA AND COMPARES. METADATA THAT NOTHING CHECKS IS METADATA YOU CANNOT TRUST.

Lesson 5 — The scar: two arms, one wrong namespace

PREDICT BEFORE YOU READ: TWO PROGRAMS EACH WRITE A DEFINE.XML. BOTH FILES OPEN FINE IN A BROWSER, BOTH LOOK LIKE TEXTBOOK DEFINE-XML, AND A HUMAN SKIMMING THEM SIDE BY SIDE MIGHT NEVER SPOT A DIFFERENCE. YET EVERY STANDARD VALIDATOR ON EARTH CAN READ ONE AND NONE CAN READ THE OTHER. WHAT SINGLE STRING DECIDES WHICH IS WHICH?

First, what a namespace actually is

In XML, an element called `ItemDef` is not really called `ItemDef`. Its full name is `{namespace}ItemDef` — the tag plus a family name, declared once at the top of the file as `xmlns="..."` and inherited by everything inside. The namespace is a URI, but don't let that fool you: no software ever fetches it. It is compared as a **string of bytes, character for character**, like a password. When a validator, a viewer, or a reviewer's toolchain opens a define.xml, its very first act is to look for elements by full name. Right

family name: the document lights up. Wrong family name — by even one character — and the tool sees a well-formed file containing, as far as it is concerned, nothing it recognizes at all.

Which brings us to the two arms.

Arm one: the CDISC pipeline — the right family name

The path Lesson 4 followed is the modern arm. `/root/sdtm_review/services/define_pipeline.py` does not write XML itself; it is "a thin, honest shell around somebody else's tool" (its own docstring) that runs **CDISC's own generator** in a pinned checkout, then validates the result against **CDISC's own schema**, `define2-1-0.xsd`. Look at the opening of the file it produced — quoted from `data/mapping_plans/Cardiology/define.xml` on this disk, verified this session:

```
<ODM ... ODMVersion="1.3.2" FileType="Snapshot"
     Originator="360i Define-XML Team" SourceSystem="odmlib"
     xmlns="http://www.cdisc.org/ns/odm/v1.3"
     xmlns:def="http://www.cdisc.org/ns/def/v2.1" ...>
```

Here is the detail beginners find genuinely surprising, so slow down for it: this is a **DEFINE-XML 2.1** document, and yet its main namespace says `odm/v1.3` and its version says `1.3.2`. That is not a mistake. Define-XML v2.1 is built on top of ODM version 1.3.2 — the define layer adds its own vocabulary in the second namespace, `def/v2.1`, while the base document remains ODM 1.3.2. The version numbers of the two layers simply don't match, and were never supposed to. Anyone who "fixes" that mismatch by intuition breaks the file.

Arm two: the older writer — a family name that doesn't exist

Now the second arm. `/root/sdtm_review/services/define_xml.py` is an older adapter, still live — `/root/sdtm_review/scripts/batch_run.py` calls its `generate_define` once per domain in every batch run. It translates the mapper's spec into the shape consumed by a separate external tool, `/root/define-maker`, which does the actual writing. Open the adapter at lines 30–31 and read the constants, quoted exactly (verified this session):

```
#| skip-check  (verbatim, services/define_xml.py lines 30-31)
ODM_NS = "http://www.cdisc.org/ns/odm/v2.1"
DEF_NS = "http://www.cdisc.org/ns/def/v2.1"
```

And the generator agrees with it — `/root/define-maker/core/define_generator.py` line 24 declares the same `odm/v2.1` string (verified by grep this session), and the artifacts prove it ran that way. Here is a real file from a real batch run on this disk, `data/batch_runs/regression_check/domains/AE/define_AE.xml`, opening line verified this session:

```
<ODM xmlns:def="http://www.cdisc.org/ns/def/v2.1"
     xmlns="http://www.cdisc.org/ns/odm/v2.1"
     ODMVersion="2.1" FileOID="Study.SYNTH-001" FileType="Snapshot" ...>
```

`odm/v2.1`. It looks more modern than `odm/v1.3` — surely a Define-XML **2.1** document should live in an ODM **2.1** namespace? You can feel how the mistake happened; it is the intuitive "fix" from a paragraph ago. But per the anatomy book's finding: **there is no `odm/v2.1` namespace in the Define-XML 2.1 standard.** Define-XML 2.1 rides on ODM 1.3.2, namespace `odm/v1.3` — exactly what arm one emits. What I can verify directly on this box, and did this session, is the mechanical consequence: the string in arm two's code and artifacts matches no string in the XSD arm one validates against, so a schema check of an arm-two file against `define2-1-0.xsd` could never pass — and, tellingly, the arm-two path runs **no schema validation at all**, which is why nothing ever complained.

Both files are well-formed XML. Both look perfect. One wears a family name every agency tool recognizes; the other wears a family name that appears in no published schema anywhere. A file can be flawless in every visible way and still be unreadable to every tool that matters, because **IN STANDARDS WORK, THE NAMESPACE IS THE CONTRACT.**

The weld: the checker that only reads the wrong dialect

It gets better — or worse. Remember `define_recon.py` from Lesson 4, the module that checks the promises? Its namespace constant, line 38, quoted exactly (verified this session):

```
#| skip-check  (verbatim, services/define_recon.py line 38)
ODM = "{http://www.cdisc.org/ns/odm/v2.1}"
```

The promise-checker is welded to the **WRONG** arm. Today the pairing happens to work, because `batch_run.py` feeds recon the legacy files, and `v2.1`-tagged checker meets `v2.1`-tagged file. But point `reconcile` at the correct define — the CDISC-generated Cardiology file — and walk through what happens, because the failure shape is the real lesson:

1. `parse_define` opens the file and iterates `ItemDef` elements — using the full name `{...odm/v2.1}ItemDef`.
2. Every ItemDef in the file is named `{...odm/v1.3}ItemDef`. Wrong family name. The iterator finds **zero** elements.
3. No error is raised — an empty document and a wrong-namespace document are indistinguishable to the parser. `parse_define` calmly returns an empty index.
4. Check DR001 ("variable in dataset, absent from define") now compares the dataset against an empty define — and flags **every single variable** as undocumented.

The anatomy book calls this failure loud but wrong-loud, and that phrase is worth keeping: the report screams "your define declares nothing!" when the truth is "I read your define wearing the wrong glasses." A silent failure hides; a wrong-loud failure actively points investigators at the wrong suspect — the healthy file, instead of the checker's own hardcoded constant. (The fix, noted in the anatomy book, is two lines: try `odm/v1.3`, fall back to `odm/v2.1`, report which matched. As of this session, line 38 still says what it says.)

The rule the scar teaches

Ask how this survived so long, and the answer generalizes to everything you will ever build against a standard. The old arm's output was checked — by tools on the same arm. The writer wrote `odm/v2.1`; the checker checked for `odm/v2.1`; green lights all around. A closed loop of self-agreement. The one authority never consulted was the only one that counts: the schema published by the standards body. Arm one gets this right structurally — `define_pipeline.validate()` runs CDISC's `define2-1-0.xsd` over every file and reports `ok: false` on failure, and its docstring says why: "producing a define.xml that a regulator would reject is worse than producing none, and the failure is silent."

VALIDATE AGAINST THE PUBLISHED SCHEMA, NOT AGAINST YOUR OWN WRITER. Your own writer and your own checker can share a bug forever, in perfect harmony. The published schema is the one referee that didn't come to the game on your bus.

Break it — one character, total blindness

Take Lesson 4's skeleton and change the namespace by a single character — `v1.3` to `v1.4`. Then write the check that arm two never had. The check is five lines, and it is the cheapest insurance in this entire book:

```
import xml.etree.ElementTree as ET

EXPECTED = "http://www.cdisc.org/ns/odm/v1.3"
WRONG    = "http://www.cdisc.org/ns/odm/v1.4"   # one character off

root = ET.Element(f"{{{WRONG}}}ODM", ODMVersion="1.3.2")
ET.SubElement(root, f"{{{WRONG}}}ItemDef", Name="AETERM", DataType="text")
doc = ET.tostring(root, encoding="unicode")

# the 5-line check arm two never had
reparsed = ET.fromstring(doc)
actual   = reparsed.tag[1:].split("}")[0]              # family name off the root
found    = reparsed.findall(f"{{{EXPECTED}}}ItemDef")  # correct glasses
print(f"namespace: {actual}\nItemDefs visible to a correct reader: {len(found)}")
assert actual == EXPECTED, f"WRONG NAMESPACE: {actual!r}"
```

Run it and watch two things happen in order. First the print: the correct reader sees `0` ItemDefs — your AETERM definition is right there in the string, and simultaneously invisible; that is the recon failure from this lesson reproduced on your desk. Then the assert fires and names the exact wrong string. Compare-against-a-constant is all it takes; the tragedy of arm two is not that the check was hard, but that nobody wrote it. Now un-break it — set `WRONG` equal to `EXPECTED` — and the reader sees 1 ItemDef and the assert passes.

(For real inbound files, one more habit from the codebase: `define_recon.py` deliberately parses with `defusedxml` instead of plain ElementTree, because a define.xml often arrives from outside — a sponsor, a vendor — and stdlib parsers accept external-entity tricks by default. Parse your own files however you like; parse strangers' XML defensively.)

Quiz 5 (answers below)

1. Both arms produce well-formed, valid-looking XML. In one sentence, what makes the `odm/v2.1` file unusable anyway?
2. The recon module pointed at a correct define reports that every variable is undocumented. Why is this "wrong-loud" failure arguably more dangerous than a crash?
3. The two-arm bug survived because writer and checker agreed with each other. State the rule that breaks that kind of closed loop.

TAKEAWAY: A NAMESPACE IS A BYTE-FOR-BYTE CONTRACT, NOT A DECORATION — AND THE ONLY CHECK THAT MEANS ANYTHING IS THE ONE AGAINST THE STANDARD BODY'S PUBLISHED SCHEMA, BECAUSE YOUR OWN WRITER AND YOUR OWN CHECKER CAN SHARE THE SAME WRONG IDEA INDEFINITELY.

Answers to Part 2 quizzes

Quiz 4

1. Three readers need three shapes: a human in a meeting reads the plan table, machines consume the DDS JSON, and regulatory software reads define.xml. The obligation it creates is consistency — three copies of one fact can drift, which is exactly why the pipeline generates all three from one source and why recon exists to re-check the final promise against the data.
2. Staleness. The length was true when measured; if the data is regenerated (a longer `event_term` value arrives) and the define is not, the measurement becomes a false promise — the break-it exercise's `PALPITATIONS ON EXERTION` at 24 characters against a stale `Length="12"`. Only re-running the comparison (DR003) catches it.

3. The ItemDef describes what the variable is (definition — type, length, label, reusable anywhere), while the ItemRef describes how this dataset uses it (usage — position, mandatory-or-not, role); the same variable could be defined once and used differently in different datasets, so the two kinds of fact live on two different elements.

Quiz 5

1. Its root namespace, `http://www.cdisc.org/ns/odm/v2.1`, matches no namespace in the published Define-XML 2.1 standard (which is built on ODM 1.3.2, namespace `odm/v1.3`), so every schema-aware tool looks up elements by a full name that the file's elements simply do not have — and sees nothing.
2. A crash points at itself; wrong-loud output points away from itself. The report confidently blames the healthy file ("your define declares nothing") for a defect that lives in the checker's own hardcoded namespace constant — so the investigation starts in the wrong place, and confidence in a correct artifact is destroyed instead of the bug being found.
3. Validate against the published schema, not against your own writer. An external referee — the standard body's XSD, an independent tool — must sit in the loop, because any check you wrote yourself can inherit the very assumption your writer got wrong, and the two will pass each other forever.

Part 3 — Practice: interrogate, rebuild, adapt

You know what a define.xml is (Part 1) and you have walked one real variable from mapping plan to finished XML — and stood at the scar where two writers disagreed about a single namespace string (Part 2). Part 3 turns that knowledge into ability: first you learn to interrogate a define with an AI without ever trusting the AI, then you rebuild the reconciliation check from a spec, and finally you learn to reshape the same skeleton for three very different clients.

Lesson 6 — Interrogating a define with AI

PREDICT BEFORE YOU READ: YOU PASTE A DEFINE.XML EXCERPT INTO AN AI AND ASK "IS THIS DEFINE CORRECT?" IT ANSWERS, CONFIDENTLY AND IN DETAIL. WHAT IS THE ONE INPUT YOU DID NOT GIVE IT — WITHOUT WHICH NO ANSWER, YES OR NO, DESERVES YOUR TRUST?

Hold that thought. Part 2's spine was: **THE DEFINE IS A PROMISE ABOUT THE DATA; RECON CHECKS THE PROMISE.** An AI holding just the define holds one hand of a two-handed comparison — it can comment on form, but any verdict about truth is manufactured. The three prompts below turn that rule into paste-ready tools. Each is complete: no placeholders, synthetic data already inside.

Prompt (a) — THE TUTOR PROMPT

Use this when you want the AI to teach you Part 2's central distinction, anchored to a concrete file instead of floating theory.

```
You are a patient tutor for CDISC Define-XML. Teach me ONE lesson and
nothing more: the difference between an ItemDef (what a variable IS)
and an ItemRef (how a dataset USES it), and why the two live on
different elements.

Anchor everything to this synthetic define.xml excerpt. Do not invent
elements that are not here:

  <ODM xmlns="http://www.cdisc.org/ns/odm/v1.3"
       xmlns:def="http://www.cdisc.org/ns/def/v2.1"
       ODMVersion="1.3.2" FileType="Snapshot">
    <Study OID="STD.DEMO">
      <MetaDataVersion OID="MDV.1" Name="Demo Study Metadata">
        <ItemGroupDef OID="IG.AE" Name="AE" Repeating="Yes"
                      Purpose="Tabulation">
          <ItemRef ItemOID="IT.AE.AETERM" Mandatory="Yes"
                   OrderNumber="1" Role="Topic"/>
          <ItemRef ItemOID="IT.AE.AESEV" Mandatory="No" OrderNumber="2"/>
        </ItemGroupDef>
        <ItemDef OID="IT.AE.AETERM" Name="AETERM" DataType="text"
                 Length="12"/>
        <ItemDef OID="IT.AE.AESEV" Name="AESEV" DataType="text"
                 Length="8">
          <CodeListRef CodeListOID="CL.AESEV"/>
        </ItemDef>
        <CodeList OID="CL.AESEV" Name="Severity" DataType="text"/>
      </MetaDataVersion>
    </Study>
  </ODM>

Ground facts you must treat as given, even against your own memory:
- This is a Define-XML v2.1 document. Its base namespace is ODM v1.3
  and its ODMVersion is 1.3.2. That mismatch is CORRECT by design:
  Define-XML 2.1 is an extension built on ODM 1.3.2.
- Attribute names and namespace URIs are byte-for-byte contracts.

Structure of the session:
1. Explain the lesson in plain words using ONLY the excerpt above.
2. Quiz me with 3 questions, ONE at a time; wait for my answer.
3. If my answer is vague ("the metadata says so", "it's defined at the
   top"), do not accept it. Make me commit to the exact element name
   and attribute — e.g. not "the length" but "Length= on the ItemDef
   with OID IT.AE.AETERM" — then tell me if I am right.
4. Finish with one pencil-and-paper exercise on this excerpt (e.g.
   add a third variable correctly) and its worked answer.
```

Why it is built this way, one line per choice:

- **The excerpt is pasted, not described** — the AI teaches the file in the room, not a remembered average of every define it has read; Part 2's whole method was reading real artifacts, not summaries.
- **The namespace ground-fact is force-fed** — Lesson 5's scar: `odm/v1.3` under a 2.1 define looks like a bug and an unbriefed AI will helpfully "correct" it, exactly the intuition that produced the broken arm in `/root/sdtm_review/services/define_xml.py`.
- **Vague answers are drilled to exact element/attribute names** — because in this standard the names are the substance: a namespace one character off makes a file invisible (Lesson 5), so "it's in the metadata" is not an answer, `Length=` on `ItemDef OID="IT.AE.AETERM"` is.
- **ItemDef vs ItemRef as the one lesson** — Lesson 4's definition-versus-usage split, the single most transferable fact about the file's anatomy.

Prompt (b) — THE DEFINE REVIEW, done right

This is the important one. Notice what makes it different from "please review my define": it hands the AI **BOTH HANDS** of the comparison — the define excerpt and the measured facts about the dataset — and it forbids any verdict the pasted evidence cannot support.

```
You are reviewing a Define-XML excerpt against measured facts about the
dataset it describes. You work ONLY from what is pasted here. Your own
knowledge of CDISC standards may name rule families but may NEVER supply
a missing fact.

DEFINE EXCERPT (Define-XML v2.1; namespaces correct and not at issue):

  <ItemDef OID="IT.AE.AETERM" Name="AETERM" DataType="text" Length="12">
    <def:Origin Type="Collected"/>
  </ItemDef>
  <ItemDef OID="IT.AE.AESEV" Name="AESEV" DataType="text" Length="8">
    <CodeListRef CodeListOID="CL.AESEV"/>
  </ItemDef>
  <ItemDef OID="IT.AE.AESTDTC" Name="AESTDTC" DataType="datetime"
           Length="19"/>
  <CodeList OID="CL.AESEV" Name="Severity" DataType="text">
    permitted values: MILD | MODERATE | SEVERE   (non-extensible)
  </CodeList>

MEASURED DATA FACTS (from the produced AE dataset):

  AETERM   longest value present: 24 characters
           ("PALPITATIONS ON EXERTION")
  AESEV    distinct values: MILD, MODERATE, SEVERE, GRADE 4
           longest value present: 8 characters
  AESTDTC  values are ISO 8601 datetime text, e.g. 2026-01-15T09:30
  (No facts are provided about how any variable was collected: no aCRF
  page, no source column, no collection system evidence.)

RULE FAMILY KEY (use these labels in your table):
  SD1231  define Length vs actual data length
  SD0037  codelist values in define vs values actually in data
  CT2001/CT2002  codelist terms vs CDISC CT; additions to a
          non-extensible codelist
  DD0068  Length populated where the data type is not
          Integer/Float/Text (e.g. datetime)
  OD0071  float variable missing SignificantDigits
  DD0074  Origin declared at variable level conflicting with
          value-level origins / origin claims

OUTPUT: one markdown table, one row per CLAIM the define makes, columns
exactly:
  element (with OID) | the claim it makes | the pasted data fact that
  tests it | verdict | rule family

Verdict rules — non-negotiable:
- MATCHES        the pasted fact confirms the claim.
- MISMATCH       the pasted fact contradicts the claim; quote both sides.
- CANNOT VERIFY  MANDATORY whenever no pasted data fact tests the claim.
  Never replace it with a guess, a probability, or "likely fine".
Every claim gets exactly one verdict. End with one summary line that
counts verdicts and states plainly whether this define can be called
checked — it cannot, if any row says CANNOT VERIFY.
```

Run it and mark it like homework, because the planted evidence decides every row: AETERM's `Length="12"` against a measured 24 is MISMATCH (SD1231 — the exact stale-length drama from Part 2's break-it exercise). AESEV's codelist against a data value `GRADE 4` is MISMATCH (SD0037, with CT2001/

CT2002 in play because the list is non-extensible — no one may talk `GRADE 4` onto it). AESEV's `Length="8"` against a measured 8 is the one honest MATCHES. AESTDTC is the sleeper: a `Length` populated on a `datetime` variable is invalid regardless of the data (DD0068). And AETERM's `def:Origin Type="Collected"` must come back CANNOT VERIFY — the parenthetical in the data facts says so out loud — with DD0074 as the family. An AI that "confirms" the origin anyway has just failed the interrogation.

The design choices:

- **Both sides pasted** — Lesson 4's recon posture made literal: promise in one hand, measurement in the other. `/root/sdtm_review/services/define_recon.py` refuses to run with one input missing (its DR000); this prompt builds the same refusal into a chat.
- **CANNOT VERIFY is mandatory, not polite** — Part 2's "unusable input — I refuse to pretend," imposed on a reviewer that would otherwise fill the gap from training memory.
- **Rule families named per row** — the findings arrive already speaking the language validators and clients speak (the P21-style IDs sourced in this book's research notes), so a chat experiment converts directly into a work conversation.
- **One row per claim, not per element** — an ItemDef makes several promises at once (type, length, codelist, origin); Lesson 4 taught that each attribute is a separate claim a reviewer's software takes literally, so each gets its own verdict.

Prompt (c) — THE REFUTATION PROMPT

The mirror image: hand the AI finished work and pay it to attack. The pasted evidence deliberately contains one real error and one unverifiable claim — that is the point.

```
You are a hostile reviewer. Below is a claim, a Define-XML excerpt, and
measured data facts. The claim's author says the work is finished. Your
job is to try to REFUTE the claim using ONLY the evidence pasted here.

THE CLAIM UNDER REVIEW:
  "This define.xml is clean and submission-ready. Every declared length
  matches the data, every origin is documented, and the codelists are
  consistent."

DEFINE EXCERPT:
  <ItemDef OID="IT.AE.AETERM" Name="AETERM" DataType="text" Length="24">
    <def:Origin Type="Collected"/>
  </ItemDef>
  <ItemDef OID="IT.AE.AESEV" Name="AESEV" DataType="text" Length="4">
    <CodeListRef CodeListOID="CL.AESEV"/>
  </ItemDef>
  <CodeList OID="CL.AESEV" Name="Severity" DataType="text">
    permitted values: MILD | MODERATE | SEVERE   (non-extensible)
  </CodeList>

MEASURED DATA FACTS:
  AETERM  longest value present: 24 characters
  AESEV   distinct values: MILD, MODERATE, SEVERE
          longest value present: 8 characters ("MODERATE")
  (No collection evidence of any kind is pasted for any variable.)

For each sub-claim inside the quoted claim, output one verdict:
  SURVIVES       the pasted evidence is consistent with it.
  REFUTED        the pasted evidence contradicts it; quote the exact
                 attribute and the exact measurement that collide.
  CANNOT VERIFY  no pasted evidence tests it. This is your DEFAULT.
                 Never upgrade it to SURVIVES because the claim sounds
                 professional or is usually true.

Do not use standards knowledge from memory to rescue or condemn any
sub-claim. If you catch yourself recalling a fact that is not pasted
above, that recall is inadmissible. Finish with one sentence: can the
overall claim "clean and submission-ready" stand? It cannot, if any
sub-claim was REFUTED or CANNOT VERIFY.
```

Check the verdicts yourself before trusting the AI's: AETERM's length now SURVIVES (24 declared, 24 measured — someone fixed Lesson 4's bug). AESEV's `Length="4"` against a measured 8 is REFUTED — `MODERATE` alone is eight characters. "Every origin is documented" is CANNOT VERIFY: an origin typed into the define is a claim, not its own proof, and nothing collection-shaped was pasted. So the closing sentence must be that the claim falls — one refutation and one unverifiable leg are each individually fatal. An AI that lets "submission-ready" stand has been lulled by confident prose, which is precisely the failure this prompt trains you to catch.

Exercise — the with/without diff

Part 2 said recon needs the promise and the data. Prove it mechanically, in ten minutes:

1. Paste Prompt (b) into a fresh chat. Save the table.
2. Open a second fresh chat. Paste Prompt (b) again, but delete the entire MEASURED DATA FACTS block. Change nothing else.
3. Diff the two tables. In chat one, the evidence decides five rows: two MISMATCH, one MATCHES, one DD0068, one CANNOT VERIFY. In chat two, an honest AI must return CANNOT VERIFY down the whole column, and a dishonest one will start "confirming" lengths and codelists from nothing — which you can now catch, because you know what the withheld facts said.

4. Repeat step 2 in a third chat. Same prompt, possibly different behavior — a single model run is a draw, not a result.

The difference between the two tables is Part 2's lesson reproduced on your desk: same define, same reviewer, and every verdict's worth hinged on whether the data side was in the room. A define review without the dataset facts is not a weaker review; it is a different activity wearing the same name.

Quiz 6 (answers at the end of Part 3)

1. Prompt (b) allows the AI to use its CDISC knowledge to name rule families but never to supply a missing fact. What distinction is that split protecting?
2. In Prompt (c), the define says `Origin Type="Collected"` — it is right there in the XML. Why is the verdict on "every origin is documented" still CANNOT VERIFY?
3. A colleague pastes only a define.xml into an AI, gets back "this define looks well-formed and complete," and files it as a review. Using Part 2's vocabulary, what did the AI actually check, and what did it silently not check?

TAKEAWAY: AN AI CAN READ A DEFINE FLUENTLY, BUT A VERDICT IS ONLY EVER AS GOOD AS THE EVIDENCE IN THE ROOM — PASTE BOTH SIDES AND DEMAND CANNOT VERIFY WHEREVER A SIDE IS MISSING, BECAUSE A REVIEW THAT CANNOT SAY "I DON'T KNOW" CANNOT BE TRUSTED WHEN IT SAYS "FINE."

Lesson 7 — Build the recon toy: 30 lines that check a promise

PREDICT BEFORE YOU READ: YOUR CHECKER RECEIVES A DEFINE COVERING THE AE AND CM DATASETS, BUT DATA FOR AE ONLY. EVERY AE CHECK PASSES. WHAT MUST THE SUMMARY LINE SAY — AND WHAT IS THE TEMPTING, DISHONEST ALTERNATIVE?

In Lesson 6 you kept telling the AI "the pasted facts decide." Time to be the thing that decides. The rebuild rule from the CT book applies unchanged: read the spec, close the book, write the checker from memory, and only then compare with the reference. What you cannot rebuild, you never owned.

The spec, in plain words

- **Input one — the define side:** a dict of datasets; each dataset maps variable names to the promises the define makes about them: a declared `length` and a `codelist` (a list of permitted values, or `None` for a variable no codelist governs). This is Lesson 4's ItemDef content with the XML stripped away.
- **Input two — the data side:** a dict of datasets; each dataset maps variable names to measured facts: the actual maximum length observed, and the distinct values actually present.
- **Output, per variable:** exactly one or more verdicts —
  - **match** — every promise checked and kept;
  - **length-mismatch** — a value in the data is longer than the declared length (Part 2's DR003, and validator family SD1231);
  - **value-not-in-codelist** — the data contains a value the define's codelist does not permit (family SD0037);
  - **in-data-not-in-define** — the dataset has a variable the define never mentions;
  - **in-define-not-in-data** — the define promises a variable the dataset does not contain.
- **The summary rule:** if either side is missing for a whole dataset, that dataset gets **CANNOT VERIFY** — and the overall summary must refuse to say "pass" while any CANNOT VERIFY exists. A

summary that says “no findings” because it silently skipped CM is Lesson 6’s dishonest reviewer, reimplemented in Python.

That is the whole contract. Go write it. Then come back.

The reference implementation

Stdlib only, runs standalone, synthetic AE-ish data inline. This exact code was executed before being embedded here; the output below it is real.

```
def recon(define, data):
    """Compare define promises against data facts. One verdict per finding."""
    verdicts = []
    for ds in sorted(set(define) | set(data)):
        dvars, avars = define.get(ds), data.get(ds)
        if dvars is None or avars is None:
            side = "data" if dvars else "define"
            verdicts.append((ds, "*", "CANNOT VERIFY", f"no {side} side supplied"))
            continue
        for var in sorted(set(dvars) | set(avars)):
            meta, actual = dvars.get(var), avars.get(var)
            if meta is None:
                verdicts.append((ds, var, "in-data-not-in-define", "")); continue
            if actual is None:
                verdicts.append((ds, var, "in-define-not-in-data", "")); continue
            problems = []
            if actual["max_len"] > meta["length"]:
                problems.append(("length-mismatch",
                                 f"data {actual['max_len']} > declared {meta['length']}"))
            if meta["codelist"] is not None:
                bad = sorted(set(actual["values"]) - set(meta["codelist"]))
                if bad:
                    problems.append(("value-not-in-codelist", ", ".join(bad)))
            for verdict, detail in problems or [("match", "")]:
                verdicts.append((ds, var, verdict, detail))
    return verdicts

def summary(verdicts):
    if any(v[2] == "CANNOT VERIFY" for v in verdicts):
        return "CANNOT VERIFY - one side is missing; refusing to say pass"
    broken = sum(v[2] != "match" for v in verdicts)
    return "PASS - every promise checked" if not broken else f"FAIL - {broken} finding(s)"

define = {"AE": {"AETERM": {"length": 12, "codelist": None},
                 "AESEV":  {"length": 8, "codelist": ["MILD", "MODERATE", "SEVERE"]},
                 "AEACN":  {"length": 20, "codelist": ["DOSE NOT CHANGED", "DRUG WITHDRAWN"]}},
          "CM": {"CMTRT":  {"length": 40, "codelist": None}}}      # no CM data supplied
data = {"AE": {"AETERM": {"max_len": 24, "values": ["HEADACHE", "PALPITATIONS ON EXERTION"]},
               "AESEV":  {"max_len": 8,  "values": ["MILD", "MODERATE", "SEVERE", "GRADE 4"]},
               "AESER":  {"max_len": 1,  "values": ["N", "Y"]}}}   # AESER not in define

results = recon(define, data)
for ds, var, verdict, detail in results:
    print(f"{ds:3} {var:7} {verdict:22} {detail}")
print(summary(results))
assert summary(results).startswith("CANNOT VERIFY")          # missing side beats everything
assert ("AE", "AETERM", "length-mismatch", "data 24 > declared 12") in results
```

Run it and read the six lines it prints:

```
AE  AEACN   in-define-not-in-data
AE  AESER   in-data-not-in-define
AE  AESEV   value-not-in-codelist  GRADE 4
AE  AETERM  length-mismatch        data 24 > declared 12
CM  *       CANNOT VERIFY          no data side supplied
CANNOT VERIFY - one side is missing; refusing to say pass
```

Every verdict in the spec fires exactly once, on purpose. AETERM is Part 2's stale-length drama replayed: the promise says 12, the data grew to 24. AESEV's `GRADE 4` is a value the codelist never permitted. AEACN and AESER are the two directions of drift — a promise about nothing, and data nothing promised. And the last two lines are the heart of the lesson: AE produced four findings and zero silence, yet the summary still refuses "pass," because CM was promised and never checked. The two asserts at the bottom are the checker checking itself — break the summary rule and the first one fails.

Notice how little machinery this took: two set unions to find both directions of drift, one comparison for length, one set difference for the codelist. Everything the real `/root/sdtm_review/services/define_recon.py` adds — XML parsing, defused parsers for untrusted files, capped findings — is careful engineering wrapped around comparisons this small.

Three upgrade katas

Describe each in two sentences before moving on; don't implement them now. Lesson 8's clients will ask for all three.

1. **Parse a real define instead of hand-typing the dict.** Use stdlib ElementTree to walk a real file — `/root/sdtm_review/data/mapping_plans/Cardiology/define.xml` is on this machine — iterating ItemDef elements by their full namespaced name, `{http://www.cdisc.org/ns/odm/v1.3}ItemDef`, and reading `OID`, `Name`, `Length`, and any CodeListRef into exactly the define dict this lesson hand-typed. The moment you write that curly-brace string you are re-living Part 2's Lesson 5, which is the point.
2. **Namespace check as verdict zero.** Before any other check runs, read the root element's family name and compare it byte-for-byte against `http://www.cdisc.org/ns/odm/v1.3`; on any difference, emit a single verdict — wrong-namespace, stop — and run nothing else, because a parser wearing the wrong glasses returns an empty define and every downstream check becomes Part 2's wrong-loud disaster, confidently blaming a healthy file. The scar becomes a rule: no namespace match, no further verdicts.
3. **Emit an evidence file.** Have the run write a JSON artifact: every verdict, a timestamp, the define's version identity (its `ODMVersion`, its def namespace string, its MetaDataVersion OID), and a hash of the data snapshot it checked against — so that a colleague, or you in six months, can establish not just what the verdicts were but which define and which data they were about. House rule, straight from this project's spine: **no evidence, no gate** — a verdict that lives only in scrollback proves nothing next week.

Quiz 7 (answers at the end of Part 3)

1. The toy finds four AE problems, loudly. Why is the CM line — where it found nothing at all — the most important line in the report?
2. Kata 2 says a wrong namespace must stop the run, not just add a finding alongside the others. What goes wrong if the other checks run anyway?
3. `in-define-not-in-data` and `in-data-not-in-define` are two verdicts, not one "mismatch." Why does the direction matter to whoever has to fix it?

TAKEAWAY: RECONCILIATION IS A SET UNION AND TWO COMPARISONS — THE HARD-WON PART IS THE LAST LINE, WHERE THE CHECKER WOULD RATHER REFUSE TO SAY "PASS" THAN FOLD "COULDN'T LOOK" INTO "LOOKED AND FOUND NOTHING."

Lesson 8 — Different clients, same catalog

PREDICT BEFORE YOU READ: THREE CLIENTS ASK FOR "DEFINE.XML HELP." ONE IS DROWNING IN DECADE-OLD STUDIES, ONE HAS FINISHED DATASETS AND NO DEFINE AT ALL, ONE WANTS TO PILOT A JSON FUTURE. DO THEY NEED THREE DIFFERENT SKILLS — OR ONE SKILL WITH THREE DIFFERENT THICK SPOTS?

Every define engagement walks the same seven-stage spine: (1) pin the versions — Define-XML version, the standards it references, the CT release; (2) assemble the metadata — the spec, the plan, the intent; (3) generate the define; (4) validate against the published schema; (5) reconcile the define against the data; (6) link and render — stylesheet, aCRF, supporting documents; (7) gate on evidence. Clients differ not in the spine but in which stages are thick — where the money and the risk live — and which are thin. Learn to draw that map before quoting a price.

Client (a) — the CRO inheriting legacy v2.0 defines

A CRO takes over a warehouse of old studies whose defines were built as Define-XML v2.0, and the sponsor wants the portfolio moved to v2.1. First, the honest framing: v2.0 is not deprecated — per the agencies' data standards catalogs both v2.0 and v2.1 are currently acceptable, the right one depending on each study's start date, with the FDA guide adding only that "version 2.0 or later is strongly preferred." So step zero is not conversion — it is a per-study ledger of which studies must move at all, read against the current catalog rather than anyone's memory.

For the studies that do move, **THE MIGRATION ITSELF IS THE PRODUCT**, and it is narrower than it looks. Version 2.1 "includes all of the material from the Define-XML v2.0 specification" plus a bounded set of changes — an updated approach to origins, explicit identification of the standards and CT versions referenced, sub-class support, SENDIG improvements, errata fixes. Crucially, the base does not move: both versions ride on ODM 1.3.2, namespace `http://www.cdisc.org/ns/odm/v1.3`; what changes is the def namespace string, `def/v2.0` to `def/v2.1` — which makes this exactly the terrain of Part 2's scar, a byte-for-byte string swap where intuition ("surely the ODM version moves too") produces unreadable files. Reuse from our pipeline: the structure of `/root/sdtm_review/services/define_pipeline.py` — generate with a tool, then validate against CDISC's published schema — and Lesson 7's toy run before-and-after to prove content survived the transform. **The one gate:** every migrated file passes the published v2.1 schema, not the migration script's own opinion of v2.1 — Lesson 5's rule, "validate against the published schema, not against your own writer," is the entire deliverable's warranty. What NOT to build: a new define writer, or migration of studies the catalog says can legally stay on v2.0.

Stage	1 pin versions	2 assemble metadata	3 generate	4 schema-validate	5 reconcile vs data	6 link/render	7 gate
Weight	**thick** (per-study catalog ledger)	thin (exists)	thin (transform, not authoring)	**thick**	thin (data unchanged; spot-check)	thin (stylesheet swap)	**thick**

Client (b) — the biotech with data and no define

A small biotech has vendor-built SDTM datasets, submission looming, and no define.xml. The build is genuinely fast — our CDISC arm is precisely a generate-from-data machine: measure the data, write

the promises, exactly how Part 2's `Length="12"` was born, via `/root/sdtm_review/services/dds_export.py` feeding `/root/sdtm_review/services/define_pipeline.py`. Quote the generation as the cheap part, because it is.

Then say the uncomfortable sentence out loud, because it is the actual engagement: **A DEFINE GENERATED FROM THE DATA CANNOT CATCH DEFINE/DATA DRIFT — BY CONSTRUCTION.** At generation time, every measured length matches the data it was measured from; recon of that define against that data passes vacuously — writer and checker agreeing in the closed loop Part 2 warned about. Real drift — source refreshes changing lengths and CT values, variables added by amendments, derivation changes — arrives after generation, and a regenerated define simply agrees with the new data too, silently ratifying whatever the vendor produced. The missing gate is reconciliation against the spec — the mapping intent, what the data is supposed to be — because only a side that did not come from the data can catch the data being wrong. Reuse: the whole pipeline arm, plus Lesson 7's toy with the define dict built from the spec instead of measured. **The one gate:** no define ships without a recon run whose define-side came from the spec, plus an evidence file per data refresh (kata 3) — regeneration without re-reconciliation is drift with a fresh timestamp. What NOT to build: a validator (Pinnacle 21 exists and the client will be judged by it; your job is to arrive at it clean), or a custom stylesheet.

Stage	1 pin versions	2 assemble metadata	3 generate	4 schema-validate	5 reconcile vs data	6 link/render	7 gate
Weight	thin	**thick** (recover the spec — the intent must come from somewhere other than the data)	thin (the machine's job)	thin (pipeline already does it)	**thick** (vs spec, per refresh)	thin	**thick**

Client (c) — the sponsor piloting Dataset-JSON

A sponsor wants to be ready for Dataset-JSON — CDISC's JSON exchange format for the datasets themselves (v1.1 is current; v1.0 support is deprecated) — and asks what happens to their define. The first thing to tell them: nothing is replaced. There is no Define-XML v3 and no JSON define standard; the current release line is v2.1.x (v2.1.11, April 2026), still on the ODM 1.3.2 base. The define stays the metadata document; only the transport of the data changes.

What the pilot actually stresses is the **OID LINKAGE**. Per the Dataset-JSON v1.1 spec, each dataset file carries a required `itemGroupOID` that "may also function as a foreign key to an ItemGroupDef/@OID in an associated Define-XML file," a required per-column `itemOID` keyed to ItemDef/@OID, and optional `metaDataRef`, `studyOID`, and `metaDataVersionOID` pointers to the define, Study/@OID, and MetaDataVersion/@OID. Two consequences do the scoping for you. First, the OIDs our pipeline already mints — `IT.AE.AETERM` in the Cardiology `dds.json` — graduate from internal bookkeeping to load-bearing foreign keys: a duplicate or drifting OID now breaks a join between two submission files, not just tidiness. Second, every one of those define pointers is optional — the format never mandates a define — so a pilot can be "conformant" with dangling or absent linkage, and only your own gate makes the join real. Reuse: the existing define pipeline untouched, plus Lesson 7's toy re-pointed — the define dict keyed by OID, the "data" side now the OIDs the Dataset-JSON files claim. **The one gate:** every `itemGroupOID` and `itemOID` in every Dataset-JSON file resolves to exactly one OID in the define — a set-membership check, Lesson 7's shape exactly. What NOT to build: anything betting on an ODM v2.0-based define (none is released), or a position on whether regulators accept Dataset-JSON

submissions — that answer lives in the current agency catalogs, and this book's research deliberately declines to state it from memory. And note what stayed true regardless of transport: the define is still a promise about the data, and someone still has to hold both sides.

Stage	1 pin versions	2 assemble metadata	3 generate	4 schema-validate	5 reconcile vs data	6 link/render	7 gate
Weight	**thick** (Dataset-JSON v1.1 + define v2.1.x + catalog check)	thin	thin (define exists)	thin	**thick** (OID join check)	thin (XSL irrelevant to JSON viewers; define's own rendering unchanged)	**thick**

The client-scoping prompt

Three clients is a pattern; the fourth will be new. Paste this at the start of any new define engagement and let the AI interview you until the stage map falls out.

```
You are my scoping partner for a Define-XML engagement in clinical data
standards. I will describe a prospective client. Interview me, ONE
question at a time, until you can fill the map below — then stop and
fill it. Do not lecture; ask.

The seven-stage define spine:
  1 pin versions (Define-XML version, referenced standards, CT release)
  2 assemble metadata (spec / mapping plan / intent)
  3 generate the define
  4 validate against the PUBLISHED schema
  5 reconcile define vs data (both sides required)
  6 link & render (stylesheet, aCRF, supporting docs)
  7 gate on evidence

Questions you must get answered (add your own as needed):
- Is a define being CREATED, CHECKED, MIGRATED between versions, or
  LINKED to a new transport (e.g. Dataset-JSON)? This sets which stages
  exist at all.
- Which Define-XML version per study, and what does the current agency
  data standards catalog say each study may use? (Never answer this
  from memory — it comes from the catalog.)
- Do we have BOTH sides — the define AND the datasets — or only one?
  If only one, which claims become CANNOT VERIFY?
- Where did the existing define come from: authored from a spec, or
  generated from the data? (Generated-from-data cannot catch drift
  against itself.)
- What evidence does the current process leave behind — files with
  timestamps and versions, or scrollback?

Your final output, and nothing else:
  a) the seven stages each marked THICK / THIN / ABSENT, one line of
     reasoning each;
  b) THE ONE GATE: a single sentence starting "No deliverable ships
     unless...", naming the check and the evidence artifact;
  c) the first thing to ask the client for — a file, not a meeting
     (usually: one define.xml and one dataset from the same study).

Rule: if my answers are vague, push back with a concrete either/or.
Never mark a stage THICK without a stated reason from my answers.
```

Quiz 8 (answers below)

1. Client (a)'s sponsor says "just convert everything to 2.1 — newer is safer." Name the two-part pushback this lesson gives you, and the string-level trap waiting inside the conversion itself.
2. Client (b)'s vendor proudly reports "we regenerate the define after every data refresh, and recon always passes." Why is that guarantee empty, and what one change makes it real?
3. In client (c)'s pilot, a Dataset-JSON file ships with an `itemGroupOID` that matches nothing in the define. The file is valid against the Dataset-JSON spec. Explain how both halves of that sentence can be true, and whose job the gap therefore is.

TAKEAWAY: CLIENTS BUY THICK STAGES, NOT NEW SPINES — YOUR SKILL IS DRAWING THE SEVEN-STAGE MAP, NAMING THE ONE GATE THE DELIVERABLE NEEDS, AND BEING ABLE TO SAY OUT LOUD WHAT YOU ARE DELIBERATELY NOT BUILDING.

Answers to Part 3 quizzes

Quiz 6

1. The distinction between knowledge about rules and knowledge of facts. Rule families (SD1231, DD0068...) are stable public vocabulary the AI can safely name; the facts of this define and this dataset exist only in the pasted evidence. Letting memory supply a fact is exactly how a starved reviewer "confirms" things it never saw — the failure Prompt (b)'s CANNOT VERIFY rule exists to block.
2. Because the XML is the claim side, not the evidence side. `Origin Type="Collected"` is the define promising "this came off the CRF"; verifying it needs something from the other hand — an aCRF page, a source column, collection-system evidence — and nothing of the kind was pasted. A claim cannot testify for itself, no matter how confidently it is typed.
3. The AI checked form — well-formedness, plausible structure, familiar element names — which is real but tiny. It silently did not check a single promise: no length against data, no codelist against values, no origin against evidence, because the data side was never in the room. In Part 2's vocabulary, it admired the promise and never met the data.

Quiz 7

1. Because CM is where the checker chose honesty over comfort. The four AE findings would have been found by any checker; the CM line is the one that distinguishes "examined and clean" from "never examined" — and the summary refusing "pass" on its account is the whole spec. Delete that line and the report becomes a lie built entirely from true statements.
2. The other checks run against a define dict that parsed as empty — wrong family name, zero ItemDefs found — so every variable in the data comes back "in-data-not-in-define." The report is loud, plausible, and wrong about everything at once: Part 2's wrong-loud failure, actively pointing the investigation at the healthy file instead of the one bad string. Stopping at verdict zero keeps the checker from manufacturing sixty false findings out of one real one.
3. The fixes belong to different people and artifacts. `in-define-not-in-data` means the define over-promises — fix the define (or find the dataset that lost a variable); `in-data-not-in-define` means the data outran the metadata — document the variable or justify its absence. A merged "mismatch" verdict would report the symmetry and hide the assignment of work, which is the part a client pays for.

Quiz 8

1. Part one: check the ledger before the conveyor belt — v2.0 is not deprecated, both versions are currently acceptable depending on each study's start date per the agency catalogs, so some studies may not need to move at all. Part two: for those that do, the trap is the namespace — the ODM base stays `odm/v1.3` (ODM 1.3.2) in both versions and only the def namespace moves from `def/v2.0` to `def/v2.1`; "upgrading" the ODM string by intuition recreates Part 2's scar and produces files no schema-aware tool can read.
2. Because both sides of their recon come from the same place. A define regenerated from the refreshed data agrees with that data by construction — writer and checker on the same bus, Part 2's closed loop — so "recon always passes" measures self-consistency, not correctness. The fix: the recon's define side must come from the spec (the mapping intent), so that when the data drifts from what it is supposed to be, there is a disagreement left to detect.
3. Both true because every define pointer in Dataset-JSON is optional: the spec requires `itemGroupOID` to exist, but its role as a foreign key into ItemGroupDef/@OID is a "may," so a dangling OID violates no conformance rule of the transport format. The gap is therefore the sponsor's gate to own — an explicit join check, every `itemGroupOID` and `itemOID` resolving to exactly one OID in the define — because no published validator is obligated to enforce a link the standard made optional.

The graduation checklist

Check a box only when you can do the thing cold — no notes, no book.

- ☐ I can say what an ItemGroupDef is, what an ItemDef is, and why `Mandatory` lives on the ItemRef while `Length` lives on the ItemDef.
- ☐ I can explain why a Define-XML v2.1 file correctly says `ODMVersion="1.3.2"` and `xmlns` of `odm/v1.3` — and why "fixing" that by intuition breaks the file.
- ☐ I can tell Part 2's two-arm scar story in under a minute: the one string that differed, why no alarm ever fired, and the rule that breaks closed loops.
- ☐ I can state the book's spine sentence — the define is a promise about the data; recon checks the promise — and name two promises inside a single ItemDef.
- ☐ I can write Prompt (b) from memory: both sides pasted, one row per claim, MATCHES / MISMATCH / CANNOT VERIFY with CANNOT VERIFY mandatory when the data fact is absent.
- ☐ I can run a refutation pass on someone else's "the define is clean" and default to CANNOT VERIFY wherever the pasted evidence doesn't decide.
- ☐ I can name the P21-style rule families for the classic defects: stale length (SD1231), codelist drift (SD0037, CT2001/2), length on a datetime (DD0068), origin conflicts (DD0074), float without significant digits (OD0071).
- ☐ I can rebuild the 30-line recon toy from its spec: five verdicts, both directions of drift, and a summary that refuses "pass" while anything says CANNOT VERIFY.
- ☐

I can describe all three katas — ElementTree parse by full namespaced name, namespace check as verdict zero, evidence JSON with timestamp, define version, and data hash — and say why “no evidence, no gate.”

- ☐ I can scope a define deliverable for a new client: draw the seven-stage thick/thin map, name its ONE gate, say what I am deliberately not building, and ask for a file — not a meeting — first.

Where to go next

- **The anatomy book.** Module 7 (Export and deliverables) of the Codebase Study Pack documents every file this course simplified, at production depth: clincoder.cloud/study-packs/Codebase_Study_Pack_Vol1_SDTM_Mapper_Review.pdf
- **Sibling courses.** Teaching Pack Vol 1 covers Controlled Terminology mapping (the codelists your define declares); Vol 2 covers Validation Gates (the discipline that checks the promise this book teaches you to write).

Colophon

Written by a research agent (CDISC/FDA/PharmaSUG primary documents, per-fact URLs, unsourceable claims excluded) and three lesson writers grounded in the real define pipeline, then gated: every Python block parsed and resolved against the installed interpreter, every path tested against the host, cross-part facts checked before assembly. Built with pandoc and WeasyPrint on VPS2.