TEACHING PACK · CLINCODER LEARNER SERIES · VOLUME 1

Controlled Terminology Mapping in SDTM

The concepts first, the code second · learn by interrogating a real tool with AI · predict, ask, verify, break, rebuild. Every example is a real row of synthetic data moving through a real mapping engine, and every scar is a true story with a rule at the end. 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: how a messy raw value like `Mild` becomes the submission value `MILD` in an SDTM dataset, why that tiny step carries regulatory weight, and how to make an AI do the work without ever trusting it to be right.

Every lesson runs the same loop, borrowed from the Learner's Companion:

1. **Predict** — before reading, write three sentences guessing the answer to the lesson's opening question. Being wrong here is the point.
2. **Ask** — read the lesson, or better, paste it into Claude with the tutor prompt from Part 3 and let it teach you interactively.
3. **Verify** — run the lesson's snippet. Every snippet is standalone: no ClinCoder imports, nothing to install beyond pandas.
4. **Break** — do the "break it" exercise. Watching the check fail teaches more than watching it pass.
5. **Rebuild** — close the book and write the toy version from memory.

THE ONE RULE THAT CARRIES THE WHOLE SUBJECT: an AI may propose a mapping; only deterministic code may approve one. Every scar story in Part 2 is a place where that boundary was blurred, and every exercise in Part 3 exists to make the boundary a habit.

The tool examples come from the real ClinCoder mapping engine on VPS2 at the basis commits printed on the cover; the terminology facts come from NCI-EVS and CDISC sources cited inline. All data is synthetic.

Part 1 — The concepts

Before we touch a single line of pipeline code, you need three ideas: what controlled terminology actually is, where the official copy of it lives and why its date matters, and where it belongs inside any data tool you will ever build or audit. That is Part 1. Everything here uses one synthetic adverse-event file and facts taken directly from the published standards, so by the time Part 2 walks you into a real mapping engine, nothing in it will look mysterious.

Lesson 1 — Why `Mild` is wrong and `MILD` is right

PREDICT BEFORE YOU READ: A SITE COORDINATOR TYPES `MILD` INTO A DATA-ENTRY SCREEN TO DESCRIBE A HEADACHE. THE DATASET THAT REACHES THE REGULATOR MUST SAY `MILD`. IS THE DIFFERENCE COSMETIC — A STYLE PREFERENCE A REVIEWER WOULD SHRUG AT — OR A REAL DEFECT THAT SOFTWARE WILL FLAG? COMMIT TO AN ANSWER NOW; CHECK YOURSELF AT THE END.

The dropdown-list analogy

Imagine a paper form with a blank line labeled Country. A thousand people fill it in and you get `USA`, `U.S.`, `United States`, `America`, and one memorable `'Merica`. Five spellings, one country, and now every computer that reads the form needs a translation table just to count Americans.

Now imagine the same form as a web page where Country is a **dropdown list**: you may pick `UNITED STATES` or nothing at all. Free text is forbidden. Counting is now trivial, because the list of possible answers was decided before anyone answered.

Controlled terminology (CT) is that dropdown list, published in advance, for the columns of clinical trial data. CDISC — the standards body for clinical data — defines it as "the set of codelists and valid values used with data items within CDISC-defined datasets," supplying "the values required for submission to FDA and PMDA in CDISC-compliant datasets" (cdisc.org). Note what it does not do: CT "does not tell you WHAT to collect." The site can collect severity however its form is designed. CT standardizes how the answer is represented in the submitted electronic dataset. Codelists exist both for "questions" (which test was run) and for "answers" (what the result was).

The lists themselves are maintained and distributed by the National Cancer Institute's Enterprise Vocabulary Services (EVS) as part of the NCI Thesaurus, free of charge and "without licensing restrictions" (cancer.gov).

A codelist up close

Open the published file `SDTM Terminology.txt` (a plain text file anyone can download from NCI EVS) and every row has the same eight columns:

```
Code | Codelist Code | Codelist Extensible (Yes/No) | Codelist Name
     | CDISC Submission Value | CDISC Synonym(s) | CDISC Definition
     | NCI Preferred Term
```

Two kinds of rows live in that file. A **CODELIST** row names a whole list — for adverse event severity, that row is **C66769**, "Severity/Intensity Scale for Adverse Events," submission value `AESEV`, extensible = No. A **term** row names one allowed value inside a list. Every codelist and every term carries its own **NCI C-code** — a permanent ID in the NCI Thesaurus, like a passport number that survives any renaming.

Here is the term our headache needs, C-code **C41338**, exactly as published (evs.nci.nih.gov):

Layer	Value for this concept
CDISC Submission Value	`MILD`
CDISC Synonym(s)	`1; Grade 1`
NCI Preferred Term	`Mild Adverse Event`
CDISC Definition	"A type of adverse event that is usually transient and may require only minimal treatment or therapeutic intervention. The event does not generally interfere with usual activities of daily living."

One concept, four names — and they have sharply different jobs:

- **Submission value** — the exact string that goes into the dataset. CDISC's FAQ is blunt: "CDISC Submission Values are part of a conformant dataset and are a required part of a submission to FDA and PMDA," and "CDISC rules dictate that a single concept should only have one submission value" (cdisc.org).
- **Synonyms** — mapping aids only: "Synonyms are published to help with mapping and finding the correct submission value to use." May a synonym appear in a submitted dataset? The FAQ's whole answer is "No."
- **NCI Preferred Term** — the concept's name inside the NCI Thesaurus. Useful for humans and ontologies, never for datasets.
- **Definition** — every term has one, so two people arguing about a borderline case have a referee.

There is a fifth word you will hear constantly: **DECODE**. In a define.xml file (the machine-readable data dictionary that accompanies a submission), each coded value can be paired with a human-readable display value — the decode. The classic mistake is submitting the decode or a synonym string where the submission value belongs. Keep the definition that minimal for now; you will see decodes in the wild later.

The whole `AESEV` codelist has exactly three members: C41338 `MILD`, C41339 `MODERATE`, C41340 `SEVERE`. That is it — small, closed, memorable, which is exactly why it is the standard classroom example.

For the best proof that submission value and NCI preferred term are genuinely different names, look at the outcome codelist **C66768** ("Outcome of Event," submission value `OUT`, extensible = No). One of its members is C-code **C48275**: submission value `FATAL`, synonyms `5; FATAL; Grade 5`, NCI Preferred Term **"Death Related to Adverse Event"**, definition "The termination of life as a result of an adverse event. (NCI)". You submit the five-letter string `FATAL`; the thesaurus calls the same concept by a five-word name. Same passport number, different name on different documents.

Our running example file

This book uses one synthetic dataset — 78 made-up adverse events for made-up patients, no real person anywhere — at `/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv`. Its `AE_SEVERITY` column holds exactly the values `Mild`, `Moderate`, `Severe`: readable English, and by the letter of the standard not legal submission values, because the published strings are `MILD`, `MODERATE`, `SEVERE`. Its `AE_OUTCOME` column, by contrast, arrives already speaking CT: `RECOVERED/RESOLVED`, `NOT RECOVERED/NOT RESOLVED`, and one `FATAL`. Real studies are like this — some columns come in fluent, some need translation, and you cannot tell which is which without the published list in hand.

The SAS bridge

If you come from SAS: this is PROC FORMAT. A format maps raw values to display values through a lookup table you control. CT is the same mechanism with two twists — the format catalog is published by NCI EVS, not written by you, and the agency effectively runs it against your data on arrival. You do not get to invent entries.

Why the agency cares

This is not folklore; it is written down. The FDA's Study Data Technical Conformance Guide (June 2026 edition) says, in section 6.1.3:

> "Sponsors should use the terminologies and code lists in CDISC Controlled Terminology, which can be found at the NCI (National Cancer Institute) Enterprise Vocabulary Services. For variables where no standard terms exist, or if the available terminology is insufficient, the sponsor should propose its own terms. The sponsor should provide this information in the define.xml file and in the relevant RGs." (fda.gov)

Two honest footnotes. First, the TCG is a technical specifications document "incorporated by reference into" the FDA's binding guidance on standardized study data, and its own header says it "Contains Nonbinding Recommendations" — the binding hook is the parent guidance. Second, no fetched CDISC page states a case-sensitivity rule in so many words; what we can say is that the submission value is the required string, and the required string is `MILD` — so `Mild` simply is not it. The TCG makes the same point for the MedDRA dictionary: "The spelling and capitalization of MedDRA terms should match the way the terms are presented in the MedDRA dictionary."

So: your Predict answer. `Mild` vs `MILD` is not cosmetic. It is the difference between a string on the published list and a string off it.

Ten lines of code, four layers of names

You do not need Python fluency here — read it as a labeled filing cabinet and a question asked three times.

```
# One concept, four name-layers: C41338 in codelist AESEV (C66769)
concept = {
    "c_code": "C41338",
    "submission_value": "MILD",
    "synonyms": ["1", "Grade 1"],
    "nci_preferred_term": "Mild Adverse Event",
}

def which_layer(raw):
    if raw == concept["submission_value"]:
        return "SUBMISSION VALUE  -> goes in the dataset as-is"
    if raw in concept["synonyms"]:
        return "SYNONYM           -> maps to MILD; never submitted itself"
    if raw == concept["nci_preferred_term"]:
        return "NCI PREFERRED TERM-> thesaurus name; never submitted"
    return "NO MATCH in this concept"

for raw in ["MILD", "Mild", "Grade 1"]:
    print(f"{raw!r:>10}  {which_layer(raw)}")
```

Run it and three lines come back: `MILD` hits the submission-value layer; `Mild` hits **nothing** (which is the whole of Lesson 1 in one line of output); and `Grade 1` hits the synonym layer.

Break it on purpose

Look at that `Grade 1` line again. An oncology CRF that grades severity 1-5 will hand you `Grade 1` all day, and the lookup found it — in the synonym column. It would be easy to conclude "found means fine" and write `Grade 1` to the dataset. That is precisely the licensed use of synonyms turned inside out: **synonyms are for finding, not for submitting.** The synonym column exists so your mapping lands on `MILD`; the FAQ's one-word answer on submitting synonyms remains "No."

Quiz

1. Of the four name-layers (submission value, synonym, NCI preferred term, definition), which one is allowed to appear in the submitted dataset?
2. A CRF collects severity as `Grade 1`. Which column of the published terminology file lets you discover the right term, and what string do you actually submit?
3. The submission value `FATAL` (C48275) has NCI Preferred Term "Death Related to Adverse Event." If NCI renamed its preferred term tomorrow, what would happen to the string in your dataset, and why?

TAKEAWAY: every controlled term is one concept with one C-code and four names, and exactly one of those names — the CDISC Submission Value — is ever allowed into the dataset.

Sources: https://www.cdisc.org/standards/terminology/controlled-terminology ; https://www.cdisc.org/kb/articles/controlled-terminology-faqs ; https://www.cancer.gov/about-nci/organization/cbiit/vocabulary/cdisc ; https://evs.nci.nih.gov/ftp1/CDISC/SDTM/SDTM%20Terminology.txt ; https://api-evsrest.nci.nih.gov/api/v1/concept/ncit/C66769?include=full ; https://api-evsrest.nci.nih.gov/api/v1/concept/ncit/C41338?include=full ; https://api-evsrest.nci.nih.gov/api/v1/concept/ncit/C66768?include=full ; https://api-evsrest.nci.nih.gov/api/v1/concept/ncit/C48275?include=full ; https://www.fda.gov/media/153632/download

---

Lesson 2 — Where the truth lives: releases, versions, and skew

PREDICT BEFORE YOU READ: TWO FILES, BOTH NAMED "SDTM TERMINOLOGY," DOWNLOADED FROM THE SAME OFFICIAL SITE SIX MONTHS APART. SAME STANDARD, SAME CODELISTS. ARE THEY INTERCHANGEABLE? IF NOT, WHICH ONE IS "THE" TRUTH FOR YOUR STUDY?

A quarterly newspaper, not a stone tablet

Controlled terminology is developed by CDISC "in collaboration with the National Cancer Institute's Enterprise Vocabulary Services (EVS)," and it is released **QUARTERLY** (cdisc.org). While a release is being assembled it goes by a package number — "a capital P followed by a number"; the current release, dated 2026-03-27, includes terms from Review Package 61. On publication, "the package number is replaced by the version date." And here is the sentence to memorize: "A version date will only change when changes have been made to the Terminology set." The date is the version. Per the CDISC FAQ, that version date is what sponsors record in define.xml to say which CT vintage a submission used.

Where does it physically live? On the NCI EVS FTP site, one folder per standard (all verified live):

- SDTM (plus CDASH files): https://evs.nci.nih.gov/ftp1/CDISC/SDTM/
- ADaM: https://evs.nci.nih.gov/ftp1/CDISC/ADaM/
- SEND: https://evs.nci.nih.gov/ftp1/CDISC/SEND/
- Define-XML: https://evs.nci.nih.gov/ftp1/CDISC/Define-XML/

(plus DDF, TMF, Glossary folders, and a Protocol folder retired to Archive). Each package is published in six formats — Excel, text, odm.xml, PDF, HTML, and OWL/RDF — each release ships a **CHANGES** file, and "terminology is versioned by date and you can find all previous versions in archive subdirectories" (cancer.gov). CT is also accessible programmatically through the CDISC Library.

One subtlety that saves real confusion: a quarterly release "replaces all older ADaM, CDASH, DDF, Define-XML, SDTM, SEND terminology files." SDTM CT and ADaM CT are separate files for separate standards, but they are versioned and republished together, on the same date. One newspaper, several sections, one dateline.

If a term you need is missing, you can ask CDISC to add it — but requests should be submitted "two months before any package cut-off date" to make that quarter's paper.

Extensible vs non-extensible: may I add my own value?

Column C of the terminology file — "Codelist Extensible (Yes/No)" — answers the most common question a mapper asks. Per the CDISC FAQ, a sponsor may include a non-standard term only if the codelist is extensible: "Yes, as long as the codelist is considered extensible (i.e., the codelist row contains 'Yes' in column C)."

Concretely, in the current file: **AESEV (C66769) AND AEOUT (C66768) ARE BOTH EXTENSIBLE = NO.** You may not invent a fourth severity or a new outcome, full stop. **ROUTE (C66729, "Route of Administration Response") is extensible = Yes** — a sponsor may add a route of administration the list lacks.

But "Yes" is not a free pass. The FDA TCG, section 6.1.2:

> "the creation of custom terms for submission is discouraged. Furthermore, the use of custom or extensible code lists should not be interpreted to mean that sponsors may substitute their own nonstandard terms in place of existing equivalent standardized terms." (fda.gov)

If custom terms truly cannot be avoided, the sponsor "should clearly identify and define them within the submission, reference them in the relevant RGs [Reviewer's Guides], and use them consistently throughout the application." Pinnacle 21's team (certara.com) adds field experience: the belief that "any new terms can be added to an extensible codelist" so long as they appear in define.xml is a misconception — "Often, extensions to standard terminology are incorrect," the preferred route is a new-term request to CDISC, and "Many requested new terms are rejected for various reasons."

There is even an FDA-sanctioned example of extending a non-extensible list: the TCG describes adding histopathology severity values like `1 OF 4` to the SEND MISEV codelist when using CT dated before 2018 — and instructs sponsors to "include the extended values in define.xml and nSDRG, and explain any resulting validation error(s) in the nSDRG." The pattern to absorb: extension means document and explain, never hide.

Version skew: the silent killer

Studies are slow. A trial that starts in 2026 may submit in 2031, and CT will have shipped twenty quarterly papers in between. The TCG (6.1.2) meets this head-on:

> "FDA recognizes that studies are conducted over many years, during which time versions of terminology may change. Sponsors should use the most recent version of the dictionary available at the start of a clinical or nonclinical study... It is common to have different studies use different versions of the same dictionary within the same application."

But pooled analyses (an ISS) "should be conducted using a single version of the terminology," with version impact described in the SDSP or Reviewer's Guide. And the two big agencies split: per Pinnacle 21, FDA "does not formally require the same version of CT for all studies in the submission," while PMDA "formally requires the same version of CT, standards, and validation engine for all studies in the submission" (certara.com). Same data, two regulators, different version discipline.

Skew is not hypothetical in this book's own backyard. In the anatomy-book session we measured that our tree pins **`TWO`** CT vintages at once: the gates check against `sdtm_ct_2025-03-28.csv` while the CORE validator pins `sdtmct-2026-03-27`. Nothing has burned down — but two clocks on the wall is exactly how skew starts, and now you know to look for it.

Do stable codelists mean you can skip the change log?

Checking the actual NCI EVS change logs (both fetched 2026-08-03): the **`2026-03-27`** SDTM CT release contains **no changes to AESEV, AEOUT, or AEACN**, and neither does the archived **2025-09-26** release. (Other 2025 quarters were not retrievable at guessed archive names, so the honest claim is "stable through the releases checked," not "unchanged all year.") The core AE codelists are stable, non-extensible, and small — which is exactly why they make good classroom examples.

So why bother with change logs? Because the same 2026-03-27 log that says "nothing for AESEV" also announces four brand-new **DVDECOD** terms (Protocol Deviations Dictionary Derived Term, request CDISC-6902): "ADVERSE EVENT(S) REPORTED LATE" (C225552), "SERIOUS ADVERSE EVENT(S) REPORTED LATE" (C225573), "FOLLOW UP FOR ADVERSE EVENT(S) REPORTED LATE" (C225559), and "FOLLOW UP FOR SERIOUS ADVERSE EVENT(S) REPORTED LATE" (C225561). Your severity mapping did not move; your protocol-deviation mapping just gained four legal values it never had before. Each quarter "NCI-EVS generates an Excel file that lists changes made from the previous quarter" — new terms, changed submission values, synonyms, definitions, retired terms, plus "instructions for mapping retired terms to new or existing terms." The diff program itself is open source (github.com/NCIEVS/nci-diff-cdisc). Reading the Changes file is the habit; the AE lists being quiet this quarter is the reason the habit feels skippable, and skew stories are what happen to people who skip it.

Two vintages in twelve lines

```
# The slice of two real SDTM CT releases that our example touches.
# Facts from the NCI EVS change logs, fetched 2026-08-03 (see Sources):
# AESEV unchanged; four DVDECOD terms added in the 2026-03-27 release.
ct_2025_09_26 = {
    "AESEV": {"MILD", "MODERATE", "SEVERE"},
    "DVDECOD": set(),   # toy slice: the four 'reported late' terms not yet published
}
ct_2026_03_27 = {
    "AESEV": {"MILD", "MODERATE", "SEVERE"},
    "DVDECOD": {
        "ADVERSE EVENT(S) REPORTED LATE",
        "SERIOUS ADVERSE EVENT(S) REPORTED LATE",
        "FOLLOW UP FOR ADVERSE EVENT(S) REPORTED LATE",
        "FOLLOW UP FOR SERIOUS ADVERSE EVENT(S) REPORTED LATE",
    },
}
for codelist in sorted(ct_2025_09_26):
    added = ct_2026_03_27[codelist] - ct_2025_09_26[codelist]
    removed = ct_2025_09_26[codelist] - ct_2026_03_27[codelist]
    if not (added or removed):
        print(f"{codelist}: unchanged")
    for term in sorted(added):
        print(f"{codelist}: ADDED   {term}")
    for term in sorted(removed):
        print(f"{codelist}: REMOVED {term}")
```

Output: one `AESEV: unchanged` line and four `DVDECOD: ADDED` lines — a homemade Changes file.

Break it on purpose

Now run the skew experiment in your head. Your mapping spec was frozen against the 2025-09-26 vintage; a site reports a deviation that is, in plain English, a serious adverse event reported late. Against the old list there is no matching term — so the mapper parks it somewhere lossy. Months later, the validator runs against the 2026-03-27 vintage, where `SERIOUS ADVERSE EVENT(S) REPORTED LATE` is a perfectly legal value:

```
new_dvdecod = {
    "ADVERSE EVENT(S) REPORTED LATE",
    "SERIOUS ADVERSE EVENT(S) REPORTED LATE",
    "FOLLOW UP FOR ADVERSE EVENT(S) REPORTED LATE",
    "FOLLOW UP FOR SERIOUS ADVERSE EVENT(S) REPORTED LATE",
}
old_dvdecod = set()
value = "SERIOUS ADVERSE EVENT(S) REPORTED LATE"
for vintage, terms in [("2025-09-26", old_dvdecod), ("2026-03-27", new_dvdecod)]:
    verdict = "valid" if value in terms else "NOT IN CODELIST"
    print(f"against {vintage}: {verdict}")
```

The verdict flips on the vintage alone — the data never changed. Map against one version and validate against another, and you get either false alarms or, worse, silence about a term you should have used. The cure is boring and absolute: pin **ONE** version date end-to-end and write it in define.xml.

Quiz

1. A submission must state which CT version it used. What exactly identifies a CT version, and where is it recorded?

2. Your study collects a route of administration that is not in the ROUTE codelist, and a fourth severity grade that is not in AESEV. Which one may you add as a sponsor term, and what must you do when you add it?
3. Your application pools three studies mapped on three CT vintages. What does FDA formally require about version uniformity, what does FDA say about the pooled (ISS) analysis, and what does PMDA require?

TAKEAWAY: controlled terminology is a dated quarterly publication, not a fixed law of nature — so pin one version date end-to-end, record it in define.xml, and read the Changes file every quarter even when your favorite codelists sat still.

Sources: https://www.cdisc.org/standards/terminology/controlled-terminology ; https://www.cdisc.org/kb/articles/controlled-terminology-faqs ; https://www.cancer.gov/about-nci/organization/cbiit/vocabulary/cdisc ; https://evs.nci.nih.gov/ftp1/CDISC/SDTM/SDTM%20Terminology.txt ; https://evs.nci.nih.gov/ftp1/CDISC/SDTM/SDTM%20Terminology%20Changes.txt ; https://evs.nci.nih.gov/ftp1/CDISC/SDTM/Archive/SDTM%20Terminology%20Changes%202025-09-26.txt ; https://api-evsrest.nci.nih.gov/api/v1/concept/ncit/C66729?include=full ; https://github.com/NCIEVS/nci-diff-cdisc ; https://www.fda.gov/media/153632/download ; https://www.certara.com/p21/blog/metrics-cdisc-terminology ; CT vintage pins in our own tree: measured in the anatomy book session.

Lesson 3 — One spine, seven stages: where CT sits in any tool

PREDICT BEFORE YOU READ: A DATA TOOL NEEDS THE CODELIST SOMEWHERE. SHOULD IT LIVE AT THE POINT WHERE VALUES GET MAPPED, THE POINT WHERE THE OUTPUT GETS CHECKED, OR BOTH? ONE OF THOSE THREE ANSWERS IS A TRAP.

The seven-stage spine

Every serious data tool — whether it is three scripts or a fleet of services — walks the same spine. Learn it once and you can find your way around any of them.

1. **Ingest.** Raw files arrive from the outside world and are copied in, untouched. Nothing is interpreted yet; the only job is to receive faithfully.
2. **Inventory.** The tool takes stock: which files, which columns, which values actually appear. You cannot map what you have not counted.
3. **Knowledge.** The reference material is loaded — for us, the controlled terminology file with its codelists, C-codes, and submission values. This stage holds what is true, independent of any one study.
4. **Propose.** Something drafts the mapping — a rule, a script, or an AI — saying "raw `Mild` should become `MILD` in `AESEV`." A proposal is an opinion, not a fact.
5. **Verify.** Deterministic checks re-derive whether each proposal is actually legal — is that string really in that codelist, in the pinned version? No judgment calls, just lookups that pass or fail.
6. **Review.** A human sees what the checks could not settle — the borderline, the ambiguous, the genuinely new. People spend attention only where machines ran out of certainty.
7. **Deliver + evidence.** The outputs ship together with the proof: what ran, against which references, with which results. If there is no evidence, as far as anyone downstream is concerned, it did not happen.

Where CT sits — and why it appears twice

Controlled terminology is **STAGE 3, KNOWLEDGE**. But knowledge that sits in a folder helps no one. It must be **fed forward into stage 4** — if an AI drafts your mappings, the legal values for `AESEV` belong in its prompt, so the proposer knows the dropdown list before it answers. And it must be **re-checked at stage 5** — the verifier independently looks up every proposed value against the same pinned CT file, trusting nothing the proposer said about itself.

That double appearance is not redundancy; it is the whole design. The proposer with CT in hand makes fewer mistakes; the verifier with CT in hand catches the ones it makes anyway. Remove either copy and you are trusting a single point of failure. Part 2 tells the story of what happened in our own engine when CT appeared in neither place — worth reading precisely because the spine makes the failure feel predictable in hindsight.

Four pitfalls you will meet

You now know enough to recognize the classic CT failures on sight. Consider these previews; each returns with real machinery in Part 2.

SYNONYM SUBSTITUTION. The published synonym columns are so convenient for mapping that people submit them. CDISC's FAQ says synonyms may not be submitted ("No"), and Pinnacle 21 documents the flavor that stings most — units: "using the synonym 'mg/g' for the standard term 'g/kg' in a regulatory submission is an error" (certara.com). Two strings a human skims past; two very different quantities to anyone doing arithmetic.

FREE TEXT MEETING A CLOSED LIST. CRFs speak human. P21's example: the non-extensible AEACN codelist (what was done about the study drug) receiving the collected value `Restarted drug`. The fix is mapping to the standard term `DRUG INTERRUPTED` — and if genuinely no standard term fits, submit the collected term with minimal formatting, because "science prevails over standards." The trap is inventing a near-miss term when a real one exists.

RAW CODING AND UNIT MISMATCHES. Upstream systems encode things their own way: an EDC exporting Sex as `0`/`1` that must become `F`/`M`; lab units collected as ng/mL that standardize to ug/L; catch-all values like `OTHER`, `MULTIPLE`, `LBALL` that are "formally not a part of standard CDISC CT" even though they are sometimes seen in practice (certara.com). None of these are typos — they are two systems speaking different dialects, and only a lookup against the published list exposes it.

`OTHER` ABUSE. When mapping gets hard, `OTHER` is always available, and that is the problem. The FDA TCG, section 6.1.2: "it is not recommended to map a collected value to 'OTHER' when a controlled term is available that matches the collected value – even if the terminology allows sponsor expansion" (fda.gov). `OTHER` is where information goes to disappear.

And when these slip through anyway, validators catch them: Pinnacle 21 rule **CT2002** fires when a value is "not found in" its codelist — community threads show it firing on both non-extensible lists (RACE in DM) and extensible ones (EPOCH). Today CT2002 carries "a P21 Message Type of Warning," with plans to reclassify invalid extensions as Errors; and FDA's technical rejection enforcement of study-data Reject issues began "2021-09-15 for most studies started after 2016-12-17" (pinnacle21.com; certara.com). The net is real.

Exercise: fill in the spine

No code this lesson. Copy the diagram and fill each blank with the stage name and one phrase for what CT is doing there (three of the seven stages should mention CT; answers at the end of Part 1).

```
raw files
   |
   v
[1] INGEST     - receive raw files untouched; CT role: ______
   |
[2] _______    - count files, columns, values;  CT role: none yet
   |
[3] _______    - load reference material;       CT role: ______
   |
[4] _______    - draft the mapping;             CT role: ______
   |
[5] _______    - deterministic legality checks; CT role: ______
   |
[6] _______    - humans handle the borderline
   |
[7] _______    - ship outputs with proof of what ran
```

Quiz

1. At which numbered stage does CT live, and at which two other stages must it appear for the design to be safe?
2. A CRF hands you `Restarted drug` for a variable governed by the non-extensible AEACN codelist. Rank these responses from best to worst: (a) map to the standard term `DRUG INTERRUPTED`; (b) map to `OTHER`; (c) add `RESTARTED DRUG` as a sponsor extension.
3. Why is submitting the synonym `mg/g` instead of the standard `g/kg` worse than a cosmetic error?

TAKEAWAY: CT is stage-3 knowledge that must travel — forward into the proposer's hands and again into the verifier's — because a codelist that only sits in a folder protects no one.

Sources: https://www.certara.com/p21/blog/metrics-cdisc-terminology ; https://www.cdisc.org/kb/articles/controlled-terminology-faqs ; https://www.fda.gov/media/153632/download ; https://www.pinnacle21.com/forum/dm-domain-error-race-value-not-found-race-non-extensible-codelist-warning-versus-error ; https://www.pinnacle21.com/forum/ct2002-epoch-value-not-found-epoch-extensible-codelist

---

Answers to Part 1 quizzes

Lesson 1

1. **The submission value, and only the submission value.** Synonyms are published "to help with mapping and finding the correct submission value"; the FAQ's answer on whether they may be submitted is "No." The NCI preferred term is the thesaurus's name for the concept; the definition is a referee for borderline cases. None of the three go in the dataset.
2. **The CDISC Synonym(s) column finds it; you submit `MILD`.** `Grade 1` is listed as a synonym of concept C41338, whose one-and-only submission value is `MILD`. Synonyms are for finding, not submitting.
3. **Nothing happens to your dataset string.** The submission value `FATAL` and the NCI Preferred Term "Death Related to Adverse Event" are different name-layers of the same concept, C48275. The dataset carries the submission value; the thesaurus name can differ from it (and already does) without touching what you submit.

Lesson 2

1. **The version date identifies the release** — "a version date will only change when changes have been made to the Terminology set" — and per the CDISC FAQ, that date is **recorded in define.xml**.
2. **The route may be added; the severity may not.** ROUTE (C66729) is extensible = Yes; AESEV (C66769) is extensible = No. And an added route is not a free pass: per the FDA TCG, clearly identify and define it in the submission, reference it in the Reviewer's Guides, and use it consistently — extension means document and explain, never hide.
3. **FDA** "does not formally require the same version of CT for all studies in the submission," but says the pooled ISS analysis "should be conducted using a single version of the terminology," with version impact described in the SDSP or Reviewer's Guide. **PMDA** "formally requires the same version of CT, standards, and validation engine for all studies in the submission."

Lesson 3

1. CT lives at **stage 3 (Knowledge)** and must also appear at **stage 4 (Propose** — fed into the proposer's prompt or rules**)** and **stage 5 (Verify** — independently re-checked**)**. The double appearance is the design.
2. **Best: (a)** — a matching standard term exists, so use it; mapping to `DRUG INTERRUPTED` is P21's own recommended fix. **Then (c), which at least fails loudly** — AEACN is non-extensible, so the sponsor extension is simply not allowed and a validator will flag it. **Worst: (b)** — it looks legal, yet the TCG explicitly recommends against mapping to `OTHER` when a matching controlled term is available; it destroys information silently, which is worse than being caught.
3. Because `mg/g` and `g/kg` are **different quantities, not different spellings** — a factor-of-a-thousand difference in meaning. Certara/P21 cite exactly this substitution as a real submission error: a synonym slipped into the dataset where the standard term `g/kg` belonged, which corrupts any downstream arithmetic, not just the formatting.

Spine diagram (filled)

```
[1] INGEST     - CT role: none (receive faithfully)
[2] INVENTORY  - CT role: none yet
[3] KNOWLEDGE  - CT role: the codelists are loaded here; this is where CT lives
[4] PROPOSE    - CT role: legal values fed to the proposer before it drafts
[5] VERIFY     - CT role: every proposed value re-checked against the pinned list
[6] REVIEW     - humans handle the borderline
[7] DELIVER+EVIDENCE - ship outputs with proof of what ran
```

Part 2 — Inside a real mapping engine

In Part 1 you learned what a codelist is: a short, published list of the only values a variable is allowed to hold. Now we go somewhere most tutorials never take you — inside a real, working mapping tool that runs on this machine, following one real (synthetic) data value all the way through it.

And then, because this is a teaching book and not a brochure, we will look at the day the tool got it wrong. Not a hypothetical. A real design mistake, still visible in the code, that quietly produced wrong answers and passed its own checks while doing it. The scar it left behind reshaped the whole system, and the lesson it bought is the single most important idea in this book.

Our running example stays the same as always: one synthetic adverse-event file, `/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv`. Seventy-eight rows of made-up patients with made-up headaches and rashes. No real person appears anywhere in this book.

Lesson 4 — Watching `Mild` become `MILD`

PREDICT BEFORE YOU READ: A RAW FILE SAYS A HEADACHE WAS `MILD`, AND THE CDISC STANDARD SAYS THE LEGAL VALUE IS `MILD`. IS THAT AN ERROR, A NON-ERROR, OR A THIRD THING?

Hold your answer. Let's follow the data and find out what the tool says.

One row walks into an engine

Open `/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv` and look at the second data row. Subject `101001`, adverse event number 2: an upper respiratory infection that started on 09-MAY-2024 and resolved. The column we care about is `AE_SEVERITY`, and for this row it says:

```
Mild
```

That value was typed by a (synthetic) site coordinator into a data-entry screen. It is perfectly readable English. It is also, by the letter of the standard, not a legal SDTM value.

Here is why. The SDTM variable this column feeds is `AESEV`, and `AESEV` is governed by a published codelist — CDISC codelist `AESEV`, NCI code C66769, which contains exactly three terms:

```
MILD    MODERATE    SEVERE
```

Uppercase. Always. That's not a style choice by our tool; those three uppercase strings are the submission values CDISC publishes, the exact bytes a regulatory reviewer's software expects to find in the dataset. `Mild` is not on the list. `MILD` is.

Step 1 — the lookup: where does the tool get the list?

The review app has exactly one module whose job is to open the official terminology file and answer "which values does codelist X contain?" It's small — 63 lines — and it lives at `/root/sdtm_review/services/ct.py`. It reads a 13 MB CSV export of the CDISC controlled-terminology package sitting at `/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv` — 1,158 codelists' worth, measured in the anatomy book session — and serves them up as a simple dictionary: codelist name in, list of legal terms out.

Ask it about our variable and you get back exactly what you'd hope:

```
terms_for("AESEV", "AESEV")  →  ['MILD', 'MODERATE', 'SEVERE']
```

For SAS programmers: this file is the raw material you would feed to `PROC FORMAT` with `CNTLIN=` to build a format catalog. A codelist is a format's domain. But keep one inversion in mind — in reporting you apply formats code → label so humans can read the output; in SDTM you submit the code side. That inversion will matter enormously in Lesson 5.

Step 2 — the comparison: three verdicts, not two

Now the tool holds two things side by side: the observed value `Mild` and the legal list `['MILD', 'MODERATE', 'SEVERE']`. A naive checker would render a binary verdict — in the list or not — and call `Mild` a failure, full stop.

The real checker, in `/root/sdtm_review/services/ta_ct.py`, is smarter, because its authors learned that not all failures cost the same. Here is the actual decision, quoted from the code (trimmed to the heart of it):

```
#| skip-check
# from ta_ct.py check_values(): a case-only miss is a different, cheaper problem
if value.lower() in lower:
    entry["expected"] = lower[value.lower()]
    base["near_misses"].append(entry)
else:
    base["violations"].append(entry)
```

Read it slowly, because this tiny `if` encodes a whole philosophy of triage:

- **Exact match.** `MILD` is in the list. Conformant. Nothing to do.
- **Near-miss.** `Mild` is not in the list, but lowercase it and it matches a legal term lowercased. The value is right in meaning, wrong only in case. The fix is mechanical — uppercase it — and the checker even hands you the answer: `"expected": "MILD"`. No human judgment required.
- **Violation.** The value doesn't match anything, even ignoring case. Something like `Slightly sore` in a severity column. No mechanical fix exists; a human has to decide what the site meant. This is a mapping problem, and it's expensive.

So the answer to the prediction question is: **A THIRD THING.** `Mild` is neither fine nor a true error — it's a near-miss, the cheapest possible defect, and separating it from real violations is the difference between a review queue a human can clear in a minute and one that buries the real problems in case noise. The checker's own docstring records that case defects were the largest single source of value disagreements between the project's R and Python pipelines — which is exactly why they get their own bucket.

Run over all 78 `AE_SEVERITY` values in our file, the verdict (measured in the anatomy book session, and reproducible with the snippet below) is: **0 conformant, 0 violations, 3 distinct near-misses** — `Mild` on 40 rows, `Severe` on 22, `Moderate` on 16. The entire column is one uppercase away from perfect.

Step 3 — the honest fourth state: unchecked

Here is the part beginners skip and professionals live by.

The v4 knowledge pack this tool runs on describes 693 SDTM variables across 21 domains. How many of those 693 can be value-checked against a published codelist the way we just checked `AESEV`?

204. (Measured in the anatomy book session; the split was 204 with checkable terms, 66 date/time formats, 13 external dictionaries, 4 multi-codelist, 2 sponsor-defined, 404 with no terminology at all.)

That means for roughly seven out of every ten variables, this checker cannot render a verdict. Dates are checked as formats, not lists. `AEDECOD` — the coded medical term — belongs to MedDRA, a licensed

external dictionary the box doesn't hold, so no code here can verify it. Hundreds of variables simply have no controlled terminology.

The tool's response to this is the most important sentence in its source. From the docstring of `/root/sdtm_review/services/ct.py`, quoting the real file: names that resolve to no codelist are

> simply not CT-checkable — callers must treat that as "unchecked", never as "passed".

UNCHECKED ≠ PASSED. A green checkmark next to a variable nobody could examine is a lie with good typography. This tool instead labels every unexamined variable with why it wasn't examined — and in Lesson 5 you'll see the disaster that happens when a system quietly does the opposite.

Try it yourself — a fifteen-line severity checker

This is the whole Lesson 4 pipeline in miniature: real file, real codelist values, three verdicts. It runs standalone with nothing but pandas.

```
import pandas as pd

CODELIST = ["MILD", "MODERATE", "SEVERE"]        # codelist AESEV (C66769)
by_lower = {t.lower(): t for t in CODELIST}

raw = pd.read_csv("/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv", dtype=str)
values = raw["AE_SEVERITY"].dropna().str.strip()

for value, n in values.value_counts().items():
    if value in CODELIST:
        verdict = "exact"
    elif value.lower() in by_lower:
        verdict = f"near-miss -> submit {by_lower[value.lower()]!r}"
    else:
        verdict = "VIOLATION (no codelist term matches)"
    print(f"{value!r:12} x{n:3}  {verdict}")
```

Output on this box:

```
'Mild'       x 40  near-miss -> submit 'MILD'
'Severe'     x 22  near-miss -> submit 'SEVERE'
'Moderate'   x 16  near-miss -> submit 'MODERATE'
```

Break it — meet a real violation

Near-misses are forgiving. Violations are not. Watch the boundary between them by feeding the same logic one misspelled value — `Sever`, a plausible data-entry slip:

```
CODELIST = ["MILD", "MODERATE", "SEVERE"]
by_lower = {t.lower(): t for t in CODELIST}

for value in ["Mild", "Severe", "Sever"]:
    if value in CODELIST:
        print(f"{value!r}: exact")
    elif value.lower() in by_lower:
        print(f"{value!r}: near-miss -> {by_lower[value.lower()]}")
    else:
        print(f"{value!r}: VIOLATION")
```

`Severe` sails through as a near-miss. `Sever` — one letter shorter — lands as a violation, because `"sever"` is simply not a key in `by_lower`. Case-folding is the only forgiveness this checker extends; it

does no spell-correction, no fuzzy matching, no "did you mean". That's a deliberate limit (the anatomy book flags it and suggests `difflib.get_close_matches` as the upgrade), and it's the right default for a first pass: a checker that guesses what you meant is a checker that will eventually guess wrong, silently.

Now un-break it: change `Sever` back to `Severe` and watch it drop back into the cheap bucket. That one-letter boundary between "mechanical fix" and "human decision" is the entire economics of data cleaning.

Quiz 4 (answers at the end of Part 2)

1. `AE_SEVERITY` contains `Moderate` and the codelist contains `MODERATE`. Which of the three verdicts does the checker return, and what extra piece of information does it attach?
2. Only 204 of the 693 variables in the v4 knowledge pack are CT-checkable. If a validation report shows no findings for `AEDECOD`, what are the two very different things that could mean — and which does this tool's "unchecked ≠ passed" rule force it to say?
3. A colleague proposes simplifying the checker to two verdicts: pass and fail. Name one concrete cost of merging near-misses into violations, using our 78-row file as the example.

TAKEAWAY: A GOOD CHECKER DOESN'T JUST SAY YES OR NO — IT SAYS EXACTLY, ALMOST (HERE'S THE FIX), NO (GET A HUMAN), OR I COULDN'T LOOK — AND IT NEVER DRESSES UP "COULDN'T LOOK" AS "YES."

Lesson 5 — The scar: the codelist that never reached the AI

Every well-designed system carries scars — places where the code got more careful because something specific went wrong. This lesson tells the story of this project's most instructive one. It's not embarrassing history; it's the reason half the machinery in Lesson 4 exists. And unlike most war stories, you can still see the wound in the source code, today, on this machine.

The setup: an engine that asks an AI to write the mapping

An earlier tool in this project, the SDTM Mapper, does something that sounds like magic: you upload a raw clinical file, and a language model writes the mapping specification — which raw column feeds which SDTM variable, and how. The function that does it is `generate_mapping_spec()` in `/root/sdtm_mapper/services/engine.py`, and its heart is a prompt: a block of text describing the source columns and the target SDTM variables, sent to the model with the instruction "produce a mapping row for every target variable."

For each target variable, the prompt builder formats one line. Here it is, quoted exactly from the file (lines 90–92):

```
#| skip-check
target_vars = "\n".join(
    f"  {v['name']} | {v['label']} | {v['type']} | core={v['core']}" + (f" | CT={v['ct']}" if
v['ct'] else "")
    for v in info["variables"])
```

Look at the tail: `CT={v['ct']}`. Whatever string the knowledge pack stored in that variable's `ct` field goes into the prompt. So for the adverse-event outcome variable, the model saw this line:

```
AEOUT | Outcome of Adverse Event | Char | core=Perm | CT=OUT
```

`CT=OUT`. Three letters. That is the name of the codelist — like telling someone "the password is stored in the password file" instead of telling them the password. The codelist called `OUT` actually contains six precise submission values (`FATAL`, `NOT RECOVERED/NOT RESOLVED`, `RECOVERED/RESOLVED`, `RECOVERED/RESOLVED WITH SEQUELAE`, `RECOVERING/RESOLVING`, `UNKNOWN`), but not one of them appears in the prompt. The model was told a list exists. It was never shown what's on it.

There's a bitter little accident buried in the knowledge pack that makes this vivid. The pack (`/root/sdtm_mapper/knowledge/sdtm_ig34.json`) was hand-typed, and the person typing it was inconsistent in a lucky-unlucky way: for `AESEV` they typed out the actual values — `"ct": "MILD/MODERATE/SEVERE"` — but for `AEOUT` they typed `"ct": "OUT"` and for `AEACN`, `"ct": "ACN"`. Just names. So severity got its terminology into the prompt by transcription accident, while outcome and action-taken never did. Same engine, same prompt template, wildly different odds of a correct answer — decided by a typist's whim nobody noticed.

What a model does when it's never shown the list

Here is the part worth slowing down for, because it teaches you something true about language models in general.

Asked to produce values for a variable whose legal list it has never seen, a model doesn't refuse. It doesn't say "I'd need the codelist." It answers — fluently, plausibly, confidently — with what it absorbed during training. And what models absorb about medical terminology is overwhelmingly the human-readable side: decodes, preferred terms, the words doctors write. Remember the format inversion from Lesson 4? In SDTM you submit the code side, but the world's text is written in the label side. A model trained on the world's text is format-forward by nature.

The project's own source code preserves the specimens. The docstring of `/root/sdtm_review/services/value_domains.py` records that a generated program emitted `CMDOSFRQ = "AS NEEDED"` — and here's the twist that makes it perfect: `AS NEEDED` isn't a random guess. The correct submission value is `PRN`, and the published CDISC decode of `PRN` (NCI code C64499) is literally "As needed" — verified in the anatomy book session against the CT package. The model gave the right answer to the wrong question. It produced the label where the code belonged, exactly like a SAS programmer applying a format in the wrong direction.

The same docstring records a second specimen: `EGTESTCD` — ECG test codes — emitted as `HR, PR, QRS, QT, QTC, RHYTHM`. Six abbreviations any cardiologist would nod at. The published submission values are `EGHRMN, PRAG, QRSAG, QTAG, RHYNOS, INTP`. Not one matched. And per the repo's docstrings, this wasn't one weak model having a bad day — it's the same mechanism, the same class of error, wherever the codelist stayed out of the prompt: readable terms where submission values belong. Which follows logically: no model, however strong, can conform to a list it was never shown. Feeding a better model the same starved prompt buys you more fluent wrongness.

Why no alarm went off

Now the second half of the wound: the safety net had a hole exactly the same shape as the failure.

The mapper has a validator (`/root/sdtm_mapper/services/validator.py`) that does check CT values — but only, as its `_ct_set()` function says in its own docstring, "ONLY when we are certain the ct note lists literal values (not a codelist name, dictionary, or format marker). Conservative by design." Hand `_ct_set` a codelist name like `OUT` or `ACN` and it returns `None` — meaning "I can't check this," and the value check is skipped.

Each half of that design is defensible. Putting the codelist name in the prompt is compact. Declining to value-check a bare name avoids false alarms. But composed, they built a corridor where wrong

values walked from the model to the output with no checkpoint anywhere: the prompt never showed the list, and the validator, seeing only a name, never opened the list either. As the anatomy book puts it, a pipeline can pass its own checks with every coded value wrong — because supplying the codelist's name is not supplying the codelist. And the checks it passed were real checks: columns present, types right, required variables populated. Structure perfect. Values fiction. The tool looked finished — was demoed as finished — before anyone executed the outputs against the actual terminology and counted.

The rule the scar bought

Out of the discovery came the principle this entire project now runs on, and the one sentence I most want you to leave this book with:

AI PROPOSES; DETERMINISTIC CODE DISPOSES.

Concretely, it means the codelist must appear at both ends of the model:

1. **BEFORE — THE VALUES GO IN THE PROMPT.** The fix lives in `/root/sdtm_review/services/value_domains.py`, whose `prompt_block()` puts the observed source values and the legal submission values in front of the model, labelled as facts. Real output from this box, for our running file (measured in the anatomy book session):

   ```
   * AESEV
       SOURCE AE_SEVERITY contains EXACTLY these 3 value(s): "Mild", "Severe", "Moderate"
       Your mapping MUST handle every one of them. A value you do not handle becomes a blank cell.
       TARGET must be a CDISC submission value from AESEV: MILD, MODERATE, SEVERE
   ```

   Nothing left to guess. `Mild → MILD` is now a reading-comprehension task, not a memory test.

2. **AFTER — THE VALUES ARE CHECKED AGAINST THE ANSWER.** The Lesson 4 machinery — `ta_ct.py`'s six-kind classification (checkable terms, ISO 8601 formats, external dictionaries, multi-codelist, sponsor-defined, none), the exact/near-miss/violation triage, and the iron rule that unchecked ≠ passed — runs over what actually got produced. And the value_domains docstring is blunt about which end matters more: the generator isn't deterministic even at temperature 0, so a better prompt only improves the odds of a draw, while a check after the fact rejects a bad one. Hope upstream, proof downstream.

The AI is still in the loop — it's genuinely good at proposing that a column called `AE_SEVERITY` feeds `AESEV`. But every value it proposes now passes through code that opened the published file and compared, byte by byte. The model gets a vote. The codelist gets a veto.

Break it — run the scar yourself

You can reproduce this failure, and its fix, in any chat with an AI assistant (Claude, or anything else). Paste this first — it is the starved prompt, the 2-line re-enactment of `CT=OUT`:

> You are mapping raw clinical data to SDTM. The raw column AE_OUTCOME must map to the variable AEOUT, which uses controlled terminology codelist OUT. List the values you would populate AEOUT with.

Read the answer carefully. You will likely get plausible outcome words — perhaps `RECOVERED`, `RESOLVED`, `ONGOING`, `FATAL`, `UNKNOWN` — delivered with total confidence, and (except by luck) not matching the published list. Now paste the fed version:

> Same task, but here is codelist OUT in full. The ONLY legal values are: FATAL; NOT RECOVERED/NOT RESOLVED; RECOVERED/RESOLVED; RECOVERED/RESOLVED WITH SEQUELAE; RECOVERING/RESOLVING; UNKNOWN. The raw column contains: "RECOVERED/RESOLVED", "NOT RECOVERED/NOT RESOLVED", "FATAL". Map each raw value to its legal AEOUT value.

Night and day. Same model, same intelligence — the only variable you changed is whether the list was in the room. And notice what you still shouldn't do with the second answer: trust it unchecked. You changed the odds; only a comparison proves the draw.

Quiz 5 (answers below)

1. The prompt line said `CT=OUT`. In one sentence, what is the difference between what the model received and what it needed?
2. Why did the mapper's validator not catch values like `AS NEEDED` — was it broken, lazy, or something more interesting?
3. State the "AI proposes, deterministic code disposes" rule as two concrete requirements about where the codelist must appear.

TAKEAWAY: A LANGUAGE MODEL ANSWERS FROM WHAT IT WAS SHOWN PLUS WHAT IT REMEMBERS — SO PUT THE CODELIST IN THE PROMPT TO IMPROVE THE PROPOSAL, AND CHECK THE CODELIST AFTER THE ANSWER BECAUSE ONLY THE CHECK IS PROOF.

Answers to Part 2 quizzes

Quiz 4

1. **Near-miss.** `Moderate` isn't in the codelist, but lowercased it matches `MODERATE` lowercased. The checker attaches the fix itself: `"expected": "MODERATE"` — the exact submission value to write.
2. It could mean "checked and clean" or "never examined." `AEDECOD` belongs to MedDRA, an external licensed dictionary this box doesn't hold, so it is one of the variables the tool cannot check — and the rule forces the report to say **unchecked**, with the reason, rather than showing a silent pass.
3. Our file has 78 severity values, every one a case-only near-miss. A pass/fail checker reports 78 failures indistinguishable from real mapping problems; the reviewer must inspect them one by one instead of applying one mechanical uppercase fix — and any genuine violation hiding among them (a `Sever`, say) is buried in the noise.

Quiz 5

1. The model received the codelist's **name** (a label pointing at a list) when it needed the codelist's **values** (the six legal strings themselves) — a pointer instead of the thing.
2. Something more interesting: it was conservative by design. Its `_ct_set()` only value-checks when the pack's `ct` note spells out literal values, and deliberately returns `None` — "can't check" — for a bare codelist name, to avoid false alarms. Sensible alone; combined with a prompt that also only carried the name, it left the exact corridor the wrong values walked through, unchecked at both ends.
3. 1. The codelist's actual values must be **in the prompt**, stated as facts, before the model answers. (b) The model's answer must be **checked against the codelist by deterministic code** afterwards — and anything that couldn't be checked must be reported as unchecked, never as passed.

Part 3 — Practice: AI as your intern, code as your inspector

You now know what a submission value is (Part 1) and you have watched a real engine map one — and watched an earlier engine get it wrong because the codelist never reached the model (Part 2, Lesson 5). Part 3 turns that knowledge into ability: first you learn to work with an AI without ever trusting it, then you rebuild the deterministic inspector from memory, and finally you learn to reshape the same skeleton for three very different clients.

Lesson 6 — Testing with AI: the interrogation method

PREDICT BEFORE YOU READ: YOU PASTE A MAPPING QUESTION INTO AN AI CHAT AND GET BACK A CONFIDENT, WELL-FORMATTED ANSWER. WHAT IS THE ONE THING THAT ANSWER CAN NEVER BE, NO MATTER HOW GOOD THE MODEL IS?

Hold your answer. Here is the working relationship this lesson installs.

The AI is your intern. A brilliant, tireless, wildly well-read intern who proposes mappings faster than you can type — and who, as Lesson 5 proved on this very machine, will answer fluently from memory whether or not it was ever shown the list. You are the senior reviewer, and the house rule is absolute: **YOU ARE NEVER ALLOWED TO ACCEPT THE INTERN'S PROPOSAL WITHOUT THE INSPECTOR** — a deterministic check against the published codelist. The intern proposes; the inspector disposes. Everything in this lesson is just that rule turned into three reusable prompts.

Each prompt below is complete: paste it into any chat AI exactly as written. No placeholders to fill in — the data and the codelists are already inside.

Prompt (a) — THE TUTOR PROMPT

Use this when you want the AI to teach you rather than work for you.

```
You are a patient clinical-data tutor. Teach me ONE lesson and nothing more:
how a raw adverse-event value becomes a CDISC controlled-terminology
submission value.

Anchor everything to these real-shaped synthetic AE rows (columns:
SUBJECT, AE_SEVERITY, AE_OUTCOME):

  101001 | Mild     | RECOVERED/RESOLVED
  102002 | Severe   | RECOVERED/RESOLVED
  103003 | Moderate | NOT RECOVERED/NOT RESOLVED
  104004 | Severe   | FATAL

Ground facts you must treat as given (do not substitute your own memory):
- Codelist AESEV (NCI C66769, non-extensible) submission values:
  MILD, MODERATE, SEVERE. The synonym "Grade 1" maps to MILD.
- Codelist for AEOUT (NCI C66768, non-extensible) submission values:
  FATAL; NOT RECOVERED/NOT RESOLVED; RECOVERED/RESOLVED;
  RECOVERED/RESOLVED WITH SEQUELAE; RECOVERING/RESOLVING; UNKNOWN.
- Synonyms are mapping aids only; they may never be submitted.

Structure of the session:
1. Explain the lesson in plain words using the rows above, no others.
2. Quiz me with 3 questions, ONE at a time. Wait for my answer each time.
3. If my answer is vague ("it gets standardized", "you clean it up"),
   do not accept it. Ask me to commit to the exact string that would
   land in the submitted dataset, then tell me if I am right.
4. Finish with one hands-on exercise I can do with these rows and a
   pencil, and its worked answer.
```

Why it is built this way: the ground-facts block exists because of Lesson 5 — a tutor answering from memory is the same starved prompt that produced `AS NEEDED`; here the codelist is in the room. The "drill vague answers" rule exists because submission values are exact strings, and "it gets standardized" is the student-side version of a green checkmark on an unchecked column.

Prompt (b) — THE MAPPING PROMPT, done right

This is the intern's work order. Read the annotations after the block — every design choice is a lesson from Parts 1 and 2 wearing work clothes.

```
You are proposing (not deciding) a controlled-terminology mapping for an
SDTM AE domain. Work ONLY from the codelists pasted below. If a raw value
does not match any submission value or listed synonym, you must output the
verdict NOT IN CODELIST — do not invent, approximate, or pick the nearest
term.

CODELIST AESEV (NCI C66769, non-extensible). Submission values:
  MILD | MODERATE | SEVERE
Published synonyms: "1" and "Grade 1" -> MILD; "Grade 2" -> MODERATE;
"Grade 3" -> SEVERE.

CODELIST OUT for AEOUT (NCI C66768, non-extensible). Submission values:
  FATAL
  NOT RECOVERED/NOT RESOLVED
  RECOVERED/RESOLVED
  RECOVERED/RESOLVED WITH SEQUELAE
  RECOVERING/RESOLVING
  UNKNOWN
Published synonyms: "U", "UNK" -> UNKNOWN.

RAW VALUES to map — handle every one; a value you skip becomes a blank
cell in the dataset:
  AE_SEVERITY: "Mild", "Severe", "Moderate"
  AE_OUTCOME:  "RECOVERED/RESOLVED", "NOT RECOVERED/NOT RESOLVED", "FATAL"

Output ONE markdown table, one row per raw value, columns exactly:
  raw value | proposed submission value | which layer matched
  (exact / case-only / synonym / NOT IN CODELIST) | confidence (high/low + one-line reason)

Rules:
- The proposed submission value must be byte-for-byte a string from the
  lists above, or the literal verdict NOT IN CODELIST.
- Synonyms may justify a mapping but may never appear in the proposed
  submission value column.
- This is a proposal. It will be verified by code against the published
  file; write nothing that assumes it is final.
```

The design choices, one line each:

- **Full codelist values pasted in** — Lesson 5's scar, inverted: `CT=OUT` gave the model a name instead of the list; this prompt never lets that happen again.
- **Every raw value enumerated, "handle every one"** — copied from the real fix in `/root/sdtm_review/services/value_domains.py`, whose prompt block warns that an unhandled value becomes a blank cell.
- **"Which layer matched" column** — Lesson 4's triage (exact / near-miss / synonym / violation) forced into the intern's own output, so its reasoning is inspectable, not just its answer.
- **Explicit NOT IN CODELIST verdict** — both codelists are non-extensible (Part 1's extensibility lesson): nothing can be invented, so the honest verdict must be a first-class output, the intern's version of "unchecked ≠ passed."
- **Synonyms as aids, never as output** — Part 1's four-name-types lesson: the CDISC FAQ's answer to "may synonyms be submitted?" is flatly "No."
- **Confidence column and "this is a proposal"** — the Lesson 5 principle in miniature: AI proposes, deterministic code disposes; the prompt itself refuses finality.

Prompt (c) — THE REFUTATION PROMPT

The mirror image: give the AI finished work and pay it to attack, not to agree. Note that the pasted mapping below deliberately contains problems — that is the point.

```
You are a hostile reviewer. Below is a codelist and a finished
controlled-terminology mapping someone claims is submission-ready. Your
job is to try to REFUTE each row using ONLY the codelist pasted here.

CODELIST OUT for AEOUT (NCI C66768, non-extensible). Submission values:
  FATAL
  NOT RECOVERED/NOT RESOLVED
  RECOVERED/RESOLVED
  RECOVERED/RESOLVED WITH SEQUELAE
  RECOVERING/RESOLVING
  UNKNOWN

THE MAPPING UNDER REVIEW:
  1. AE_OUTCOME "RECOVERED/RESOLVED"       -> AEOUT "RECOVERED/RESOLVED"
  2. AE_OUTCOME "FATAL"                    -> AEOUT "Death Related to Adverse Event"
  3. AE_OUTCOME "NOT RECOVERED/NOT RESOLVED" -> AEOUT "NOT RECOVERED"
  4. AE_SERIOUS "No"                       -> AESER "N"

For each row output exactly one verdict:
  CONFIRMED  - the proposed value is byte-for-byte on the pasted list.
  REFUTED    - it is not; state the correct submission value if one is
               unambiguous, otherwise say the row needs a human.
  CANNOT VERIFY - the pasted codelist does not govern this variable.
               This is your DEFAULT whenever the list above does not
               decide the question. Never upgrade it to CONFIRMED.

Do not use any codelist knowledge from your training memory. If you
catch yourself recalling a term that is not pasted above, that recall
is inadmissible.
```

Run it and check the verdicts yourself: row 1 is CONFIRMED; row 2 must be REFUTED — "Death Related to Adverse Event" is the NCI Preferred Term for C48275, and the submission value is `FATAL` (Part 1's best example that the two differ); row 3 is REFUTED as a truncation of `NOT RECOVERED/NOT RESOLVED`; row 4 is CANNOT VERIFY, because no AESER codelist was pasted, and an AI that "confirms" it anyway has just failed the interrogation. The default-to-cannot-verify rule is Lesson 4's fourth state — unchecked — imposed on a reviewer that would otherwise happily fill gaps from memory.

Exercise — reproduce the scar on demand

Lesson 5's disaster took months to surface in a real pipeline. You can now reproduce it in four minutes.

1. Paste Prompt (b) into a fresh chat. Save the table it returns.
2. Open a second fresh chat. Paste Prompt (b) again, but delete both CODELIST blocks and replace them with one line: `AE_SEVERITY uses codelist AESEV; AE_OUTCOME uses codelist OUT.` Names only — exactly what the old engine's `CT=OUT` prompt line gave the model.
3. Diff the two tables by eye. The severity rows may survive on luck (three famous values); watch the outcome rows and the layer column drift.
4. Repeat step 2 once more in a third chat and compare. Same prompt, possibly different answers — a reminder from Part 2 that a single model run is a draw, not a result, even before you starve it.

The with/without difference you just produced is the scar from Lesson 5, on demand, for free. Steps 1 and 2 differ by one thing only: whether the list was in the room.

Quiz 6 (answers at the end of Part 3)

1. Prompt (b) forbids the AI from outputting a synonym as the proposed submission value but allows synonyms to justify a match. Which two facts from Part 1 does that split encode?

2. In Prompt (c), why is CANNOT VERIFY the default verdict rather than a last resort?
3. Your colleague runs Prompt (b), gets a perfect-looking table, and says "great, the mapping is done." Name the missing step and the Lesson 5 sentence that names it.

TAKEAWAY: A PROMPT CAN RAISE THE ODDS OF A CORRECT PROPOSAL — PASTING THE CODELIST RAISES THEM ENORMOUSLY — BUT NO PROMPT CAN PRODUCE A VERDICT; ONLY THE INSPECTOR CAN.

Lesson 7 — Build the inspector: a 30-line CT checker from memory

PREDICT BEFORE YOU READ: YOUR CHECKER EXAMINES TWO OF A DATASET'S TWELVE COLUMNS AND FINDS NO PROBLEMS. WHAT IS THE HONEST ONE-LINE SUMMARY — AND WHAT IS THE DISHONEST ONE?

In Lesson 6 you kept saying "the inspector will verify it." Time to be the inspector. The rebuild rule: close this book after reading the spec, write the checker yourself from memory, and only then compare with the reference. What you cannot rebuild, you never owned.

The spec, in plain words

- **Input:** a DataFrame column, plus a codelist for that column given as two parts: the list of submission values, and a dictionary of synonyms (synonym string → the submission value it points to).
- **Output, per distinct value:** exactly one verdict —
  - **exact**: byte-for-byte a submission value; nothing to do.
  - **near-miss-case**: matches a submission value once both are lowercased; the fix is mechanical, and the checker must hand back the corrected string.
  - **synonym-used**: matches a published synonym; report the submission value it maps to — the synonym itself must never be proposed for submission.
  - **violation**: matches nothing, even forgiving case; a human decision, not a mechanical fix.
- **The summary rule:** for any column with no codelist on file, the report must say **UNCHECKED** — and must refuse to fold unchecked columns into any "pass." A summary that says "no findings" while seven columns went unexamined is Lesson 4's lie-with-good-typography.

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

The reference implementation

Runs standalone — pandas only, sample data inline mirroring the values in `/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv`, plus two planted edge cases.

```
import pandas as pd

CODELISTS = {  # submission values + published synonyms: AESEV (C66769), AEOUT (C66768)
    "AE_SEVERITY": (["MILD", "MODERATE", "SEVERE"],
                    {"grade 1": "MILD", "1": "MILD", "grade 2": "MODERATE",
                     "grade 3": "SEVERE"}),
    "AE_OUTCOME": (["FATAL", "NOT RECOVERED/NOT RESOLVED", "RECOVERED/RESOLVED",
                    "RECOVERED/RESOLVED WITH SEQUELAE", "RECOVERING/RESOLVING",
                    "UNKNOWN"],
                   {"unk": "UNKNOWN", "u": "UNKNOWN"}),
}

def check_column(series, submission, synonyms):
    by_lower = {s.lower(): s for s in submission}
    verdicts = {}
    for v in series.dropna().str.strip().unique():
        if v in submission:          verdicts[v] = ("exact", v)
        elif v.lower() in by_lower:  verdicts[v] = ("near-miss-case", by_lower[v.lower()])
        elif v.lower() in synonyms:  verdicts[v] = ("synonym-used", synonyms[v.lower()])
        else:                        verdicts[v] = ("violation", None)
    return verdicts

def report(df):
    for col in df.columns:
        if col not in CODELISTS:
            print(f"{col:12} UNCHECKED - no codelist on file; this is NOT a pass")
            continue
        for value, (verdict, fix) in check_column(df[col], *CODELISTS[col]).items():
            arrow = f" -> submit {fix!r}" if verdict not in ("exact", "violation") else ""
            print(f"{col:12} {value!r:24} {verdict}{arrow}")

df = pd.DataFrame({  # mirrors raw_ae.csv values, plus edge cases
    "AE_SEVERITY": ["Mild", "Severe", "Moderate", "Grade 1", "Sever"],
    "AE_OUTCOME":  ["RECOVERED/RESOLVED", "FATAL", "UNK", "Recovering/Resolving", "Ongoing"],
    "AE_SERIOUS":  ["No", "Yes", "No", "No", "Yes"],
})
report(df)
assert check_column(df["AE_SEVERITY"], *CODELISTS["AE_SEVERITY"])["Sever"][0] == "violation"
```

Run it and read the output against the spec. `Mild`, `Severe`, `Moderate` — the three real values from our 78-row file — all land in near-miss-case with the fix attached, exactly as the production checker in `/root/sdtm_review/services/ta_ct.py` classified them in Lesson 4. `Grade 1` rides the synonym column to MILD (the published AESEV synonym from Part 1's worked example) while MILD, not "Grade 1", is what gets submitted. `Sever` and `Ongoing` are violations: no case trick, no synonym, human required. And `AE_SERIOUS` prints UNCHECKED in so many words, because this checker would rather admit ignorance than manufacture a pass. The final `assert` is the checker checking itself — delete the `else` branch and it fails.

Notice what the ordering of the `if` chain encodes: exact beats case beats synonym. A value is judged at the strictest layer that accepts it, so the report never claims a synonym rescue for a value that was already legal.

Three upgrade katas

Don't implement these now — but be able to describe each in two sentences, because Lesson 8's clients will ask for all three.

1. **Load a real codelist instead of hand-typing one.** The published `SDTM Terminology.txt` is a tab-delimited file whose columns are: Code, Codelist Code, Codelist Extensible (Yes/No), Codelist Name, CDISC Submission Value, CDISC Synonym(s), CDISC Definition, NCI Preferred Term. The

kata: read it, select the rows whose Codelist Code is C66769, and build the same `(submission_values, synonyms)` pair this lesson hand-typed — splitting the synonym column on `;`, since MILD's entry reads "1; Grade 1". Your checker's data now comes from the same NCI EVS file the regulators point to, not from your memory of it.

2. **Extensible-codelist handling.** Add a third element per codelist — `extensible: yes/no` — plus a sponsor-additions list. For an extensible codelist (ROUTE, C66729, is the stock example), a value on the sponsor-additions list downgrades from violation to a warning: "sponsor extension — must be defined in define.xml and the Reviewer's Guide." For a non-extensible codelist (AESEV, AEOUT), the sponsor-additions list is ignored and the verdict stays violation — no configuration may talk the checker out of the published rule.
3. **Emit an evidence file.** Have `report()` also write a JSON file: every verdict, the run timestamp, and the codelist version string (the CT release date, e.g. the one in this machine's cached copy at `/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv` — the version date is what sponsors record in define.xml). House rule, straight from this project's engineering spine: **no evidence, no gate.** A checker whose output is scrollback that vanished proves nothing next week; a timestamped verdict file pinned to a codelist version is something a colleague — or an auditor — can re-run and compare.

Quiz 7 (answers at the end of Part 3)

1. Why does the spec force the checker to return the corrected string for a near-miss-case instead of just flagging it?
2. In kata 2, why must the sponsor-additions list be powerless against a non-extensible codelist?
3. Your rebuilt checker prints nothing at all for a column it has no codelist for. It has no false findings. Why does it still fail the spec?

TAKEAWAY: THE INSPECTOR IS THIRTY LINES, AND ITS HARDEST-WON LINE IS NOT A MATCH RULE — IT IS THE REFUSAL TO SAY "PASS" ABOUT ANYTHING IT DID NOT CHECK.

Lesson 8 — Different clients, same spine

PREDICT BEFORE YOU READ: THREE CLIENTS ASK FOR "CT HELP." ONE CONVERTS OLD STUDIES, ONE ONLY WANTS A PRE-SUBMISSION CHECK, ONE WANTS TO GOVERN THEIR OWN ADDED TERMS. DO THEY NEED THREE DIFFERENT SKILLS — OR ONE SKILL WITH THREE DIFFERENT THICK SPOTS?

Every CT engagement walks the same seven-stage spine you met in Part 1: (1) pin the CT version, (2) inventory the raw values, (3) propose the mapping, (4) apply it, (5) verify against the codelist, (6) document — define.xml and the Reviewer's Guide, (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 (done in an afternoon or skipped honestly). Learn to draw that map before quoting a price.

Client (a) — the CRO with a warehouse of legacy studies

A CRO wants bulk conversion of legacy studies to SDTM. The beginner's instinct is that the hard part is mapping volume. It isn't — the mappings repeat (every study has an AE_SEVERITY; you have seen all three of its values). The product here is **VERSION SKEW MANAGEMENT**. CT is released quarterly, versioned by date, with every previous version kept in archive subdirectories; studies run over many years, so different studies legitimately sit on different CT versions. FDA's Technical Conformance Guide says exactly this — sponsors should use the most recent version available at study start, different versions across studies in one application is common — but pooled analyses (an ISS) "should

be conducted using a single version of the terminology," version impact goes in the Reviewer's Guide, and PMDA (per the P21/Certara summary) formally requires one CT version across the whole submission. Skew is not a bug to apologize for; it is the deliverable to manage.

Reuse from the ClinCoder CT layer: the loader pattern of `/root/sdtm_review/services/ct.py` — one module that opens a terminology file and answers "which values does codelist X contain?" — plus the Lesson 7 triage. What changes: the loader must take the version as a parameter instead of hard-coding one file, and stage 1 stops being a formality and becomes a per-study ledger: study → CT version date → file it was verified against. The quarterly Changes files (NCI-EVS publishes a diff each release, including retired terms with mapping instructions; the diff tool itself is open source as nci-diff-cdisc) are your raw material for "what moved between this study's version and the pooling version." **The one gate:** every study's evidence file names the CT version its values were verified against — a verdict without a version date is inadmissible, because "valid" is only ever valid against a dated list.

Stage	1 pin version	2 inventory	3 propose	4 apply	5 verify	6 document	7 gate
Weight	**thick**	thick	thin (repeats)	thin	**thick**	thick (RG skew note)	thick

Client (b) — the small biotech that wants "just check our datasets"

A small biotech has SDTM datasets already built by a vendor and wants them checked before submission. No mapping, no conversion: stages 2, 3 and 4 are somebody else's finished work. You are being hired as stages 5 and 7 only — verification and gate — which is precisely what a validator rule like Pinnacle 21's CT2002 ("value not found in codelist") does, and community threads show it firing on both non-extensible codelists (RACE in DM) and extensible ones (EPOCH). Your Lesson 7 checker is a transparent, explainable sibling of that rule — and transparency is the sales pitch, because the client's real question is "will we get flagged, and why?"

Reuse: the whole Lesson 7 inspector, essentially unchanged, pointed at their datasets column by column; the triage matters commercially here, because a report that separates 78 near-miss-case findings (one mechanical fix) from 2 true violations (two human decisions) is a bill the client understands. What changes: nothing is generated, so the deliverable is the report itself — which makes the honesty rule the entire product. **The one gate:** every CT-governed variable in every dataset gets an explicit verdict, and every variable the checker could not examine is listed as UNCHECKED with the reason (external dictionary, no codelist, sponsor-defined). A check-only engagement that silently skips columns is worse than no engagement — the client walks into submission holding your green light.

Stage	1 pin version	2 inventory	3 propose	4 apply	5 verify	6 document	7 gate
Weight	thin (read theirs from define.xml)	thin	absent	absent	**thick**	thin	**thick**

Client (c) — the sponsor with in-house dictionaries

A sponsor maintains in-house dictionaries and wants governance for their extensible-codelist additions. The published rules give you the skeleton: a sponsor may add a term only where the codelist's Extensible column says Yes; FDA's TCG adds that creating custom terms "is discouraged," that extensibility must not be read as permission to substitute in-house terms for existing standard ones, and that unavoidable custom terms must be identified and defined in define.xml, referenced in the Reviewer's Guides, and used consistently. The P21/Certara position is blunter still: extensions to

standard terminology are often simply incorrect, the preferred route is requesting a new term from CDISC (two months before a package cut-off to make that release) — and many requested terms get rejected, which is itself information about your term.

One more risk belongs in the design even though no published incident of it was fetched for this book — treat this paragraph as design reasoning, not regulation. A sponsor extension coined today can collide with an official term published in a later quarterly release: same concept, different string, and now every ongoing study holds the unofficial spelling. The published mechanism that makes this survivable is the quarterly Changes file, which lists new and retired terms with mapping instructions — so governance means someone diffs every release against the sponsor-additions list. Reuse: kata 2's warn-not-fail checker and kata 3's evidence file. What changes: the sponsor-additions list becomes a governed artifact with an owner, a justification per term ("no official term existed as of release YYYY-MM-DD"), and a quarterly review step. **THE ONE GATE:** no term enters the sponsor list without recorded evidence that the current CT release was searched — submission values and synonyms, since the official term may exist under a spelling nobody at the sponsor uses — and no addition to a non-extensible codelist, ever, no matter who asks.

Stage	1 pin version	2 inventory	3 propose	4 apply	5 verify	6 document	7 gate
Weight	**thick** (quarterly diff)	thin	thin	thin	thick (warn tier)	**thick** (define.xml + RG)	thick

The client-scoping prompt

Three clients is a pattern; the fourth will be new. This prompt makes the AI interview you until the stage map falls out. Paste as-is at the start of any new engagement.

```
You are my scoping partner for a controlled-terminology (CT) engagement in
clinical data standards. I will describe a prospective client. Your job is
to 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 CT spine:
  1 pin the CT version   2 inventory raw values   3 propose mapping
  4 apply mapping        5 verify against codelist
  6 document (define.xml / Reviewer's Guide)      7 gate on evidence

Questions you must get answered (add your own as needed):
- Are datasets being CREATED, CHECKED, or GOVERNED? (This sets which
  stages exist at all.)
- One CT version or many? Any pooling (ISS) across studies? Any PMDA
  submission in scope?
- Any extensible-codelist additions or in-house dictionaries? Who owns
  them today?
- What evidence does the client's current process leave behind — files,
  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).

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

Quiz 8 (answers below)

1. Client (a) has two studies on different CT versions and wants a pooled safety analysis. Which stage suddenly thickens, and what does the FDA TCG say about the pooled analysis specifically?
2. Client (b)'s previous vendor's report showed "0 CT findings" across all domains. Name two very different things that could mean, and which lesson taught you the distinction.
3. Client (c) proposes adding "VERY SEVERE" to their AE severity dictionary. Two independent reasons your governance gate refuses — one from the codelist file itself, one from the FDA TCG.

TAKEAWAY: CLIENTS BUY THICK STAGES, NOT NEW SPINES — YOUR SKILL IS DRAWING THE SEVEN-STAGE MAP, NAMING THE ONE GATE, AND REFUSING TO SHIP WITHOUT ITS EVIDENCE.

Answers to Part 3 quizzes

Quiz 6

1. 1. Synonyms exist to help you find the correct submission value — the codelist publishes them as mapping aids; (ii) synonyms may never be submitted — the CDISC FAQ's answer is "No." So the prompt lets them justify a match but bans them from the output column.
2. Because a reviewer that treats "cannot verify" as a last resort will fill the gap from training memory — the exact failure mode of Lesson 5. Defaulting to CANNOT VERIFY makes admitting ignorance cheaper than inventing knowledge, which is the same design as the checker's UNCHECKED state.
3. The missing step is running the deterministic inspector over the table — checking every proposed value byte-for-byte against the published codelist. The Lesson 5 sentence: **AI proposes; deterministic code disposes.** A perfect-looking table is a proposal with good posture.

Quiz 7

1. Because the fix is mechanical and the checker already computed it: `Mild` lowercased equals `MILD` lowercased, so `"expected": "MILD"` costs nothing to attach and turns a review-queue item into a find-and-replace. Flag-only output pushes deterministic work back onto a human — the wrong direction.
2. Because extensibility is a property of the published codelist (the Extensible Yes/No column), not of the sponsor's configuration. If a local list could downgrade violations on AESEV or AEOUT, the checker would no longer verify against the standard — it would verify against the sponsor's wishes.
3. Silence is indistinguishable from "examined and clean." The spec requires the summary to say UNCHECKED for every column without a codelist, because the report's reader — a client, a reviewer, you in six months — cannot tell a column that passed from a column nobody looked at unless the report tells them.

Quiz 8

1. Stage 1 (pin the version) thickens into version reconciliation: the TCG accepts different CT versions across studies in one application, but says pooled analyses "should be conducted using a single version of the terminology" — so someone must re-express one study's values in the pooling version and document the impact in the Reviewer's Guide.
2. Either every checkable value was verified and conformed, or most variables were never checkable (external dictionaries, no codelist, dates) and the report folded "couldn't look" into "no findings." Lesson 4's fourth state — unchecked ≠ passed — is the distinction; your first question to the vendor is "show me the unchecked list."

3. 1. AESEV's codelist row carries Extensible = No — the file itself refuses additions, and kata 2's checker makes the sponsor list powerless there. (ii) The TCG discourages custom terms and forbids reading extensibility as license to substitute non-standard terms — and "VERY SEVERE" is a substitute for the existing standard term SEVERE, not a new concept.

The graduation checklist

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

- ☐ I can say what a submission value is, and why `Mild` on the CRF and `MILD` in the dataset are different objects.
- ☐ I can name the four name-types on a codelist row (submission value, synonym, NCI preferred term, definition) and state which one is submittable.
- ☐ Given `FATAL` / "Death Related to Adverse Event", I can say which is submitted and why the other exists.
- ☐ I can explain extensible vs non-extensible using one concrete codelist of each kind, and state what extension still requires (define.xml + Reviewer's Guide, not silence).
- ☐ I can tell the Lesson 5 scar story in under a minute: what `CT=OUT` withheld, why no alarm fired, and the sentence the project runs on now.
- ☐ I can write Prompt (b) from memory — codelist pasted in, every raw value enumerated, layer column, NOT IN CODELIST verdict — and say which lesson each feature comes from.
- ☐ I can run a refutation pass on someone else's mapping and default to "cannot verify" where the pasted codelist doesn't decide.
- ☐ I can rebuild the 30-line inspector from the spec: four verdicts, corrected string attached to near-misses, UNCHECKED never folded into a pass.
- ☐ I can describe all three katas — load the real terminology file, warn-tier sponsor additions, evidence JSON with codelist version — and say why "no evidence, no gate."
- ☐ I can scope a CT deliverable for a new client: draw the seven-stage thick/thin map, name the ONE gate, and state the first file I'd ask for.

Where to go next

- **The anatomy book.** When a lesson here makes you want the full internals, Module 2 of the Codebase Study Pack (Knowledge and terminology) documents every file this course simplified, at production depth: clincoder.cloud/study-packs/Codebase_Study_Pack_Vol1_SDTM_Mapper_Review.pdf
- **The next subject.** This teaching-pack format — concepts first, one scar, one toy build, AI-tutor prompts — repeats for any subject the anatomy book covers: code generation and sandboxing, validation gates, define.xml. The format is the tool; CT was just the first lesson plan.

Colophon

Written by three lesson-writer agents from (a) online NCI-EVS/CDISC sources gathered and cited this session and (b) the real CT layer of the ClinCoder mapping engine, then gated: every Python block parsed and resolved against the installed interpreter, every path tested against the host, and the whole book reviewed for contradictions before assembly. Built with pandoc and WeasyPrint on VPS2.