TEACHING PACK · CLINCODER LEARNER SERIES · VOLUME 2

Validation Gates in Clinical Data

Conformance rules, CDISC CORE, evidence discipline · and the true story of a gate that could not fail. Learn by interrogating a real tool with AI · predict, ask, verify, break, rebuild. A dataset can pass every rule and still be wrong — this book teaches the difference. 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: what it means for clinical data to be checked — the difference between a rule firing, a gate passing, and data actually being right — and how to build checks that can genuinely fail.

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, nothing to install beyond pandas), **Break** (do the break-it exercise), and **Rebuild** (write the toy from memory).

THE ONE RULE THAT CARRIES THE WHOLE SUBJECT: a gate that cannot fail is not a gate, and only a run writes evidence. Part 2 tells the true story of a gate in our own codebase that passed without testing anything, and every exercise in Part 3 exists to make the negative control — proving your gate fails on known-bad data — a habit.

Tool examples come from the real ClinCoder gate layer on VPS2 at the basis commits on the cover; regulatory and standards facts come from FDA and CDISC sources cited inline. All data is synthetic.

Part 1 — The concepts

Before Part 2 walks you inside the real validation gates running on this machine, you need three ideas that make everything there legible: what a validation rule actually is and who writes them, what severity and rejection really mean at a regulator's door, and why even a perfectly clean validation report is not the same thing as correct data. Everything in this part uses one tiny synthetic adverse-event table — imaginary patients, imaginary headaches — and facts taken directly from published FDA and CDISC sources.

Lesson 1 — What a validation rule is (and the three kinds)

PREDICT BEFORE YOU READ: YOUR CLINICAL DATASET IS ABOUT TO BE CHECKED BY VALIDATION SOFTWARE BEFORE IT GOES TO THE FDA. WHO WROTE THE RULES THAT SOFTWARE RUNS — THE SOFTWARE VENDOR, THE STANDARDS BODY THAT DESIGNED THE DATA FORMAT, OR THE REGULATOR ITSELF? PICK ONE; CHECK YOURSELF AT THE END.

The airport-security analogy

Think about the security line at an airport. Every bag goes through the same scanner, and the scanner applies the same checklist to each one: no liquids over the limit, no blades, nothing on the prohibited list. It is fast, it is consistent, and it catches a very specific kind of problem — known contraband, described in advance.

Now notice what the scanner does **NOT** do. It does not open your bag and confirm that the thing labeled "camera" is actually a camera and not a brick. It does not check that you packed what you meant to pack. A bag can sail through security and still be the wrong bag.

A **VALIDATION RULE** for clinical data is one line of that checklist: a precise, written-down check that a machine can apply to every row of every dataset, the same way, every time. And the scanner's limitation comes with it, exactly. Hold that thought for Lesson 3.

The data we are checking

One piece of vocabulary first. A finished drug trial submits its data to the FDA in a standard tabular format called **SDTM**, designed by **CDISC** (the clinical-data standards body). Its **AE dataset** holds one row per adverse event — a headache, a rash, a hospitalization — in columns with fixed names: `AEDECOD` is the standardized medical term, `AESEV` the severity, `AESER` a yes/no flag for "was this serious?". That is all the CDISC you need for this book.

Three kinds of rules, straight from the FDA

The FDA's own manual for submitted study data — the Study Data Technical Conformance Guide, or **TCG** (June 2026 edition, fda.gov) — says validation "helps ensure study data are compliant, useful, and will support meaningful review and analysis," and then lays out exactly three types of validation rules:

1. **Rules from standards development organizations** (e.g., CDISC), which "assess conformance to their published standards." Is your data shaped the way the standard says?
2. **FDA eCTD Technical Rejection Criteria for Study Data**, which check conformance to the standards in the FDA Data Standards Catalog. Is your submission package even openable at the door? (These are the automated reject gate — the star of Lesson 2.)

3. **FDA Business and Validator Rules**, which "assess if data support regulatory review and analysis." Even if it is shaped correctly, can a reviewer actually work with it?

Notice the division of labor: the standards body checks its own standard; the regulator checks its own door and its own reviewers' needs. Different authors, different questions, one checklist at the scanner.

Who publishes what, in numbers

- **CDISC** publishes conformance rules for its own standards. The current release, "SDTM and SDTMIG Conformance Rules v2.0" (29 Nov 2021, cdisc.org), covers SDTM through v2.0 and SDTMIG through v3.4 — that release alone added 59 new rules, retired 49, and updated 28. CDISC rule IDs look like **CG0042**.
- **FDA** publishes two artifacts on its Study Data Standards Resources page (fda.gov): **Business Rules v1.5** (May 2019) — plain-language statements of what review requires — and **Validator Rules v1.6** (December 2022) — the detailed machine checks that enforce them. FDA validator rule IDs look like **SD0002** or **CT2001**.
- **Pinnacle 21** (now part of Certara) is the best-known implementer: software that runs these rules. Its public rules pages (standards.pinnacle21.certara.net) organize SDTM rules in a table with Rule ID, Message, Description, Domains, and applicability flags for FDA, PMDA, and NMPA.

Here is the part that surprises almost everyone. Open the FDA Validator Rules v1.6 spreadsheet (we parsed it) and you find **732 RULES** — each with a "Publisher" column. The counts: **368 published by CDISC, 299 by FDA**, and 65 with no publisher listed. So the "P21 rule IDs" people talk about — `SD0002`, `CT2001` — are actually FDA Validator Rule IDs, and most of them are cross-references: each points back to a CDISC conformance rule (`CG####`) or an FDA business rule (`FDAB###`). One check, up to three names, depending on who is holding the checklist. Per Certara's explainer (certara.com), where no CDISC rule exists for a standard, the FDA rule simply carries no Publisher ID.

Three real rules for one AE table

Verbatim from the v1.6 spreadsheet, here are the three rules this whole part will keep reusing:

- **SD0002** (publisher: CDISC, cross-ref CG0014/CG0208). Message: "NULL value in variable marked as Required." Required variables "cannot be NULL for any records," in any domain. A blank `AEDECOD` trips this.
- **SD0009** (publisher: CDISC, cross-ref **CG0042**; AE domain). Message: "No qualifiers set to 'Y', when AE is Serious." When `AESER` is `'Y'`, at least one of the seriousness-criteria variables (`AESCAN`, `AESCONG`, `AESDISAB`, `AESDTH`, `AESHOSP`, `AESLIFE`, or `AESMIE`) is expected to be `'Y'`. In plain English: you cannot call an event "serious" without saying what made it serious — died? hospitalized? life-threatening?
- **CT2001** (publisher: FDA, cross-ref FDAB017). Message: "Variable value not found in non-extensible codelist." Some columns have a fixed menu of legal values that sponsors may not extend; `AESEV` allows exactly `MILD`, `MODERATE`, `SEVERE`, so a creative `SEVERE!!` is off the menu.

Notice the spread: one structural rule (a required field is empty), one logical rule (two fields contradict each other), one vocabulary rule (a value is off the published list). Most rules you will ever meet are one of these three flavors.

A three-rule scanner in thirty lines

Read this like the airport checklist: three rules, one bag at a time, and — crucially — the output is a list of **FINDINGS**, not a yes/no.

```
import pandas as pd

# Synthetic AE table: three imaginary events, one deliberately broken row.
ae = pd.DataFrame({
    "USUBJID": ["SYNTH-001-101001", "SYNTH-001-101002", "SYNTH-001-101003"],
    "AEDECOD": ["Headache", "", "Nausea"],
    "AESEV":   ["MILD", "SEVERE!!", "MODERATE"],
    "AESER":   ["N", "Y", "Y"],
    "AESDTH":  ["", "", "Y"],   # toy slice: 1 of the 7 seriousness criteria
})

AESEV_TERMS = {"MILD", "MODERATE", "SEVERE"}   # non-extensible codelist

def validate(df):
    findings = []
    # SD0002-style: a Required variable may never be NULL
    if "AEDECOD" not in df.columns:
        findings.append({"rule": "SD0002", "row": None,
                         "message": "Required variable AEDECOD absent from dataset"})
    else:
        for row, value in df["AEDECOD"].items():
            if value == "":
                findings.append({"rule": "SD0002", "row": row,
                                 "message": "NULL value in Required variable AEDECOD"})
    # SD0009-style: serious AE must name a seriousness criterion
    for row in df.index:
        if df.loc[row, "AESER"] == "Y" and df.loc[row, "AESDTH"] != "Y":
            findings.append({"rule": "SD0009", "row": row,
                             "message": "AESER is 'Y' but no criterion is 'Y'"})
    # CT2001-style: value must come from the non-extensible codelist
    for row, value in df["AESEV"].items():
        if value not in AESEV_TERMS:
            findings.append({"rule": "CT2001", "row": row,
                             "message": f"AESEV value {value!r} not in codelist"})
    return findings

for finding in validate(ae):
    print(finding)
```

Run it and three findings come back — all three pointing at row 1, the middle patient: a blank `AEDECOD`, a serious event with no stated criterion, and `SEVERE!!` off the codelist. One broken record, three named, located, countable problems. That shape is the design lesson: a finding tells a human which rule, which row, what to fix. A bare `False` would only tell them to start searching.

Break it on purpose

Now simulate the sloppiest possible dataset — the required column is not merely blank, it is gone:

```
#| skip-check
findings = validate(ae.drop(columns=["AEDECOD"]))
```

The first branch fires and you get one finding with `row: None`: the variable is absent from the dataset entirely. Two things to notice. First, a good rule checks for absence, not just blankness — “the column isn’t there” is the more catastrophic cousin of “the cell is empty,” and SD0002’s real-world text covers required variables being NULL “for any records.” Second, notice what did not happen: the validator did not crash on the missing column. A checker that crashes reports nothing, and nothing looks a lot like “no problems” to a hurried reader. Silence is the worst output a scanner can produce.

Quiz 1 (answers at the end of Part 1)

1. Name the three types of study data validation rules in the FDA's taxonomy, and for each, who publishes it.
2. `SD0009` and `CG0042` describe the same check on `AESER`. Why does one check have two IDs?
3. Our mini validator returns a list of dictionaries instead of `True` / `False`. Give two concrete things a reviewer can do with a finding that a boolean would not let them do.

TAKEAWAY: a validation rule is one line of a machine-run checklist; the checklist has three authors — the standards body (conformance), the regulator's door (rejection criteria), and the regulator's reviewers (business/validator rules) — and a good checker reports findings with a rule ID and a row, never a bare yes/no.

Sources: https://www.fda.gov/media/153632/download ; https://www.fda.gov/regulatory-information/search-fda-guidance-documents/study-data-technical-conformance-guide-technical-specifications-document ; https://www.cdisc.org/standards/foundational/sdtmig/sdtm-and-sdtmig-conformance-rules-v2-0 ; https://www.fda.gov/industry/data-standards/study-data-standards ; https://www.fda.gov/media/103587/download?attachment ; https://www.fda.gov/media/116935/download?attachment ; https://standards.pinnacle21.certara.net/validation-rules/SDTM ; https://www.certara.com/blog/new-fda-validator-rules-v1-6-explained/

Lesson 2 — Severity, rejection, and who actually enforces what

PREDICT BEFORE YOU READ: YOUR VALIDATION REPORT COMES BACK WITH 12 ERRORS AND 300 WARNINGS. HAS THE FDA REJECTED YOUR SUBMISSION? WILL IT? COMMIT TO AN ANSWER — REJECT, ACCEPT, OR "WRONG QUESTION" — AND CHECK YOURSELF AT THE END.

Two gates at the airport, not one

Go back to the airport, but look more carefully: there are two very different checkpoints. The first is the **BOARDING-GATE TURNSTILE** — a machine scans your boarding pass, and if the barcode does not read, the gate physically does not open. No conversation, no judgment, no appeal at the gate. The second is the **customs interview** — an officer looks at your declared items and asks questions. Odd things in your bag do not turn you around at the border; they start a conversation, and a good explanation ends it.

Submitted study data faces exactly this pair. A tiny set of automated checks can bounce the whole submission at the door, unread; everything else — including most of what your report calls "Error" — is customs: material for a conversation with a human reviewer, to be fixed or explained. Confusing the two gates is the classic beginner mistake, in both directions.

Error, Warning, Notice — whose words are these?

Open a Pinnacle 21 validation report and, per a PharmaSUG 2019 walkthrough of exactly these reports (pharmasug.org, paper DS-119), it "contains information as errors, warnings, and notices," with an Issue Summary tab that "breaks down issues by severity (error, warning, and notice)." Same paper: "Errors always have High severity; whereas Warnings have either Low or Medium severity. Errors must be corrected; however, Warnings should be corrected," while noting that "some warning and errors may be acceptable" depending on the study.

So severity is real and useful. But here is the twist worth a highlighter: **THOSE SEVERITY LABELS ARE THE TOOL'S, NOT THE FDA'S.** Severity was dropped from the FDA's own Validator Rules before v1.6, because

the CDISC Conformance Rules and FDA Business Rules they derive from publish no severity at all (certara.com) — and parsing the v1.6 spreadsheet confirms it: no severity column. Error/Warning/Notice is a tool-side classification layered on top of the published rules; two tools could rank the same finding differently, and neither ranking is "what FDA says."

The turnstile: Technical Rejection Criteria

What actually bounces a submission is much narrower. The TCG (Section 8.1.2 and Appendix F, fda.gov) describes the **TECHNICAL REJECTION CRITERIA (TRC)** as "automated validations by the Center (CDER or CBER) inbound processing system using the FDA Specifications for eCTD Validation Criteria." For the current eCTD v3.2.2 submission format, the TRC map to a handful of specific validation error numbers:

- **1789** — the study tagging file must be valid;
- **1734** — the TS (Trial Summary) dataset must be present;
- **1736** — required dataset files must be present;
- **1735** — files must carry the correct file tag.

(Under the newer eCTD v4.0 format the same gate rides on error codes US-eCTD4-516/517.) And the TCG states the mechanism plainly: "If there are no high validation errors within the eCTD submission, the submission will continue to be processed." Read that again with the emphasis reversed: high eCTD errors stop processing — the turnstile locks. A missing Trial Summary dataset is a rejection. A `SEVERE!!` in row 1? That is not on the turnstile's list. It is customs material.

The customs interview: correct or explain

For everything the turnstile lets through, the TCG (Section 8.1.2) gives sponsors one duty, and it is worth quoting exactly: sponsors "should evaluate their study data before submission" against the SDO conformance rules, the TRC, and the FDA Business Rules, and "should either correct any discrepancies… or explain meaningful discrepancies in the relevant Reviewer Guide (RG)."

Correct or explain. Not "achieve zero findings." The Reviewer Guides are the designated place for those explanations — the TCG calls RGs "recommended as an integral part of a standards-compliant study data submission," describing "any special considerations or directions or conformance issues that may facilitate an FDA reviewer's use of the submitted data." There is one for clinical study data (the **cSDRG**, filed in eCTD Module 5), one for nonclinical (the **nSDRG**, Module 4), and the **ADRG** for analysis data, which "provides a summary of ADaM conformance findings." A PharmaSUG 2023 paper on sponsor practice (pharmasug.org, SS-059) completes the loop: where sponsors deviate from standard approaches, validation software "may report a warning message, which should be explained in the reviewer guide."

This reframes what a validation report is: not a pass stamp you earn, but the opening move of a documented conversation with a reviewer. A finding you understood and explained in the SDRG is a healthy submission.

Findings, now with severity

Take Lesson 1's findings, add the one field this lesson is about, and split the two gates apart. Standalone code — the findings list is written out as Lesson 1's run (plus one new, nastier problem) would produce it:

```
findings = [
    {"rule": "TS-MISSING", "row": None,
     "message": "TS (Trial Summary) dataset not found in submission"},
    {"rule": "SD0002", "row": 1,
     "message": "NULL value in Required variable AEDECOD"},
    {"rule": "SD0009", "row": 1,
     "message": "AESER is 'Y' but no criterion is 'Y'"},
    {"rule": "CT2001", "row": 1,
     "message": "AESEV value 'SEVERE!!' not in codelist"},
]

# Tool-side severity ladder (Reject models the TRC turnstile;
# Error/Warning/Notice is P21-style tool vocabulary, not FDA-published).
SEVERITY = {
    "TS-MISSING": "Reject",   # toy id echoing eCTD error 1734: TS must be present
    "SD0002": "Error",
    "SD0009": "Error",
    "CT2001": "Error",
}

for f in findings:
    f["severity"] = SEVERITY[f["rule"]]

reject_class  = [f for f in findings if f["severity"] == "Reject"]
explain_class = [f for f in findings if f["severity"] != "Reject"]

print(f"REJECT-CLASS  (turnstile: fix, or the submission bounces): {len(reject_class)}")
for f in reject_class:
    print(f"   {f['rule']:<10} {f['message']}")
print(f"EXPLAIN-CLASS (customs: correct, or explain in the Reviewer Guide): {len(explain_class)}")
for f in explain_class:
    print(f"   {f['rule']:<10} [{f['severity']}] row={f['row']} {f['message']}")
```

Run it and the report splits cleanly: one finding that threatens the submission itself, three that a reviewer will expect you to have fixed or explained. Same list, two utterly different consequences — and the severity field is what encodes the difference.

Break it on purpose: severity inflation and deflation

Now sabotage one line — replace the ladder with a shrug:

```
#| skip-check
SEVERITY = dict.fromkeys(SEVERITY, "Warning")
```

Re-run, and watch the reject-class count drop to zero. The missing TS dataset — a genuine turnstile-locker under eCTD error 1734 — now sits quietly among the warnings, and nothing in the report's shape says "this one ends the submission." That is severity **deflation**, and it is not hypothetical sabotage: severity is tool-side, remember, so it is exactly the kind of thing a config change or a home-grown checker can get wrong with no rule text changing at all.

The opposite failure is just as real. Mark everything `Error` — severity **inflation** — and the team burns weeks "fixing" findings that DS-119 plainly says "may be acceptable" and merely need an SDRG explanation, while the one finding that could bounce the submission waits in the same undifferentiated pile. Severity is a routing decision: inflate it and you exhaust people; deflate it and you blindside them. Both failures leave the rule logic untouched — which is why reports, not just rules, deserve review.

Quiz 2 (answers at the end of Part 1)

1. Error/Warning/Notice appear in every P21 report. Who publishes those severity levels — CDISC, FDA, or the tool — and what changed before FDA Validator Rules v1.6?
2. Which of these two findings can bounce a submission unread, and which starts a conversation: (a) TS dataset absent (eCTD error 1734); (b) `SD0009` fires because a serious AE names no seriousness criterion?
3. Complete the TCG's sentence about what sponsors should do with discrepancies found before submission: "correct any discrepancies… or ____." Where does that second option physically live in a submission?

TAKEAWAY: severity is the tool's routing label, rejection is the regulator's narrow automated turnstile, and everything in between runs on one duty — correct or explain in the Reviewer Guide — because a validation report is the start of a conversation with a reviewer, not a pass stamp.

Sources: https://pharmasug.org/proceedings/2019/DS/PharmaSUG-2019-DS-119.pdf ; https://www.certara.com/blog/new-fda-validator-rules-v1-6-explained/ ; https://www.fda.gov/media/153632/download ; https://www.fda.gov/drugs/electronic-regulatory-submission-and-review/ectd-submission-standards-ectdv322-and-regional-m1 ; https://www.fda.gov/media/179723/download?attachment ; https://pharmasug.org/proceedings/2023/SS/PharmaSUG-2023-SS-059.pdf

---

Lesson 3 — CORE, engines, and why a clean report is not clean data

PREDICT BEFORE YOU READ: TWO VALIDATION TOOLS, BOTH CLAIMING TO IMPLEMENT THE SAME PUBLISHED RULES, RUN AGAINST THE SAME DATASET. DO THEY REPORT THE SAME FINDINGS? IF YOUR INSTINCT SAYS "OBVIOUSLY YES — SAME RULES, SAME DATA," HOLD THAT INSTINCT UP TO THE LIGHT AT THE END.

When the rulebook is prose, every referee is different

For years, conformance rules were published the way laws are: as documents. CDISC shipped its rules in Excel spreadsheets — precise-looking prose in cells — and every vendor, sponsor, and in-house team wrote their own code to enforce them. CDISC's own diagnosis of how that went is the reason this lesson exists: rules published that way "are not machine-executable and get interpreted inconsistently across organizations" (cdisc.org). Same sentence, five implementations, five subtly different referees. Two validators can disagree on the same data, and the disagreement is not a bug in either one — it is the gap between prose and code.

CORE: the rulebook that runs

CDISC's answer is **CORE** — the CDISC Open Rules Engine, part of the CDISC Open Rules Project, whose stated aim is "a governed set of unambiguous executable Conformance Rules" for each foundational standard (cdisc.org). Instead of prose that every team re-interprets, the rule is code: rules are authored in YAML (USDM rules in JSONata), stored in the CDISC Library, and fetched via its API — a free API key is all you need. The engine that runs them is open source under the MIT license, registered under COSA (the CDISC Open Source Alliance), free to members and non-members alike, with CDISC stating no plans to commercialize; the open-source transition to GitHub was completed on 31 August 2022 (github.com/cdisc-org/cdisc-rules-engine).

The state of play, per cdisc.org as of early 2026: **1,563 OF 1,880 TARGET RULES COMPLETE — 83%** — spanning SDTMIG, SENDIG, the FDA Business Rules, TIG, and USDM. FDA signed a research collaboration agreement with CDISC in 2024 (CDER/CBER) to incorporate the FDA Business Rules into CORE, and

CORE v1.0 is targeted for mid-2026, merging the reference engine with Verisian's SQL-based engine from the 2025 partnership. Releases have been steady — v0.13.0 in October 2025 through **v0.16.0 on 1 May 2026** (GitHub). Running it looks like this:

```
./core validate -s sdtmig -v 3-4 -d /path/to/datasets
```

— standard, version, data folder; flags exist for a Define-XML path, a controlled-terminology package, MedDRA/WHODrug dictionaries, and JSON/XLSX/CSV output. One detail deserves special affection: each rule in a CORE report ends in one of four statuses — **SUCCESS, SKIPPED, ISSUE REPORTED, OR EXECUTION ERROR** (github.com, CLI reference). That last one makes "the rule itself crashed" its own visible outcome, distinct from "no issues found." Remember Lesson 1: silence is the worst output a scanner can produce, and CORE refuses to let a broken check impersonate a passing one.

Two boring-sounding facts here carry real teeth. First, CORE is the Reference Implementation: vendors may adapt it or build their own engines, and a certification program is planned to verify that third-party engines produce results matching the reference (cdisc.org) — that is the long-term cure for the five-referees problem. Second, **engine version drift**: Certara's own guidance warns that running outdated FDA engines "means results won't match what the agency sees" (certara.com). Your validator's version is part of your result. Two honest reports can disagree because one of them is simply older.

The honest boundary: what a clean report does not say

Now the most important paragraph in Part 1. Suppose the report comes back clean — every rule SUCCESS, zero findings. What do you actually know?

Stay strictly with what the sources say. The PharmaSUG paper that walks through P21 reports states it flatly: "Pinnacle 21 does not guarantee 100% compliance. However, it does a good job detecting most data issues that would otherwise delay submission" (pharmasug.org). And look again at the FDA's own taxonomy from Lesson 1: conformance rules "assess conformance to their published standards"; business and validator rules "assess if data support regulatory review and analysis." Compliant. Useful. Reviewable. Nowhere in that vocabulary does any rule claim to check that the values are true — that the severity in row 1 matches what the site actually recorded, that the derivation that produced `AESER` was performed correctly, that the numbers are the right numbers.

Back to the airport one last time: the scanner clears the bag of known contraband; it never certifies that the bag contains what the luggage tag promises. A dataset can pass every published rule and still be wrong data — wrong values, wrongly derived, faithfully formatted. Conformance is a property of shape; correctness is a property of content; the checklist only sees shape. Part 2 shows what that gap looks like in a real pipeline on this machine — a baseline where conformance-passing programs scored 0/11 on data equivalence, and a gate that glowed green while proving nothing — but the scar tissue belongs there, not here.

Where validation lives: stage 5 of 7, and only part of it

This book series hangs every tool on the same seven-stage spine: **1 INGEST → 2 INVENTORY → 3 KNOWLEDGE → 4 PROPOSE → 5 VERIFY → 6 REVIEW → 7 DELIVER + EVIDENCE**. Raw files arrive untouched; the tool takes stock; reference material (standards, codelists, rules) is loaded; something drafts the work; deterministic checks re-derive whether the draft is legal; humans handle what checks cannot settle; outputs ship with proof of what ran.

Everything in this part — SD0002, the TRC turnstile, CORE — lives at **STAGE 5, VERIFY**. But place it precisely: conformance rules are only one kind of verify. They answer "is this shaped like the

standard says?" A full stage 5 also has to answer "does this data equal what it should equal?" — data-equivalence checking — and stage 7 has to answer "can I prove the checks actually ran?" — evidence discipline. Those two are Part 2's subject, and the reason it exists: a pipeline whose entire stage 5 is a conformance checker has verified its grammar and taken its arithmetic on faith.

Exercise: three findings, three one-liners (no code this lesson)

You are preparing the SDRG for study SYNTH-001. Validation produced three findings. For each, write the one-line response that belongs in the Reviewer Guide — or say why no line can help. (Answers at the end of Part 1.)

1. **TRC-class:** TS (Trial Summary) dataset not found in the submission (maps to eCTD error 1734).
2. **Explainable warning:** a warning fires because a sponsor-specific deviation from the standard approach was used, exactly as SS-059 describes.
3. **False positive:** a rule flags a value as off-codelist, but against the CT version your define.xml pins, the value is legal — the validator was run with a different terminology vintage.

Quiz 3 (answers below)

1. Why did CDISC build CORE? Quote or paraphrase the specific problem with Excel-published rules, and name the mechanism (a planned program) meant to keep third-party engines honest.
2. A CORE report line reads EXECUTION ERROR. What happened, and why is it a separate status from SUCCESS with no findings?
3. Your dataset passes every conformance rule in the current engine. Name two distinct things that can still be wrong, using this lesson's vocabulary of shape vs content and the seven-stage spine.

TAKEAWAY: CORE turns prose rules into one executable, open-source referee — but even a perfect referee only judges shape; a clean conformance report tells you the data is compliant and reviewable, never that it is correct, which is why stage 5 of the spine needs more than rules.

Sources: https://www.cdisc.org/core ; https://github.com/cdisc-org/cdisc-rules-engine ; https://github.com/cdisc-org/cdisc-rules-engine/blob/main/README.md ; https://raw.githubusercontent.com/cdisc-org/cdisc-rules-engine/main/docs/cli-reference.md ; https://api.github.com/repos/cdisc-org/cdisc-rules-engine/releases ; https://pharmasug.org/proceedings/2019/DS/PharmaSUG-2019-DS-119.pdf ; https://www.certara.com/blog/new-fda-validator-rules-v1-6-explained/ ; https://www.fda.gov/media/153632/download ; https://pharmasug.org/proceedings/2023/SS/PharmaSUG-2023-SS-059.pdf

---

Answers to Part 1 quizzes

Quiz 1

1. **(1) Standards-development-organization rules** — published by CDISC for its own standards (the conformance rules, IDs like CG0042); **(2) the eCTD Technical Rejection Criteria for Study Data** — published by FDA, checking conformance to the standards in the FDA Data Standards Catalog at the door; **(3) FDA Business and Validator Rules** — published by FDA, which "assess if data support regulatory review and analysis."
2. Because they are **the same check held by two publishers**. CDISC wrote the conformance rule for its own standard (CG0042); FDA's Validator Rules v1.6 carries it as SD0009 with CDISC listed in its Publisher column and CG0042 as the Publisher ID. Of the 732 rules in that spreadsheet, 368 are

CDISC-published and 299 FDA-published — most "P21 rule IDs" are really FDA Validator Rule IDs cross-referencing someone else's rule.

3. A finding names **which rule** fired and **which row** it fired on, so a reviewer can (a) go straight to the offending record and fix precisely that, and (b) count, sort, and route findings by rule — impossible with a single `False`, which only says "something, somewhere." (Also fair: a finding can be explained in a Reviewer Guide; a boolean cannot.)

Quiz 2

1. **The tool publishes them.** Error/Warning/Notice is P21-style tool vocabulary (per the PharmaSUG DS-119 description of the report). Before v1.6, FDA **dropped severity from its own Validator Rules** — because the CDISC Conformance Rules and FDA Business Rules they derive from publish no severity — so today no severity column exists in the FDA spreadsheet, and any severity you see was assigned tool-side.
2. **(a) bounces; (b) converses.** A missing TS dataset maps to eCTD error 1734, one of the Technical Rejection Criteria enforced by the Center's automated inbound processing — a high eCTD error stops processing unread. SD0009 is a validator-rule finding: serious material for review, but its path is correct-or-explain, not automatic rejection.
3. "...or **explain meaningful discrepancies in the relevant Reviewer Guide (RG)**" (TCG Section 8.1.2). Physically, that explanation lives in the Reviewer Guides shipped with the submission: the cSDRG in eCTD Module 5 for clinical study data, the nSDRG in Module 4 for nonclinical, and the ADRG for analysis data.

Quiz 3

1. Because Excel-published rules "are not machine-executable and get interpreted inconsistently across organizations" — every implementer became a slightly different referee. The keep-honest mechanism is the planned **certification program** verifying that third-party engines produce results matching CORE, the Reference Implementation.
2. **The rule itself failed to run** — the check crashed or could not execute, which is entirely different from "the check ran and found nothing." Making it a distinct status stops a broken check from impersonating a passing one; without it, an engine bug would look exactly like clean data.
3. Any two of: the **content can be wrong even though the shape is right** — values untrue to source, since conformance rules assess conformance to standards and reviewability, never truth (and the tool itself "does not guarantee 100% compliance"); **stage 5 is incomplete** — rules are only one kind of verify, and no data-equivalence check has confirmed the numbers equal what they should; **stage 7 is unproven** — without evidence of what ran, against which engine and version, a "clean" claim cannot be audited (and version drift means an outdated engine's results "won't match what the agency sees").

Exercise (the three SDRG one-liners)

1. **No one-liner can help — this one must be fixed.** The TRC are automated rejection criteria applied by the inbound processing system before any reviewer opens the Reviewer Guide; an explanation cannot be read by a turnstile. Add the TS dataset and re-validate.
2. Something like: "Rule X reports a warning on [variable]: the study deviated from the standard approach because [reason]; the data are complete and consistent as submitted, and the deviation is described in Section [n] of this guide." This is exactly the SS-059 pattern — sponsor deviation, validator warning, explanation in the reviewer guide — and the TCG's "explain meaningful discrepancies in the relevant RG" duty in action.

3. Something like: “Rule Y flags [value] against codelist [Z]; the submission’s define.xml pins CT version [date], in which [value] is a published term — the finding reflects the validation engine’s terminology vintage, not a data discrepancy.” A false positive still gets a line: a reviewer running different engine settings will see the same finding, and the TCG asks for conformance information “that could facilitate review” to be provided in the RG.

Part 2 — Inside real gates

In Part 1 you learned what a validation gate is: a small, deterministic check that stands between a piece of work and the word "done," and that must be able to say no. Now we go where most tutorials never take you — inside a real, working validation system that runs on this machine, watching one deliberately broken adverse-event row get caught by name.

And then, because this is a teaching book and not a brochure, we will study the day a gate on this very box said yes for weeks while proving nothing at all. Not a hypothetical — the retraction is still written into the code's own docstring, and you will read it verbatim. The rule that scar bought is the single most important sentence in this book: **ONLY A RUN WRITES EVIDENCE; A GATE THAT CANNOT FAIL IS NOT A GATE.**

All data in this part is synthetic. Our patients — subjects `101001` and `101002` of imaginary study SYNTH-001 — have imaginary headaches. No real person appears anywhere in this book.

Lesson 4 — A rule fires: watching a broken AE row get caught

PREDICT BEFORE YOU READ: A VALIDATION RULE CRASHES HALFWAY THROUGH CHECKING A DATASET — IT HITS A BUG IN THE RULE, NOT THE DATA. SHOULD THE REPORT SAY "PASS" (THE RULE FOUND NOTHING), "FAIL" (SOMETHING'S WRONG), OR SOMETHING ELSE? WHAT WOULD EACH CHOICE COST?

Hold your answer. Let's build the row, walk it through a real engine's logic, and see what the honest choice is.

One broken row, three defects

Here is a two-row adverse-event table. The first row is clean. The second row carries three planted defects — one of each classic kind:

```
USUBJID            AETERM     AEDECOD    AESEV      AESTDTC
SYNTH-001-101001   HEADACHE   Headache   MODERATE   2026-01-05
SYNTH-001-101002   NAUSEA     (blank)    SEVERE!!   05JAN2026
```

Defect one: `AEDECOD` — the coded medical term, a Required variable in SDTM — is blank. Defect two: `AESEV`, the severity, says `SEVERE!!`, and no codelist on Earth contains `SEVERE!!`; the legal values are exactly `MILD`, `MODERATE`, `SEVERE`. Defect three: the start date is written `05JAN2026`, a SAS-style date, where the standard demands ISO 8601 (`2026-01-05`).

A human reviewer would spot all three in about four seconds. The question this lesson answers is: how does a program spot them, and — more interesting — what shape does its answer take?

What a rule actually is

Open `/root/sdtm_review/services/validator.py`, the in-house conformance checker in the review tool. A "rule" there is not an abstract object or a config file. It is a short block of ordinary code that looks at the data and, if it dislikes what it sees, appends one small dictionary — a **finding** — to a growing list. The helper that does the appending takes six fields: a rule id, a severity, the variable concerned, a status, a human-readable message, and a count. That six-field row is the finding shape for the whole engine. Our blank `AEDECOD` produces, in the real tool, a finding essentially like this:

```
#| skip-check
{"id": "CG0030", "severity": "Error", "variable": "AEDECOD",
 "status": "fail", "message": "AEDECOD is blank on 1 record(s).", "count": 1}
```

Notice everything that little dict is not. It is not a boolean. It is not an exception that halts the program. It is not a percentage. It names the rule, the variable, the count, and what happened — so a reviewer can go fix the one thing it points at. The severities are ranked (`Reject` worst, then `Error`, `Warning`, `Notice` — the file's own line: `SEV_WEIGHT = {"Reject": 10, "Error": 5, "Warning": 2, "Notice": 0}`), and the overall verdict is just a ladder: any Reject makes the report Reject, else any Error makes it Error, and so on down to Pass. Findings first; the verdict is merely a summary of the findings.

That is the first design lesson of this whole part: **A GOOD VALIDATOR RETURNS FINDINGS, NOT BOOLEANS.** A `False` tells you to start searching. A finding tells you where to stop.

The three verdicts — and why there must be three

Now the prediction question. The project's wrapper around CDISC's official rules engine, `/root/sdtm_review/services/core_engine.py`, keeps rule outcomes in distinct buckets and refuses to merge them:

- **pass** — the rule ran, and found nothing wrong.
- **fail** — the rule ran, and found something wrong. This is a statement about the data.
- **execution_error** — the rule itself could not run (it crashed, or was missing an input). This is a statement about the checker, and it says nothing about the data at all.

And it goes one step further: any status it doesn't recognise is filed as `execution_error`, on the written grounds that an unknown status is not good news. Real numbers make the stakes vivid. A stored run over the Cardiology study (measured in the anatomy book session) showed 89 rules passed, 12 failed — and 177 ended in execution errors. Now imagine collapsing those buckets. Fold `execution_error` into `pass` and you claim 266 of 278 clean — when 177 of those "clean" rules never examined anything. Fold it into `fail` and you bury 12 real data problems under 177 checker problems, and the reviewer learns to ignore the report. Either collapse turns a measurement into a lie; they just lie in opposite directions.

So the answer to the predict question is: **SOMETHING ELSE** — a third verdict, kept separate all the way to the screen. "I could not look" is a different sentence from "I looked and it's fine" and from "I looked and it's broken," and a report that can't say all three sentences will eventually say a false one.

The honest boundary: conformance is not correctness

One more idea before you build your own validator, and it is the most important limit in this lesson.

Every rule above checks conformance — is the required column populated, is the value on the codelist, is the date in the right format. None of them checks correctness — is this the right value for this patient. A dataset where every severity was accidentally shifted by one row would pass all three rules gloriously: every cell populated, every value legal, every date ISO. Structurally perfect, factually scrambled.

This project measured that gap instead of hand-waving it. When the pipeline's outputs were finally graded the hard way — every produced cell compared against a known-correct answer dataset — the first honest baseline was **0 OF 11 DOMAINS CORRECT** (measured in the anatomy book session), on outputs that were sailing through conformance checks. Conformance rules are the spell-checker; correctness needs a fact-checker, and those are different machines. A validator that only speaks conformance

must say so — which is why the tools in this repo carry disclaimers in their own source stating that a clean report is not clinical review.

Try it yourself — a three-rule validator that returns findings

Everything above, runnable, in one screen. Two rows, three rules, findings not booleans, three possible verdicts per rule:

```
import re
import pandas as pd

ae = pd.DataFrame([
    {"USUBJID": "SYNTH-001-101001", "AETERM": "HEADACHE", "AEDECOD": "Headache",
     "AESEV": "MODERATE", "AESTDTC": "2026-01-05"},
    {"USUBJID": "SYNTH-001-101002", "AETERM": "NAUSEA", "AEDECOD": "",
     "AESEV": "SEVERE!!", "AESTDTC": "05JAN2026"},
])

ISO = re.compile(r"^\d{4}(-\d{2}(-\d{2})?)?$")
CT_AESEV = {"MILD", "MODERATE", "SEVERE"}

def required_populated(df):
    findings = []
    for var in ("USUBJID", "AETERM", "AEDECOD"):
        n = int((df[var].str.strip() == "").sum())
        if n:
            findings.append({"variable": var, "message": f"blank on {n} record(s)"})
    return findings

def ct_membership(df):
    bad = sorted(set(df["AESEV"]) - CT_AESEV)
    return [{"variable": "AESEV", "message": f"value(s) outside codelist: {bad}"}] if bad else []

def iso_dates(df):
    bad = sorted(v for v in df["AESTDTC"] if not ISO.fullmatch(v[:10]))
    return [{"variable": "AESTDTC", "message": f"non-ISO date(s): {bad}"}] if bad else []

RULES = [("required_populated", required_populated),
         ("ct_membership", ct_membership),
         ("iso_dates", iso_dates)]

def run_rules(df, rules):
    results = []
    for name, fn in rules:
        try:
            findings = fn(df)
            status = "fail" if findings else "pass"
        except Exception as e:
            status = "execution_error"
            findings = [{"variable": "?", "message": f"rule raised {type(e).__name__}: {e}"}]
        results.append({"rule": name, "status": status, "findings": findings})
    return results

for r in run_rules(ae, RULES):
    print(r["rule"], "->", r["status"])
    for f in r["findings"]:
        print("   ", f["variable"], "|", f["message"])
assert [r["status"] for r in run_rules(ae, RULES)] == ["fail", "fail", "fail"]
```

Output on this box:

```
required_populated -> fail
    AEDECOD | blank on 1 record(s)
ct_membership -> fail
    AESEV | value(s) outside codelist: ['SEVERE!!']
iso_dates -> fail
    AESTDTC | non-ISO date(s): ['05JAN2026']
```

All three planted defects, each caught by name, each with the exact variable and count a fixer needs. Read the `run_rules` loop again: the `try/except` is where the three-verdict philosophy lives. A rule that returns findings means `fail`; a rule that returns nothing means `pass`; a rule that blows up is caught and filed as `execution_error` — and, crucially, the crash of one rule doesn't stop the other rules from running.

Break it — make the checker itself crash

Now watch the classic lie up close. Add a fourth rule with a bug in it — a typo'd column name — and feed it data with a real defect too:

```
def run_rules(rows, rules):
    results = []
    for name, fn in rules:
        try:
            findings = fn(rows)
            status = "fail" if findings else "pass"
        except Exception as e:
            status = "execution_error"
            findings = [{"message": f"rule raised {type(e).__name__}: {e}"}]
        results.append({"rule": name, "status": status, "findings": findings})
    return results

rows = [{"AESEV": "MILD"}, {"AESEV": "Grade 3"}]

def ct_membership(rows):
    bad = sorted({r["AESEV"] for r in rows} - {"MILD", "MODERATE", "SEVERE"})
    return [{"message": f"outside codelist: {bad}"}] if bad else []

def broken_rule(rows):
    return [r["AESVE"] for r in rows]        # typo: no such column

for r in run_rules(rows, [("ct_membership", ct_membership),
                          ("broken_rule", broken_rule)]):
    print(f'{r["rule"]:14} -> {r["status"]:15}',
          r["findings"][0]["message"] if r["findings"] else "")
```

Output:

```
ct_membership  -> fail            outside codelist: ['Grade 3']
broken_rule    -> execution_error rule raised KeyError: 'AESVE'
```

Two different sentences, kept apart. Now do the thought experiment the collapse-happy designer skips. Change the `except` block to set `status = "pass"` ("the rule found no findings, after all") and the broken rule quietly greenlights every dataset forever — a checker bug becomes invisible clean data. Change it to `status = "fail"` and the report claims a data problem that doesn't exist; the team burns an afternoon hunting a defect in the wrong file, and next time they trust the report less. The distinct third verdict costs one word in the report and buys you the difference between "our checker is broken" and "our data is broken" — two problems with entirely different owners.

Un-break it (fix `AESVE` to `AESEV`) and the fourth rule drops to an honest `fail` on `Grade 3`, which is a genuine data finding.

Quiz 4 (answers at the end of Part 2)

1. Your validator returns `False` for a 40-column dataset. Your colleague's returns `{"rule": "required_populated", "variable": "AEDECOD", "message": "blank on 1 record(s)"}`. Both are "correct." What can the person reading the second report do that the person reading the first cannot?
2. In the stored Cardiology run, 89 rules passed, 12 failed, and 177 ended in execution errors. A dashboard shows "89/101 rules pass (88%)." Is that number defensible, and what must sit beside it to keep it honest?
3. A dataset passes every rule in this lesson — nothing blank, every value on its codelist, every date ISO. Name one way it can still be completely wrong, and which kind of check (conformance or correctness) would catch it.

TAKEAWAY: A REAL VALIDATOR ANSWERS IN FINDINGS, NOT BOOLEANS, AND IT KEEPS THREE VERDICTS APART — THE DATA FAILED, THE DATA PASSED, OR THE CHECKER ITSELF COULDN'T LOOK — BECAUSE COLLAPSING THAT THIRD VERDICT INTO EITHER OF THE OTHERS IS HOW REPORTS LEARN TO LIE.

Lesson 5 — The scar: the gate that could not fail

PREDICT BEFORE YOU READ: A CI GATE RUNS ON EVERY CHANGE AND PRINTS PASS. ONE DAY SOMEONE DELETES THE ENTIRE ENGINE THE GATE SUPPOSEDLY PROTECTS — EVERY LINE OF IT. WHAT DOES THE GATE PRINT ON ITS NEXT RUN, AND WHAT WOULD YOUR ANSWER IMPLY ABOUT WHAT THE GATE WAS EVER MEASURING?

If your instinct says "FAIL, obviously — the engine is gone," hold that instinct up against the true story below. It happened here, in this repo, and the code that replaced the broken gate wrote the confession into its own docstring so nobody could forget it.

A teacher who never meets the student

The SDTM Mapper keeps a folder of golden fixtures at `/root/sdtm_mapper/golden/` — five domains (AE, DM, DS, EX, VS), each holding a raw input file, the committed mapper program, and a hand-verified expected output. The answer key, in exam terms.

The original CI gate, `/root/sdtm_mapper/scripts/ci_gate.py`, is 36 lines long, and it does something that sounds completely reasonable: on every change, it takes each golden expected output and runs it through the conformance validator (`/root/sdtm_mapper/services/validator.py` — Lesson 4's kind of engine, ten structural rules). If every golden file scores Pass or at least 95%, the gate prints green. Its captured output is still on this box, in `/root/sdtm_mapper/evidence/step4_ci_gate.txt`:

```
CI GATE: PASS (4/4 golden domains conformant >= 95.0%)
```

Read the design again, slowly, and find the hole before I name it. The gate validates… the expected outputs. The hand-verified, frozen, committed answer files. It never runs the mapper. It never reads the mapper. The engine — the actual product, the thing that turns raw data into SDTM — is not in the loop at all.

So: delete the engine. Delete every service file, empty the whole `services` folder except the validator. Run the gate. **It passes.** The golden files are still sitting in their folder, still as conformant as the day a human verified them, and conformant files run through a conformance checker come back conformant. Forever. The replacement gate's docstring — `/root/sdtm_mapper/scripts/parity_gate.py`, quoted verbatim, and this is the retraction I promised you:

```
scripts/ci_gate.py reads the golden OUTPUT files and validates they are conformant.
Static conformant files are always conformant - it passes with the engine deleted.
It tests the answer key, not the student.
```

It tests the answer key, not the student. A teacher grading the answer key sheet every morning and posting “class average: 100%.”

Be fair to the little script for one paragraph, because the failure is subtler than “someone wrote a dumb gate.” Every individual piece is defensible. Validating golden files does catch something real: a corrupted fixture, or a regression in the validator itself. The script is fast, free, offline. The hole is in the claim — the word PASS at the end, read by humans as “the product works,” when what was measured was “the answer key is still the answer key.” A true green, read as evidence for a claim outside its scope. That is the anatomy of every false green worth fearing: not fabricated data, but a real measurement wearing a bigger claim than it earned.

And this project had already met the cruder version of the same disease. The anatomy book records — from the docstring of the mapper’s own IQ gate script — an incident where a worker fleet wrote formal IQ/OQ/PQ validation documents with every check hardcoded `pass`, for an app that had never been run end to end. Evidence claimed, nothing executed. Between the hardcoded PASS and the gate that grades the answer key, the project distilled one rule and stamped it into multiple docstrings as a house invariant: **only a run writes evidence; a gate that cannot fail is not a gate.**

What real evidence looks like

If a claim of PASS is only worth what was executed to produce it, then the artifact recording the claim has to prove an execution happened. Look at what the fixed gate leaves behind. The file `/root/sdtm_mapper/validation_package/PQ_parity_evidence.json` sits on this box right now; here are real lines from it (read this session):

```
"phase": "PQ-parity",
"pass": true,
"executed": true,
"timestamp": "2026-07-28T01:30:33.381181+00:00",
"git_sha": "3f312c2",
```

and, per run, records like this one for the AE domain’s Python arm:

```
{"domain": "AE", "lang": "python", "result": "pass", "secs": 0.5,
 "produced_sha256": "ea2de434...", "golden_sha256": "ea2de434..."}
```

Every field is a receipt. `executed: true` plus a timestamp says when something ran. `git_sha` says which code ran. `secs` says it took wall-clock time — deleted engines don’t take 0.5 seconds. And the two SHA-256 fingerprints say the produced file and the golden file were both actually read and hashed — matching hashes here mean the executed program reproduced the answer key byte-for-byte, this time, on this machine. Compare that to the old gate’s entire evidence trail: a line of captured console text. (The mapper’s `evidence/` folder is full of such `.txt` captures; the structured, dated JSON above is what replaced them as the standard.) None of this makes fabrication impossible — a file is just a file — but it raises the cost from “type the word PASS” to “forge timestamps, hashes and a git sha consistently,” and it gives an auditor something to check.

The design that closed the hole

The fix was not to patch `ci_gate.py`. It was to build a gate with a different shape: `/root/sdtm_mapper/scripts/parity_gate.py` copies each golden domain’s raw input and its committed mapper program

into a fresh temporary directory, **executes the program** as a subprocess with a timeout, and compares the file the program just produced against the golden expected output, cell by cell. Crash? Fail, with the return code. No output file produced? Fail, by name. One cell different? Fail, with the row, column, produced value and expected value listed. Delete the engine now and the gate has nothing to execute — it fails loudly, which is exactly the sentence the old gate could never say. The docstring states the standard it holds its own evidence file to: it "can and will say fail," because an evidence file that cannot say fail is not evidence.

The same philosophy runs through the sister repo's per-run gates in `/root/sdtm_review/services/gates.py`, whose docstring adds the final discipline — every gate must have a **negative control**: a test (in `/root/sdtm_review/tests/test_gates.py`) that feeds the gate deliberately broken data and asserts the gate goes red. Think about what a negative control is really for. It is not testing the data; it is testing the gate. A fire alarm you have never heard ring is not known to work. A gate that has never failed is in exactly the same epistemic position — maybe it's vigilant, maybe it's a painted-on smoke detector, and from the green light alone you cannot tell which.

Try it yourself — the whole scar in twenty lines

A static gate and an executing gate, side by side, with the engine deleted — plus a dated evidence file written from what actually ran:

```
import datetime, json

GOLDEN = "USUBJID,AESEV\nSYNTH-001-101001,MILD\n"   # answer key, saved long ago

def engine(deleted=False):
    if deleted:
        raise RuntimeError("engine deleted")
    return "USUBJID,AESEV\nSYNTH-001-101001,MILD\n"

def static_gate():                # ci_gate-shaped: grades the saved answer key
    return GOLDEN.startswith("USUBJID,AESEV")

def executing_gate():             # parity_gate-shaped: runs the student first
    return engine(deleted=True) == GOLDEN

print("static gate, engine deleted    :", "PASS" if static_gate() else "FAIL")
try:
    verdict = executing_gate()
    print("executing gate, engine deleted :", "PASS" if verdict else "FAIL")
except RuntimeError as e:
    verdict = False
    print("executing gate, engine deleted : FAIL --", e)

evidence = {"gate": "executing-demo", "executed": True, "pass": bool(verdict),
            "timestamp": datetime.datetime.now(datetime.timezone.utc).isoformat()}
with open("demo_evidence.json", "w") as fh:
    json.dump(evidence, fh, indent=2)
print(json.dumps(evidence, indent=2))
```

Output:

```
static gate, engine deleted    : PASS
executing gate, engine deleted : FAIL -- engine deleted
```

One import chain away from the product, the static gate is serenely green over a smoking crater. The executing gate tells the truth, and the evidence file it writes carries `"executed": true`, a verdict that can be false, and a timestamp — the three fields that separate evidence from decoration.

Break it — plant the trap, then build the alarm that catches it

Now the exercise that turns this lesson into a reflex. Take Lesson 4's mini validator and sabotage it with the two-line trap — a gate that returns pass without running anything:

```
def honest_gate(rows):
    bad = [r for r in rows if r["AESEV"] not in {"MILD", "MODERATE", "SEVERE"}]
    return {"status": "fail" if bad else "pass", "n_bad": len(bad)}

def trapped_gate(rows):
    return {"status": "pass", "n_bad": 0}     # the trap: never looks at rows

KNOWN_BAD = [{"AESEV": "Grade 3"}, {"AESEV": ""}]   # planted defects

for name, gate in (("honest_gate", honest_gate), ("trapped_gate", trapped_gate)):
    verdict = gate(KNOWN_BAD)["status"]
    ok = ("gate can fail -- trustworthy" if verdict == "fail"
          else "GATE IS BROKEN: passed planted-bad data")
    print(f"{name:13} on known-bad data -> {verdict:4}  ({ok})")
```

Output:

```
honest_gate   on known-bad data -> fail  (gate can fail -- trustworthy)
trapped_gate  on known-bad data -> pass  (GATE IS BROKEN: passed planted-bad data)
```

The loop at the bottom is the five-line self-test — the negative control. Read what it does: it feeds the gate data with defects planted on purpose, and its assertion is inverted from every test you've written so far — it demands the gate **fail**. Passing known-bad data is the one thing an honest gate can never do, so a green verdict here convicts the gate itself. Notice what makes the trap frightening: from the outside, on good data, `trapped_gate` and `honest_gate` are indistinguishable — both print pass, both look diligent, and only bad data can tell them apart. That is why the negative control is not optional garnish; on a gate with no negative control, "always green" and "broken green" are the same observation.

Now un-trap the gate (restore the real check) and rerun: the negative control goes quiet, and — only now — its passing verdicts on real data mean something.

Quiz 5 (answers below)

1. `ci_gate.py` never printed a false statement — the golden files really were conformant every time it checked. So where, exactly, was the lie?
2. An evidence file reads `{"pass": true}`. Name three fields from `PQ_parity_evidence.json` that would raise your confidence an execution actually produced it, and say what each one proves.
3. You inherit a deployment gate that has printed PASS every day for a year. Your teammate says that's reassuring. Using this lesson's vocabulary, what is the one experiment that would tell you whether it's reassuring or worthless — and what result would you demand?

TAKEAWAY: A GREEN LIGHT ONLY MEANS SOMETHING IF THE MACHINE BEHIND IT EXECUTED THE THING IT VOUCHES FOR AND COULD HAVE GONE RED — SO EVERY GATE NEEDS A RUN BEHIND ITS EVIDENCE AND A NEGATIVE CONTROL THAT PROVES IT CAN FAIL.

Answers to Part 2 quizzes

Quiz 4

1. **Act without searching.** The finding names the rule that fired, the exact variable (`AEDECOD`), and the count — the reader can open the dataset at the named column and fix the one thing described. The bare `False` says only "something, somewhere, in 40 columns" and forces the reader to re-derive everything the validator already knew and threw away.
2. It is defensible **only** as "89 passed of the 101 rules that actually evaluated" — a pass rate whose denominator excludes the unevaluated. To stay honest it must sit next to the third number: "177 rules could not run (execution_error)." Shown alone, "88%" invites the reading "88% of everything is fine," when 177 of 278 rules never examined the data at all. The count of what could not be measured is part of the measurement.
3. Any correctness failure survives: severities shifted onto the wrong rows, the wrong source column mapped into `AETERM`, every date valid ISO but from the wrong visit. All cells populated, legal, well-formatted — conformance clean. Only a correctness check catches it: comparing produced values against a known-correct answer (the cell-by-cell oracle grading that produced the 0-of-11 baseline), or a human who knows what the data should say. Conformance checks the container; correctness checks the contents.

Quiz 5

1. Not in any sentence the gate printed — in the **claim its green light carried**. It measured "the frozen answer-key files are conformant" (true every time) but was read, and positioned in CI, as "the mapping engine works." The measurement was real; its scope was not what everyone believed. A gate lies not only by printing false words but by letting a true word stand for a claim it never tested — which is why deleting the engine, the strongest possible product failure, could not move the verdict.
2. Any three of: `executed: true` — an explicit assertion that a run happened, distinguishing this file from a document typed by hand or model; `timestamp` — pins the run to a moment, so evidence can be checked against deploy history and can go stale visibly; `git_sha` — names exactly which version of the code was executed, so the claim attaches to code you can check out; `secs` per run — wall-clock duration, which a nothing-executed fabrication has no reason to have; the `produced_sha256` / `golden_sha256` pair — fingerprints proving both files were actually read and compared, and letting an auditor re-hash the goldens today.
3. Run the **negative control**: feed the gate input that is broken on purpose — known-bad data, or the lesson's stronger version, remove the thing the gate supposedly protects — and demand a **red** result. If it goes red, the year of green was surveillance and is reassuring. If it stays green on planted defects, the gate cannot fail, the year of green was decoration, and "always green" was indistinguishable from "broken green" all along.

Part 3 — Practice: interrogate, build, adapt

You know the three kinds of validation rules and who publishes them (Part 1), and you have stood inside a real gate — watched one catch a broken row by name, and watched another stay green with the engine deleted (Part 2). Part 3 makes it a trade: first you learn to interrogate findings reports with an AI that is never allowed the last word, then you rebuild the gate from a spec in thirty lines, and finally you learn to reshape the same discipline for three very different clients.

Lesson 6 — Interrogating findings with AI

PREDICT BEFORE YOU READ: YOU PASTE A VALIDATOR FINDINGS REPORT INTO AN AI CHAT AND ASK "WHICH OF THESE MATTER?" IT ANSWERS INSTANTLY, CONFIDENTLY, AND IN A BEAUTIFUL TABLE. WHAT IS THE ONE INPUT IT NEEDED TO ANSWER HONESTLY — AND DID YOU GIVE IT?

Hold that. The working relationship this lesson installs is the one Part 2 earned the hard way: the AI proposes, and only a run — or a documented fact from a run — disposes. An AI that triages findings without seeing the dataset's facts is the hardcoded-PASS fleet from Part 2's Lesson 5: conclusions no execution ever backed.

Each prompt below is complete. Paste it into any chat AI exactly as written — the findings, rule facts, and dataset facts (where allowed) are already inside. All data is synthetic, from imaginary study SYNTH-001.

Prompt (a) — THE TUTOR PROMPT

Use this when you want to be taught, not served.

```
You are a patient clinical-data tutor. Teach me ONE lesson and nothing
more: how to read a conformance-validator findings report — what each
finding is, who published the rule behind it, and what a sponsor is
expected to DO about it.

Anchor everything to this synthetic findings report (study SYNTH-001,
AE dataset) and to no other examples:

  SD0002 | Error   | NULL value in variable marked as Required | AEDECOD | 3 records
  SD0009 | Error   | No qualifiers set to 'Y', when AE is Serious | 1 record
  CT2001 | Error   | Variable value not found in non-extensible codelist | AESEV | 2 records

Ground facts you must treat as given (do not substitute your own memory):
- FDA's Technical Conformance Guide lists three types of validation rules:
  CDISC conformance rules, FDA eCTD Technical Rejection Criteria, and
  FDA Business/Validator rules.
- SD0002 and SD0009 are published by CDISC (Publisher IDs CG0014/CG0208
  and CG0042); CT2001 is an FDA rule (FDAB017).
- "Error" is the validation tool's severity word, not FDA's — FDA dropped
  severity from its Validator Rules; CDISC publishes none.
- The sponsor's obligation is: correct the discrepancy, or explain it in
  the Study Data Reviewer's Guide (SDRG).

Structure of the session:
1. Explain the lesson in plain words using only the three findings above.
2. Quiz me with 3 questions, ONE at a time. Wait for my answer each time.
3. If my answer is vague ("that one's bad", "we'd clean it up", "it's a
   warning-type thing"), do not accept it. Make me commit to the exact
   rule ID, its publisher, and what "Error" does and does not mean for
   that finding — then tell me if I am right.
4. Finish with one exercise I can do with this report and a pencil,
   plus its worked answer.
```

Why it is built this way: the ground-facts block pins the tutor to sourced facts instead of model memory. The severity fact is in there because “Error” feels like a regulatory category and is actually a tool-side label (Part 1’s severity lesson); a tutor allowed to improvise will teach the feeling. And rule 3 — drill vague answers to exact rule IDs — is the student-side negative control: an answer that cannot be wrong (“we’d clean it up”) is not an answer, just as a gate that cannot fail is not a gate.

Prompt (b) — FINDINGS TRIAGE, done right

This is the intern’s work order. The annotations after the block map every design choice back to a lesson.

```
You are triaging a conformance-validator findings report for study
SYNTH-001. You PROPOSE dispositions; a human with the dataset decides.
Work ONLY from the findings, rule facts, and dataset facts pasted below.

THE FINDINGS REPORT (AE domain unless stated):
  F1. SD0002 | Error | NULL value in variable marked as Required | AEDECOD | 3 records
  F2. SD0009 | Error | No qualifiers set to 'Y', when AE is Serious | 1 record
  F3. CT2001 | Error | Variable value not found in non-extensible codelist | AESEV | 2 records
  F4. Study-data check: no TS dataset found in the submission package.

RULE FACTS (treat as given):
- SD0002 (CDISC CG0014/CG0208): Required variables cannot be NULL.
- SD0009 (CDISC CG0042): when AESER='Y', at least one seriousness
  criterion variable (AESCAN, AESCONG, AESDISAB, AESDTH, AESHOSP,
  AESLIFE, AESMIE) is expected to be 'Y'.
- CT2001 (FDA FDAB017): values must come from the CDISC codelist; new
  terms cannot be added to a non-extensible codelist.
- A missing TS dataset falls under FDA's eCTD Technical Rejection
  Criteria (eCTD validation error 1734): an automated inbound check
  that can reject the submission before any reviewer sees it.
- For everything that is not rejection-class, the sponsor either
  corrects it or explains it in the SDRG.

DATASET FACTS (everything known about SYNTH-001; if a finding's facts
are not here, they were NOT supplied):
- The package sent for validation contained ae.xpt and dm.xpt only.
- The 3 records with blank AEDECOD are adverse events still ongoing at
  the data cut; MedDRA coding for them happens in the next scheduled
  coding cycle.
- The single AESER='Y' record has AESDTH='Y' in the dataset.

Output ONE markdown table, one row per finding, columns exactly:
  finding | disposition (TRC-reject-class / explain-in-SDRG /
  suspected-false-positive / CANNOT VERIFY) | one-line SDRG explanation
  (only where the disposition is explain-in-SDRG) | which pasted fact
  decided it

Rules:
- CANNOT VERIFY is your verdict whenever the dataset facts above do not
  decide the finding. It is the default, not the last resort. Never
  substitute plausibility for a fact.
- Do not soften TRC-reject-class: it means the submission can bounce
  automatically, whatever severity word the tool printed.
- suspected-false-positive still requires naming the fact that
  contradicts the finding, and a recommended next step (re-run,
  engine/version check) — never silent dismissal.
```

The design choices, one line each:

- **Dispositions, not severities** — Part 1: Error/Warning/Notice is the tool's vocabulary; the sponsor's real categories are the TCG's — rejection-class, correct, or explain in the SDRG. The table forces the regulatory categories over the tool's.
- **TRC as its own class** — Part 1's rejection lesson: the Technical Rejection Criteria are an automated gate at the door; F4 must never sit in the same bucket as a codelist typo.
- **A dataset-facts block, closed** — Part 2's "only a run writes evidence": the AI may only dispose of what a fact from the data decides, and the block states that anything absent was not supplied, so silence can't be read as license.
- **CANNOT VERIFY as default** — the human-facing twin of Part 2's `execution_error`: "I could not look" kept distinct from "I looked and it's fine." F3 has no fact about the two AESEV values, so the honest verdict is CANNOT VERIFY, not a guess dressed as triage.

- **False positives get a next step** — F2's fact (AESDTH='Y' is present) contradicts the finding, making it a statement about the checker — wrong rule version, stale engine — and checker problems get investigated, not shrugged off.
- **"You PROPOSE; a human with the dataset decides"** — the Part 2 spine in one line: the table is a proposal until an execution or a person verifies it.

Run it. The defensible answer: F4 → TRC-reject-class; F1 → explain-in-SDRG (ongoing events, coding cycle named — the classic correct-or-explain case); F2 → suspected-false-positive (the criterion the rule demands is populated; check engine and rule version); F3 → CANNOT VERIFY. An AI that triages F3 anyway has failed the interrogation — and now you know it will fail quietly elsewhere.

Prompt (c) — THE REFUTATION PROMPT

The mirror image: hand the AI a victory claim and pay it to attack. The pasted materials contain a planted lie — that is the point.

```
You are a hostile validation auditor. Below is a claim, the gate log
behind it, and a file listing. Your job is to REFUTE the claim if it
can be refuted, using ONLY what is pasted here.

THE CLAIM: "The validation run passed — zero findings. ae.xpt is
submission-ready. We're done."

THE GATE LOG (verbatim):
  GATE: ae_conformance_check
  VERDICT: PASS (0 findings)
  timestamp: 2026-07-01T09:14:22+00:00
  note: validated golden reference outputs in /golden/ - all conformant

THE FILE LISTING (from the submission folder, today):
  ae.xpt          modified 2026-07-03 16:41
  dm.xpt          modified 2026-07-01 08:02
  ae_gate_evidence.json   NOT FOUND

Attack the claim with, at minimum, these questions, answering each from
the pasted material only:
1. Did the gate EXECUTE anything against ae.xpt, or did it grade static
   reference files? What does its own log say it validated?
2. Where is the evidence file — the artifact with executed-true, the
   input row count, a content hash of what was read, and the code
   version that ran? What does its absence permit?
3. Compare the gate timestamp with the file timestamps. State precisely
   what the PASS can and cannot vouch for.
4. Even if the run were current and real: list what a conformance PASS
   does NOT check, and what the claim "submission-ready" would
   additionally require.
Verdict: SUSTAINED or REFUTED, with the single strongest reason.
```

The trap you planted: the gate's PASS is dated **2026-07-01**, and `ae.xpt` was modified **2026-07-03** — the report predates the file it supposedly vouches for by two days. Whatever that gate validated, it was not this `ae.xpt`. Add the log's own confession ("validated golden reference outputs" — Part 2's answer-key gate, word for word) and the missing evidence file, and the claim is refuted three independent ways. Question 4 is Part 1's closing lesson wearing armor: a conformance PASS says nothing about correctness, nothing about the TRC-level package checks, nothing about the business rules — "compliant" was never "correct," and neither was ever "done."

Exercise — the starvation experiment

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 DATASET FACTS block. Say nothing else.
3. Compare tables. With the facts gone, the only defensible column is CANNOT VERIFY on F1–F3 (F4's fact is gone too — even the reject-class call now rests on the finding's own text). Count how many dispositions the model produced anyway, complete with fluent SDRG prose for facts it was never given.

That count is the lesson. The prose quality did not drop when the facts left the room — only the truth did. This is why the dataset-facts block exists, and why a triage table, like a green light, is only worth what was actually examined to produce it.

Quiz 6 (answers at the end of Part 3)

1. Prompt (b) makes CANNOT VERIFY the default rather than the last resort. Which Part 2 design rule is that copying, and what goes wrong for the two collapse directions if you fold "cannot verify" into "explain" or into "false positive"?
2. In Prompt (c), the strongest single refutation is the timestamp comparison. Which fields of Part 2's evidence anatomy would have made the same fraud detectable even without the file listing, and how?
3. A colleague runs Prompt (b), gets a beautiful table, and pastes the SDRG explanation for F1 straight into the Reviewer's Guide. What step was skipped, and which sentence from this book names the rule?

TAKEAWAY: A FINDINGS REPORT IS A LIST OF CLAIMS, AN AI TRIAGE IS A PROPOSAL ABOUT THOSE CLAIMS, AND NEITHER BECOMES A DISPOSITION UNTIL A FACT FROM THE ACTUAL DATA — SUPPLIED, NAMED, AND CHECKABLE — DECIDES IT.

Lesson 7 — Build the gate: 30 lines that can actually fail

PREDICT BEFORE YOU READ: YOUR GATE IS HANDED AN EMPTY DATAFRAME — ZERO ROWS. EVERY RULE RUNS, EVERY RULE FINDS NOTHING. WHAT SHOULD THE VERDICT BE, AND WHAT DOES YOUR ANSWER SAY ABOUT THE DIFFERENCE BETWEEN "FOUND NO PROBLEMS" AND "LOOKED AT NO DATA"?

In Lesson 6 you kept telling the AI "a run will decide." Time to be the run. The rebuild rule from the CT book applies: read the spec, close the book, write the gate from memory, then compare. What you cannot rebuild, you never owned.

The spec, in plain words

- **Input:** a pandas DataFrame and a list of rules — each rule a `(rule_id, function)` pair, the function taking the DataFrame and returning a list of findings (empty list = clean).
- **Per-rule status:** `pass`, `fail`, or `execution_error` — a rule that raises is caught, filed as `execution_error`, and does not stop the other rules (Part 2, Lesson 4, unchanged).
- **Gate verdict:** `pass` only when at least one row was validated **and** every rule ran **and** no rule failed. Any finding → `fail`. Any rule that couldn't run → the gate refuses to pass, and the refusal is labeled `execution_error`, never `fail` — the checker's problems and the data's problems have different owners.
- **The zero-row rule:** validated zero rows → refuse to pass. A gate that greenlights an empty input has validated nothing and claimed everything — Part 2's answer-key gate, rebuilt smaller.

- **Evidence, always:** every run — pass, fail, or refusal — writes a JSON file carrying: per-rule results, the findings, a UTC timestamp, `executed: true`, the number of rows validated, a SHA-256 content hash of the exact input examined, and a short git-style basis id fingerprinting the gate's own code. The anatomy from Part 2's `PQ_parity_evidence.json`, in miniature: when it ran, what it read, which code did the reading.
- **Exit code:** `0` only on `pass`. Everything else — fail, refusal, error — is nonzero, so a CI pipeline downstream needs no parsing to stop.
- **The negative control is part of the spec, not an afterthought:** the deliverable includes a run on known-bad data that **must** exit nonzero. A gate you have never seen fail is unproven — you cannot tell it from Part 2's trapped gate, and neither can anyone auditing you.

Go write it. Then come back.

The reference implementation

Runs standalone, stdlib plus pandas, synthetic AE-shaped rows inline. The three rules are shaped after the real published checks you met in Part 1: SD0002 (Required variable NULL), CT2001 (value off a non-extensible codelist), SD0009 (serious AE with no seriousness criterion).

```
import datetime, hashlib, json, sys
from pathlib import Path
import pandas as pd

def r_required(df):   # SD0002-shaped: Required variable may not be NULL
    n = int((df["AEDECOD"].str.strip() == "").sum())
    return [{"rule": "SD0002", "var": "AEDECOD", "msg": f"blank on {n} record(s)"}] if n else []

def r_codelist(df):   # CT2001-shaped: value must be on the non-extensible codelist
    bad = sorted(set(df["AESEV"]) - {"MILD", "MODERATE", "SEVERE"})
    return [{"rule": "CT2001", "var": "AESEV", "msg": f"not in codelist: {bad}"}] if bad else []

def r_serious(df):    # SD0009-shaped: AESER='Y' needs a seriousness criterion
    n = int(((df["AESER"] == "Y") & (df["AESDTH"] != "Y")).sum())
    return [{"rule": "SD0009", "var": "AESER", "msg": f"{n} serious AE(s) with no criterion"}] if n
else []

RULES = [("SD0002", r_required), ("CT2001", r_codelist), ("SD0009", r_serious)]

def gate(df, rules, evidence_path):
    results, findings = [], []
    for rid, fn in rules:
        try:
            f = fn(df)
            results.append({"rule": rid, "status": "fail" if f else "pass"})
            findings += f
        except Exception as e:                       # checker problem, NOT a data problem
            results.append({"rule": rid, "status": "execution_error",
                            "msg": f"{type(e).__name__}: {e}"})
    statuses = [r["status"] for r in results]
    if len(df) == 0 or not statuses:
        verdict = "execution_error"                  # validated nothing -> refuse to pass
    elif "execution_error" in statuses:
        verdict = "execution_error"                  # "could not look" is not "looked, fine"
    else:
        verdict = "fail" if "fail" in statuses else "pass"
    evidence = {"gate": "ae_gate", "verdict": verdict, "executed": True,
                "timestamp": datetime.datetime.now(datetime.timezone.utc).isoformat(),
                "n_rows_validated": int(len(df)),
                "input_sha256": hashlib.sha256(df.to_csv(index=False).encode()).hexdigest(),
                "basis": hashlib.sha256(Path(__file__).read_bytes()).hexdigest()[:7],
                "results": results, "findings": findings}
    Path(evidence_path).write_text(json.dumps(evidence, indent=2))
    return 0 if verdict == "pass" else 1

good = pd.DataFrame([{"USUBJID": "SYNTH-001-101001", "AEDECOD": "Headache",
                      "AESEV": "MODERATE", "AESER": "N", "AESDTH": ""}])
bad = pd.DataFrame([{"USUBJID": "SYNTH-001-101002", "AEDECOD": "",
                     "AESEV": "SEVERE!!", "AESER": "Y", "AESDTH": ""}])
empty = good.iloc[0:0]

for name, df in (("good", good), ("known-bad", bad), ("empty", empty)):
    code = gate(df, RULES, "ae_gate_evidence.json")
    v = json.loads(Path("ae_gate_evidence.json").read_text())
    print(f"{name:9} -> exit {code}  verdict={v['verdict']:15} rows={v['n_rows_validated']}")

assert gate(bad, RULES, "ae_gate_evidence.json") == 1     # negative control: MUST fail
assert gate(empty, RULES, "ae_gate_evidence.json") == 1   # zero rows: MUST refuse to pass
print("negative controls passed: the gate can fail")
```

Output on this box:

```
good      -> exit 0  verdict=pass            rows=1
known-bad -> exit 1  verdict=fail            rows=1
empty     -> exit 1  verdict=execution_error rows=0
negative controls passed: the gate can fail
```

Read the `gate` function against the spec, because every line is a lesson wearing a variable name. The `try/except` per rule is Lesson 4's three verdicts. The `len(df) == 0` branch is the zero-row refusal — and note it produces `execution_error`, not `fail`: an empty input is not evidence of bad data, it is the absence of a validation, and the gate says which. The evidence dict is Part 2's anatomy scaled down: `executed`, a timestamp, `n_rows_validated` (the denominator that keeps any later percentage honest), `input_sha256` (which bytes were actually examined), and `basis` (which gate code did the examining — change a rule and the fingerprint changes). The last three lines are the negative controls, inside the deliverable: the demo doesn't just show the gate passing good data, it proves the gate failing bad data and refusing empty data. Delete either `assert` and you have shipped a fire alarm nobody has ever heard ring.

Notice, too, that the good-data run exits 0 and still writes evidence: next month someone will ask "was this actually run?", and the answer must not be scrollback.

Three upgrade katas

Describe each in two sentences before Lesson 8 — the clients there will ask for all three. Don't implement them now.

1. **Rules as data, not code.** Move the three rules into a CSV — rule id, publisher, target variable, check type, parameter (the codelist values, the required-variable list) — and have the gate load and dispatch them. This is the direction the whole field moved: CDISC's CORE project exists precisely because rules published as documents "get interpreted inconsistently across organizations," and the cure is rules as governed, executable data; your CSV is the toy version of CORE's YAML.
2. **Severity classes and the reject-vs-explain split.** Add a class column to each rule — TRC-reject-class versus reviewer-facing — and make the exit code honor it: reject-class findings always block, reviewer-facing findings can be downgraded to "explain in SDRG" only by an explicit, per-finding, human-authored disposition file that the evidence JSON records. Lesson 6's triage table, executable — and the disposition file is itself evidence, so nobody waves a finding through anonymously.
3. **Wire it as a pre-commit or CI step.** The exit code was designed for this: a hook or CI job runs the gate on the changed datasets and a nonzero exit blocks the merge, exactly the shape of `/root/sdtm_mapper/scripts/parity_gate.py` guarding its repo. The institution-grade detail is scheduling the negative control too — a periodic job that feeds the gate planted-bad data and alarms if the gate passes — because Part 2's scar says a gate that is never seen failing is indistinguishable from a broken one.

Quiz 7 (answers at the end of Part 3)

1. The zero-row refusal produces `execution_error`, not `fail`. Defend that choice using Part 2's two collapse directions — what specific lie does each wrong label tell?
2. Your teammate trims the evidence JSON to `{"verdict": "pass"}` to save space. Name the field whose loss makes the Lesson 6 Prompt (c) fraud undetectable, and the field whose loss makes "pass" unfalsifiable about which data passed.
3. The two `assert` lines run the gate on known-bad and empty data and demand exit 1. What are they testing — the data or the gate — and what is the Part 2 name for this pattern?

TAKEAWAY: THE GATE IS THIRTY LINES, AND ITS HARDEST-WON LINES ARE NOT THE CHECKS — THEY ARE THE REFUSAL TO SAY PASS OVER ZERO VALIDATED ROWS, THE REFUSAL TO RELABEL A CHECKER CRASH AS A DATA FAILURE, AND THE EVIDENCE FILE THAT MAKES EVERY VERDICT, INCLUDING GREEN, AUDITABLE LATER.

Lesson 8 — Different clients, same gate discipline

PREDICT BEFORE YOU READ: THREE CLIENTS ASK FOR "VALIDATION HELP." ONE ALREADY RUNS A COMMERCIAL VALIDATOR, ONE IS PANICKING THREE WEEKS BEFORE SUBMISSION, ONE WANTS VALIDATION WIRED INTO THEIR PIPELINE FOREVER. DO THEY NEED THREE DIFFERENT PRODUCTS — OR ONE DISCIPLINE WITH THREE DIFFERENT THICK SPOTS?

Every validation engagement walks the same seven-stage spine you have now built end to end: **(1) PIN THE RULE SET AND ENGINE VERSION, (2) EXECUTE THE RUN — ACTUALLY RUN IT, ON THE ACTUAL DATA, (3) KEEP THE THREE OUTCOMES APART (PASS / FAIL / COULDN'T-LOOK), (4) TRIAGE FINDINGS — REJECT-CLASS, FIX, OR EXPLAIN, (5) WRITE THE SDRG EXPLANATIONS, (6) EMIT EVIDENCE — TIMESTAMPED, HASHED, VERSIONED, (7) PROVE THE GATE CAN FAIL — THE NEGATIVE CONTROL.** Clients differ not in the spine but in which stages are thick — where the money and the risk live — and which are thin or honestly absent. Learn to draw that map before quoting a price.

Client (a) — the CRO that runs Pinnacle 21 and wants a second opinion

A CRO validates every study with P21 and a sponsor has started asking "how do you know the tool is right?" The beginner hears "audit the vendor" and panics. The actual product is a **CROSS-CHECK**: run CDISC's own open-source CORE engine — the Reference Implementation, MIT-licensed, free, installable from PyPI — over the same datasets, and reconcile. The reframe that makes the engagement valuable: engine disagreement is not noise to explain away, it is information you were hired to surface. Inconsistent interpretation of document-published rules is CDISC's own stated reason CORE exists — and a certification program for third-party engines is planned precisely because engines drift apart. A disagreement means a rule ambiguous enough that two serious implementations read it differently — exactly the finding a "second opinion" is for.

Reuse from our gate layer: the wrapper pattern of `/root/sdtm_review/services/core_engine.py` — including its refusal to merge outcome buckets, which matters double here because CORE's own reports distinguish SUCCESS, SKIPPED, ISSUE REPORTED, and EXECUTION ERROR, and a lazy reconciliation that treats SKIPPED as agreement will manufacture false concord. **The one gate:** no cross-check report ships unless the evidence file pins both engines' names, versions, and rule-set versions, and every disagreement row carries a disposition (rule-interpretation difference / version drift / engine defect / data issue one engine missed) — an unpinned comparison is unreproducible, and Part 1 taught that running an outdated engine means results that won't match what the agency sees. **What NOT to build:** your own rules engine. Two mature engines exist; the deliverable is the reconciliation discipline, not a third opinion.

Stage	1 pin rules+engine	2 execute	3 three outcomes	4 triage	5 SDRG	6 evidence	7 negative control
Weight	**thick** (two engines, two versions)	thin (both engines automate it)	**thick** (SKIPPED ≠ agreement)	**thick** (disposition per disagreement)	thin	**thick** (side-by-side, pinned)	thin (engines are tested upstream)

Client (b) — the small biotech, three weeks out, holding a scary findings report

A small biotech's vendor delivered SDTM datasets plus a validator report with hundreds of findings; the submission date is fixed and the team is triaging by adrenaline. What they think they need is a clean report. What they actually need is Lesson 6's Prompt (b) as a professional service: **TRC-CLASS FIRST** — anything in Technical Rejection Criteria territory (the missing-TS-dataset class of problem) gets fixed unconditionally, because those checks run automatically at FDA's door and can bounce the package before a human reviews a single row. Everything else is corrected or explained — the TCG's own instruction — and validation practice literature says it plainly: some findings are acceptable and expected to remain, with the explanation, not the erasure, as the deliverable.

So the deliverable is the **EXPLAINED REPORT**: every finding carrying exactly one disposition — fixed, explained-in-SDRG with the one-line explanation drafted, or suspected-false-positive with the contradicting data fact named. Reuse: the findings shape and severity ladder of `/root/sdtm_review/services/validator.py` (findings, not booleans — the client's report is already findings; keep it that way), plus Lesson 7's kata 2 disposition file. **The one gate:** zero unclassified findings — no finding without a disposition, no "explain" disposition without its SDRG sentence written. **What NOT to build:** a clean report. Chasing zero findings burns the three weeks on cosmetics — and the tool's own literature concedes it never guaranteed completeness anyway. Forty explained findings with a complete SDRG beat a hushed-up report, because the reviewer reads the explanation but the rejection system reads the package.

Stage	1 pin rules+engine	2 execute	3 three outcomes	4 triage	5 SDRG	6 evidence	7 negative control
Weight	thin (their vendor's run, verify version)	thin (re-run once to confirm)	thin	**thick** (TRC first, then fix-or-explain)	**thick** (the actual deliverable)	thick (disposition ledger)	thin

Client (c) — the sponsor building an automated pipeline

A sponsor is standing up an in-house SDTM pipeline and wants validation "built in, not bolted on." This is Lesson 7's kata 3 at institutional scale, and the Part 2 playbook transfers almost unchanged. Gates live in CI: every dataset build runs the conformance gate, a nonzero exit blocks the merge, and — the parity-gate lesson — the gate must execute the pipeline on its fixtures, not grade frozen reference outputs, or the sponsor will one day own a green light over a deleted engine. Evidence files are the audit trail: each run's JSON, retained, so "when did this dataset last pass, under which rules, as which code" is a query, not an archaeology dig.

The piece beginners omit — and the piece that makes this client's system trustworthy for years — is **INSTITUTIONALIZING THE NEGATIVE CONTROL**: a standing CI job that feeds each gate planted-bad fixtures and goes red if any gate passes them. Reuse: the executing-gate shape of `/root/sdtm_mapper/scripts/parity_gate.py`, and the negative-control convention of `/root/sdtm_review/services/gates.py` with its tests in `/root/sdtm_review/tests/test_gates.py` — every gate paired with a test that demands it fail on broken input. **The one gate** (the gate above the gates): no pipeline stage is "validated" unless its evidence file exists, is newer than the data it vouches for — Lesson 6's timestamp fraud, made structurally impossible — and its gate has a currently-passing negative control. **What NOT to build:** dashboards first, or a house severity taxonomy. A dashboard over unproven gates is decoration over decoration; and severity words are tool-side vocabulary anyway — spend the effort on the reject-vs-explain split, which is the classification that has regulatory meaning.

Stage	1 pin rules+engine	2 execute	3 three outcomes	4 triage	5 SDRG	6 evidence	7 negative control
Weight	thick (versions pinned in CI config)	**thick** (gates run the pipeline, not fixtures)	thick	thin (routes to humans)	thin (templated)	**thick** (retained, queryable)	**thick** (scheduled, alarmed)

The client-scoping prompt

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

```
You are my scoping partner for a clinical data VALIDATION engagement.
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 validation spine:
  1 pin the rule set + engine version    2 execute the run
  3 keep three outcomes apart (pass / fail / could-not-look)
  4 triage findings (reject-class / fix / explain)
  5 SDRG explanations                    6 emit evidence
  7 negative control (prove the gate can fail)

Questions you must get answered (add your own as needed):
- Is validation being CROSS-CHECKED, TRIAGED, or INSTITUTIONALIZED?
  (Second opinion vs pre-submission cleanup vs pipeline — this sets
  the thick stages.)
- What engine(s) and versions run today, and who pins them? Would an
  FDA-side run use the same versions?
- Show me one findings report: are outcomes three-state, or is
  "couldn't check" folded into "no findings"?
- What evidence does a passing run leave behind — a dated, hashed
  file, or scrollback? Is any evidence file older than the data it
  vouches for?
- Has any of their gates EVER been seen failing? On what?

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, the evidence artifact, and how the
     gate is proven able to fail;
  c) the first thing to ask the client for (a file, not a meeting —
     ideally their most recent findings report and the evidence file
     behind their last green light).

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 two engines disagree on 14 rules. The CRO's manager asks "so which engine has the bug?" Why is that the wrong first question, and what does the existence of CDISC's CORE project tell you about the right one?
2. Client (b) insists: "just make the report clean before Friday." Give the two-part answer that redirects the engagement, citing what the TCG says sponsors should do with discrepancies and what class of finding genuinely cannot wait.

3. Client (c)'s CI gates have been green for six months and the sponsor calls that proof of quality. Using this part's vocabulary: what one standing job turns that green from "unproven" to "reassuring," and which two evidence-file fields prove each nightly run genuinely ran on current data?

TAKEAWAY: CLIENTS BUY THICK STAGES, NOT NEW SPINES — YOUR SKILL IS DRAWING THE SEVEN-STAGE MAP, NAMING THE ONE GATE THE DELIVERABLE CANNOT SHIP WITHOUT, AND BEING ABLE TO SHOW, NOT ASSERT, THAT THE GATE CAN GO RED.

Answers to Part 3 quizzes

Quiz 6

1. It copies Part 2's three-verdict rule: `execution_error` — "I could not look" — kept distinct from both pass and fail all the way to the report. Fold "cannot verify" into explain-in-SDRG and you draft Reviewer's Guide prose for facts nobody has — fabricated evidence with good manners. Fold it into suspected-false-positive and you teach the team to dismiss findings the data never contradicted. The two collapses lie in opposite directions, exactly like folding `execution_error` into pass or fail.
2. The `timestamp` plus `input_sha256` pair. A timestamp inside a proper evidence file pins the run to a moment that can be compared against the data file's history; the input hash goes further — re-hash today's `ae.xpt` and if it doesn't match the hash in the evidence, the PASS provably belongs to different bytes, no file-listing forensics required. `{"verdict": "PASS"}` alone can be true of some other file forever.
3. The skipped step: verifying the claimed dataset facts against the actual dataset — running or checking the query that shows those 3 AEDECOD blanks really are ongoing uncoded events — before the explanation ships to a regulator. The sentence: **only a run writes evidence.** The AI's SDRG line is a proposal; it becomes an explanation only when a fact from the data has been executed or inspected to back it.

Quiz 7

1. Labeling it `fail` claims the data is defective — but zero rows revealed no defect; the team hunts a data problem that does not exist and learns to distrust the gate. Labeling it `pass` is worse: the gate certifies data it never examined, which is Part 2's answer-key gate reborn — a green light whose scope is empty. `execution_error` says the true sentence: no validation occurred, and the gate refuses to convert "looked at nothing" into either claim about the data.
2. Losing `timestamp` hides the Prompt (c) fraud — with no run date, a stale report and a fresh one are indistinguishable, so a report can quietly predate the data it vouches for. Losing `input_sha256` makes "pass" unfalsifiable about which data passed — nobody can ever re-hash a file and prove the verdict does or does not belong to it. (`n_rows_validated` is the honorable third casualty: without the denominator, "pass" over zero rows reads like "pass" over a million.)
3. They test the **gate**, not the data — the data's defects are planted and known. This is Part 2's **negative control**: the one experiment that separates "always green because vigilant" from "always green because broken." Its assertion is deliberately inverted — it demands failure — because passing known-bad data is the one thing an honest gate can never do.

Quiz 8

1. Wrong first question because disagreement between engines need not mean a bug in either: the rules were published as documents, and CDISC's own motivation for CORE is that document-published rules "get interpreted inconsistently across organizations." The right first question is "which interpretation of this rule does each engine implement, and which do we — and the agency's configuration — intend?" Some of the 14 will be version drift, some ambiguity, and perhaps some a genuine defect — but that is the output of the reconciliation, not its premise.
2. Part one: findings in Technical-Rejection-Criteria territory get fixed unconditionally and first — those checks are automated at the door and can bounce the submission before review, so no explanation can save them. Part two: for the rest, the TCG's instruction is correct or explain in the Reviewer's Guide — and validation practice accepts that some findings legitimately remain, explained. So Friday's deliverable is the fully dispositioned report and drafted SDRG text, not a cosmetically clean report that hides what the reviewer will find anyway.
3. The standing negative-control job: on schedule, feed each gate planted-bad fixtures and alarm if any gate passes them. Six months of green becomes reassuring only once the gates are periodically seen failing on demand. The two fields: `timestamp` (each run pinned to its night, so a stalled job can't coast on an old file) and `input_sha256` (the hash of what was validated matches the current data, so the run provably read tonight's bytes, not last quarter's).

The graduation checklist

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

- ☐ I can name the three types of study-data validation rules from FDA's own taxonomy — CDISC conformance rules, the eCTD Technical Rejection Criteria, and FDA Business/Validator rules — and say which one can bounce a submission automatically.
- ☐ Given SD0002, SD0009, and CT2001, I can say what each checks, who publishes it, and why the "Error" beside it is the tool's word rather than FDA's.
- ☐ I can explain correct-or-explain: what the SDRG is, and why an explained finding can be a legitimate endpoint while an unexplained one never is.
- ☐ I can state the difference between conformance and correctness, and give one example of a dataset that passes every conformance rule while being completely wrong.
- ☐ I can tell the answer-key-gate story in under a minute: what `ci_gate.py` measured, why deleting the engine couldn't turn it red, and the sentence the fix wrote into its docstring.
- ☐ I can list the evidence-anatomy fields — executed, timestamp, code basis, row count, content hash — and say what specific fraud or mistake each one makes detectable.
- ☐ I can write Lesson 6's triage prompt from memory: dispositions over severities, TRC as its own class, a closed dataset-facts block, and CANNOT VERIFY as the default — and name the lesson behind each feature.
- ☐

I can run a refutation pass on a “we’re done” claim: check what was executed, demand the evidence file, compare its timestamp and hash against the data, and list what the gate never checked.

- ☐ I can rebuild the 30-line gate from the spec — three verdicts per rule, zero-row refusal, evidence JSON, honest exit code — and I can describe all three katas, including why rules-as-data is the direction CORE already went.
- ☐ I can scope a validation deliverable for a new client: draw the seven-stage thick/thin map, name the ONE gate it cannot ship without, and prove that gate can fail — by showing its negative control go red, not by promising it would.

Where to go next

- **The anatomy book.** Modules 5 (Validation and conformance) and 8 (Gates and harnesses) of the Codebase Study Pack document 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; Vol 3 covers Define-XML — the promise your validated data ships with.

Colophon

Written by a research agent (FDA/CDISC primary documents, per-fact URLs, unsourceable claims excluded) and three lesson writers grounded in the real gate code, 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.