CODEBASE STUDY PACK · CLINCODER SERIES · VOLUME 1

SDTM Mapper + SDTM Review

Module anatomy of the ClinCoder mapping engine and the study-level review layer · ingest, knowledge, the AI layer, code generation, validation, scoring, export, gates and ops. Written for a senior SAS programmer and a beginner Python programmer at the same time. The tool will change; the modules outlive it.

Prepared for	Bhanoji Duppada
Subject	/root/sdtm_mapper · /root/sdtm_review · VPS2
Chapters	77 chapters across 10 modules
Basis commits	sdtm_mapper 3f312c22a281c525e86f521fd46506891e01a887
sdtm_review	518eb50afdc224885fb3050cb9fded4ce021f5b1
Built	03 August 2026 · synthetic data only

About this book

CODEBASE STUDY PACK — CLINCODER SERIES, VOLUME 1. Subject: the SDTM Mapper (`/root/sdtm_mapper`) and SDTM Review / TA Review (`/root/sdtm_review`) codebases on VPS2.

This book is only true against the tree it was written from. Basis commits:

```
#| skip-check
sdtm_mapper  3f312c22a281c525e86f521fd46506891e01a887
sdtm_review  518eb50afdc224885fb3050cb9fded4ce021f5b1
```

If a documented file changed after these commits, trust the tree, not the book.

THE GOVERNING PREMISE. This tool will change. The web interface will be replaced, the model will be swapped, the endpoints will be renamed. The modules are what survives. Every chapter is written to be lifted: finish it and you should be able to take that module into a different tool, knowing which package it depends on, why that package and not the alternatives, exactly where it fails, and what its next version looks like.

HONESTY RULES BAKED IN. No measured number appears here that was not measured during the writing session, and where correctness is unproven the text says so — "we do not yet measure this" is a legitimate sentence in a technical book. Every module chapter ends with an explicit Limits block. No module is presented as finished.

SAFETY. Every example, value and output in this book comes from the synthetic study (`synthetic_v4`). No sponsor data, no real subject IDs, no live API keys — keys are redacted to their shape only.

TWO NOTES ON THIS EDITION. The running example is the synthetic **AE** domain rather than the EX/VS suggested in the playbook, because the synthetic study ships no EX or VS raw dataset — AE also exists in the mapper's golden set, which lets Module 4 contrast golden and generated programs on the same domain. This edition is the PDF volume only; the companion audiobook described in the playbook is not part of this build.

Module 0 — Orientation

This module is the map you read before entering the territory: two Flask applications on this host — `/root/sdtm_mapper` (the single-domain mapping engine, port 8812) and `/root/sdtm_review` (the study-level review, scoring and packaging layer, port 8813) — that together turn raw clinical CSV files into SDTM datasets with generated, executed, validated programs. Every claim below was checked against the trees as they exist today, 2026-08-03, and every code block was run on this host before being written down. By the end of this module you will know what each tool does end to end, why the dependency list is five libraries long, which fourteen synthetic adverse-event rows will follow you through all ten modules, and the three distinctions that, if blurred, will make the rest of the book unreadable.

M0.1 The two tools — one engine, forked into two products

You ask a colleague "where does the SDTM mapping happen?" and get two answers: "port 8812" and "port 8813". Both are Python processes named `app.py`. Both import a module called `services/llm.py`. Both requirements files open with the same header, byte for byte. It would be entirely reasonable — and entirely wrong — to conclude they are the same program running twice.

WHAT IT DOES. `sdtm_mapper` takes one raw dataset at a time and walks it through a pipeline: profile the columns, classify which SDTM domain it is, write a mapping specification, generate a mapping program in three languages (SAS, R, Python), execute the Python and R versions, and check the result against SDTM conformance rules. `sdtm_review` is a fork of that engine that grew a second storey: it manages whole studies rather than single files, lets human reviewers score and adjudicate what the machine produced, and packages the results (define.xml, bundles, therapeutic-area review packs) for delivery.

WHY IT EXISTS. The mapper answers "can a machine write a correct AE mapping program?" The review tool answers the question that comes next in any real organisation: "who checked it, what did they change, and what do we actually ship?" Without the mapper there is nothing to review; without the review layer the mapper's output is an unaudited artifact nobody can sign off. They were split because the mapper is promoted to a public production URL while the review harness iterates fast in a working tree — the systemd units make this concrete below.

INPUTS AND OUTPUTS. The mapper's unit of work is one file: a raw CSV such as `raw_ae.csv` (78 rows, 12 columns — met properly in M0.3) goes in via `POST /api/profile`, and out come `AE_mapping_spec.xlsx`, `AE_map.py` / `AE_map.R` / `AE_map.sas`, an executed `ae.csv` and `ae.xpt`, and a conformance report — all under `output/<job_id>/`. The review tool's unit of work is a study or a therapeutic area: many raw datasets, a catalogue, a reviewer database (`data/review.db`), and export bundles.

HOW IT WORKS. Both `app.py` files start identically — same docstring, same core imports:

```
from services.engine import profile_dataset, classify_domain, generate_mapping_spec, DOMAINS, KP
from services.codegen import generate_all_code
from services.accuracy_gate import run_with_gate
from services.docgen import write_excel, write_word
```

Then they diverge hard. The mapper (`/root/sdtm_mapper/app.py`) additionally imports `core_validator` and exposes `/api/validate_core`, an authoritative CDISC CORE conformance endpoint. The review app (`/root/sdtm_review/app.py`) instead registers four Flask blueprints, each wrapped in a `try/except` so a broken review layer cannot take the mapper pipeline down with it:

```
from routes_review import bp as _review_bp
app.register_blueprint(_review_bp)
```

The four are `routes_review.py` (whole-run scoring, backed by `services/review.py` and its `init_db()` which creates `data/review.db`), `routes_mapping.py` (per-variable human adjudication), `routes_study.py` (the study browser — the app's first background-job surface), and `routes_ta.py` (therapeutic-area group review, append-only comments, fail-closed writes). None of those four files exists in `/root/sdtm_mapper`. Run the health endpoints and the divergence is measurable: port 8812 reports `"ig": "SDTMIG v3.4 (SDTM v2.0)"` with 7 domains; port 8813 reports `"ig": "SDTMIG v4.0 (SDTM v2.1)"` with 21 domains. Same function name (`health()`), different knowledge, different products. Even the shared-looking `services/llm.py` is not shared: 417 lines in the mapper, 515 in the review fork; `diff` confirms they differ.

The end-to-end journey through either app is a fixed route sequence, and knowing it now makes every later module legible. A browser (or curl) walks one job through five POSTs: `/api/profile` (upload or pick a library file; a 12-character `job_id` is minted and the dataset is profiled and classified), `/api/spec` (an LLM writes the mapping specification; Excel and Word copies land on disk immediately), `/api/code` (the spec becomes three programs — SAS, R, Python — written as `AE_map.sas`, `AE_map.R`, `AE_map.py`), `/api/execute` (the Python program actually runs and materialises `ae.csv` and `ae.xpt`; the R program runs against the same raw input and is compared cell-for-cell to the Python output), and `/api/validate` (deterministic SDTMIG conformance rules score the executed dataset). Each step logs to an append-only audit trail via `services/audit.py` — the codebase's comments tie this to 21 CFR Part 11 traceability, and one comment records that the audit's model field was once hardcoded to a model that was not actually running, "worse than none."

Two design facts inside `/api/execute` deserve orientation-level attention because they define the product's honesty. First, `/api/code` returns every arm labelled `"unverified"` with an explicit note: "code is generated, not yet executed." Generation is not verification — nothing is trusted until it runs. Second, SAS is never executed here (there is no SAS runtime on the box). Instead the response builds a status from the R arm:

```
if rres.get("matches_python") is True:
    sas_status, sas_note = "released", "R arm executed and matched the Python arm; SAS itself was
NOT executed"
```

Two independently generated programs from one spec, agreeing cell-for-cell in two different languages, is what licenses shipping the SAS translation of that same spec — and every response says so out loud rather than letting the SAS file's presence imply it ran. When the arms disagree or the R arm fails, the status stays `"unverified"` with the reason attached. This released/unverified vocabulary recurs through Modules 4–6; it starts here.

For the SAS programmer: the closest analogy is two studies that both started from the same standard macro library and then patched their local copies independently. Nothing synchronises them. A fix applied in one tree does not exist in the other until someone ports it — the review fork's `CLAUDE.md` explicitly records one such deliberate non-port.

THE PACKAGES. Both are Flask apps — Flask is the thin HTTP wrapper (routes, JSON in, JSON out) around what is otherwise plain Python. There is no ORM, no async framework, no task queue; the review app's one background job is a Python thread. Swapping Flask for FastAPI would touch only `app.py` and the four `routes_*.py` files; the `services/` layer never imports Flask.

TRY IT YOURSELF. With no ClinCoder code at all, reproduce the load-bearing pattern both apps share — an in-memory `JOBS` dict keyed by a short random id, with unknown ids rejected as client errors rather than crashing:

```
import uuid
from flask import Flask, request, jsonify

app = Flask(__name__)
JOBS = {}

@app.route("/api/profile", methods=["POST"])
def profile():
    jid = uuid.uuid4().hex[:12]
    JOBS[jid] = {"filename": request.json.get("filename")}
    return jsonify({"job_id": jid})

@app.route("/api/spec", methods=["POST"])
def spec():
    jid = (request.get_json(silent=True) or {}).get("job_id")
    if not jid or jid not in JOBS:
        return jsonify({"error": "unknown or missing job_id"}), 400
    return jsonify({"job_id": jid, "spec_for": JOBS[jid]["filename"]})

c = app.test_client()
jid = c.post("/api/profile", json={"filename": "raw_ae.csv"}).get_json()["job_id"]
assert c.post("/api/spec", json={"job_id": jid}).get_json()["spec_for"] == "raw_ae.csv"
assert c.post("/api/spec", json={"job_id": "nope"}).status_code == 400
print("ok", jid)
```

Run it with `/usr/bin/python3`; it prints `ok` and a 12-character job id. That dict-plus-uuid pattern is the entire session model of both real apps.

LIMITS. `JOBS` is a plain Python dict: restart the service and every in-flight job vanishes, though the artifacts on disk under `output/<job_id>/` survive. There is no cross-process sharing, so neither app can run under a multi-worker server (gunicorn with workers > 1 would give each worker its own `JOBS`). Concurrency within one process is handled narrowly: `_job_lock(jid)` in `app.py` hands out one `threading.Lock` per job id, added after two visitors executing the same fixed demo job raced and one read a half-written CSV — a bug the code comments note was found by a concurrency soak script (`scripts/visitor_soak.py`), not by any sequential test. The fork relationship is undocumented in code — you can only discover which files diverged by diffing, and we did not diff all of them in this session; the 417-vs-515-line `llm.py` gap is measured, the rest of the divergence surface is not. And the R-agreement-licenses-SAS argument, honest as its labelling is, remains an inference: an R program and a Python program agreeing does not prove the untested SAS text is correct, only that the spec they share is executable to the same result twice.

HOW TO IMPROVE IT. A one-page `FORK.md` in each repo listing which files are intentionally shared, which have diverged, and which fixes were deliberately not ported would cost an hour and prevent the single most likely future accident: patching a bug in one tree and believing both are fixed.

LIFT AND REUSE. The thing worth lifting is the shape, not the code: a stateless `services/` core that never imports Flask, driven by a thin route layer, with per-feature blueprints that fail closed and are registered inside `try/except` so one broken feature cannot kill the product. Leave behind the in-memory `JOBS` dict (use a database or at least a file-backed store) and the fork-by-copy relationship itself.

M0.2 The dependencies — five libraries, everything else is ours

Open `requirements.txt` in either repo expecting the usual wall of a modern web project — an ORM, a validation framework, a task queue, an HTTP client, a cloud SDK — and you find eight pinned lines, five of which are direct dependencies. Your first instinct will be that the file is incomplete. It is not. It is the honest inventory, and its shortness is the single most important architectural fact about these tools.

WHAT IT DOES. The requirements file declares what the application imports that Python itself does not provide: `Flask==3.1.3` (the web server), `pandas==3.0.3` (all tabular data work), `openpyxl==3.1.5` (writing Excel mapping specs and reports), `pyreadstat==1.3.5` (writing SAS XPT transport files), and `python-docx==1.2.0` (writing Word specification documents). Three more lines — `Werkzeug==3.1.8`, `Jinja2==3.1.6`, `numpy==2.4.3` — are transitive dependencies pinned explicitly so a security audit of this one file reflects what actually gets installed: Werkzeug is the HTTP machinery under Flask (historically where Flask-stack CVEs land), Jinja2 is Flask's templating engine (barely used here — the UI is static files — but installed regardless), and numpy is the numeric engine under pandas. The two repos' requirements files are byte-identical (verified with `diff` this session).

WHY IT EXISTS. Everything the five libraries do not cover — SDTM domain knowledge, mapping-spec generation, LLM calls, code generation, program execution, conformance checking, review scoring — is first-party code in `services/`. That is a deliberate trade: a reader who wants to understand how a mapping spec is generated reads `services/engine.py`, not the documentation of a framework. The file's own header comment states the scope precisely: it is "packages app.py / services/ / scripts/ actually import", audited by `scripts/sec_scan.py`, and "deliberately NOT a freeze of the host interpreter." Notably absent: there is no `requests`, no `anthropic`, no `openai` — the LLM layer in `services/llm.py` talks to model backends without adding a vendor SDK to this list.

INPUTS AND OUTPUTS. Each library owns one artifact type in the AE pipeline. pandas reads `raw_ae.csv` and materialises the `ae` DataFrame; pyreadstat writes it as `ae.xpt` (SAS transport version 5, the regulatory submission format); openpyxl writes `AE_mapping_spec.xlsx` and `AE_conformance_report.xlsx`; python-docx writes `AE_mapping_spec.docx`; Flask moves the JSON between browser and pipeline. All five of those exact filenames exist today in `/root/sdtm_mapper/golden/AE/`.

HOW IT WORKS. The division of labour is visible in one line of the mapper's import block — `services/docgen.py` owns both office formats:

```
from services.docgen import write_excel, write_word
```

and in `/api/spec`, where both are produced side by side from the same spec result:

```
xlsx = write_excel(result, job["profile"], job["classification"], os.path.join(d, f"{domain}
_mapping_spec.xlsx"), study_id)
docx = write_word(result, job["profile"], job["classification"], os.path.join(d, f"{domain}
_mapping_spec.docx"), study_id)
```

For the SAS programmer, the bridge and the trap in one sentence each. Bridge: pyreadstat is your `PROC COPY` to XPORT — it exists precisely because regulators still want v5 transport files, and `file_format_version=5` appears in the generated programs for that reason. Trap: pandas is not a DATA step. A DATA step is an implicit row loop with automatic type coercion and a PDV; pandas operations are whole-column expressions, and its missing-value semantics differ so much from SAS that the codebase's standard defensive read is `pd.read_csv(path, dtype=str, keep_default_na=False)` — read

everything as text, convert deliberately — a rule you will see enforced in the repair prompt of `services/executor.py` in M4.

THE PACKAGES. Why these five and not alternatives: Flask over Django (no database-backed admin needed, no ORM wanted), pandas over polars (the LLM-generated programs must run anywhere, and pandas is the dialect every model knows best), openpyxl over xlsxwriter (openpyxl also reads xlsx, used elsewhere in the review tooling), pyreadstat over the abandoned `xport` package (actively maintained C library, reads and writes), python-docx because it is effectively the only maintained Word writer. Swap costs: replacing pandas would mean regenerating and re-verifying every generated program, not just editing library calls — the library choice is baked into the prompts, which is a coupling no SAS macro system prepares you for.

TRY IT YOURSELF. The whole non-Flask stack in eighteen lines — DataFrame to XPT and back, plus both office formats — zero ClinCoder imports:

```
import tempfile, os
import pandas as pd, pyreadstat
from openpyxl import Workbook, load_workbook
from docx import Document

d = tempfile.mkdtemp()
ae = pd.DataFrame({"USUBJID": ["SYNTH01-101001"], "AETERM": ["Insomnia"], "AESTDTC":
["2024-04-14"]})
xpt = os.path.join(d, "ae.xpt")
pyreadstat.write_xport(ae, xpt, file_format_version=5, table_name="AE")
back, meta = pyreadstat.read_xport(xpt)
assert back.shape == (1, 3) and meta.table_name == "AE"

wb = Workbook(); ws = wb.active; ws.title = "Mapping Spec"
ws.append(["Variable", "Source", "Rule"])
ws.append(["AETERM", "AETERM_VERBATIM", "direct copy"])
xlsx = os.path.join(d, "AE_mapping_spec.xlsx"); wb.save(xlsx)
assert load_workbook(xlsx)["Mapping Spec"]["A2"].value == "AETERM"

doc = Document()
doc.add_heading("AE Mapping Specification", level=1)
doc.add_paragraph("AETERM  <-  AETERM_VERBATIM (direct copy)")
doc.save(os.path.join(d, "AE_mapping_spec.docx"))
print("xpt, xlsx, docx written to", d)
```

Run with `/usr/bin/python3` (the interpreter the systemd units use); it was executed successfully on this host during the writing of this chapter.

LIMITS. The pins drift from reality in one measured place: `requirements.txt` pins `numpy==2.4.3` but `/usr/bin/python3` on this host has numpy 2.4.5 installed (Flask 3.1.3, pandas 3.0.3, openpyxl 3.1.5, pyreadstat 1.3.5 all match their pins exactly — verified by import this session). The header comment's claim that the pinned versions "carry no known advisory" was true at the time of a past `pip-audit` run; we did not re-run pip-audit in this session, so treat that as dated evidence, not a current guarantee. Also note there is a second Python on this box (`/root/.claude-tools/bin/python3`) that has none of these packages — using the wrong interpreter is the most common "it worked yesterday" failure here.

HOW TO IMPROVE IT. Add the interpreter path to the requirements header, and regenerate the pins from the actual environment (`numpy==2.4.5`) with a fresh pip-audit run so file and host agree again.

LIFT AND REUSE. The reusable idea is the policy: direct dependencies only, every pin justified by a comment, one script (`scripts/sec_scan.py`) that audits exactly this file. Any tool that generates code

for a target library should also lift the harder lesson: your dependency choice lives in your prompts, so version it like code.

M0.3 The running example — seventy-eight synthetic adverse events, followed everywhere

Subject 101001 could not sleep. From 14 April to 27 April 2024 they had moderate insomnia; it resolved, their dose was not changed. That one line of clinical fact, and 77 more like it, is the entire cast of this book. Every module that follows does something to these same rows, so by Module 7 you will know subject 102002's severe upper respiratory infection the way you know a colleague's coffee order.

WHAT IT DOES. The file `/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv` is a raw — pre-SDTM — adverse events dataset: 78 data rows plus a header, 12 columns, as it might arrive from an EDC export. It is entirely synthetic (`synthetic_v4` is the generation label): the subjects, events and dates were fabricated, and no sponsor data appears anywhere in this book. Here are five real rows from the file, transposed for readability:

Column	row 1	row 2	row 4	row 7	row 11
SUBJECT	101001	101001	102002	103003	102005
AE_NUMBER	1	2	1	1	1
AETERM_VERBATIM	Insomnia	Upper respiratory infection	Upper respiratory infection	Fatigue	Diarrhoea
AE_PT	INSOMNIA	UPPER RESPIRATORY TRACT INFECTION	UPPER RESPIRATORY TRACT INFECTION	FATIGUE	DIARRHOEA
AE_SOC	PSYCHIATRIC DISORDERS	INFECTIONS AND INFESTATIONS	INFECTIONS AND INFESTATIONS	GENERAL DISORDERS	GASTROINTESTINAL DISORDERS
AE_START	14-APR-2024	09-MAY-2024	12-JUN-2024	15-MAY-2024	21-MAY-2024
AE_END	27-APR-2024	17-MAY-2024	30-JUN-2024	17-MAY-2024	31-MAY-2024
AE_SEVERITY	Moderate	Mild	Severe	Mild	Severe
AE_SERIOUS	No	No	Yes	No	No
AE_OUTCOME	RECOVERED/ RESOLVED	RECOVERED/ RESOLVED	RECOVERED/ RESOLVED	RECOVERED/ RESOLVED	RECOVERED/ RESOLVED
AE_ACTION	DOSE NOT CHANGED	DOSE NOT CHANGED	DRUG WITHDRAWN	DOSE NOT CHANGED	DOSE NOT CHANGED
AE_ONGOING	No	No	No	No	No

WHY IT EXISTS. A study pack needs one dataset it never abandons, because the pipeline's whole point is transformation: you can only judge whether a step did its job if you know exactly what went in. AE is the right domain for the job — small enough to hold in your head, rich enough to exercise the interesting machinery: verbatim-vs-coded terms (AETERM_VERBATIM vs AE_PT), DD-MON-YYYY

dates that must become ISO 8601, a Yes/No flag that must become CDISC Y/N controlled terminology, and one genuinely serious event (row 4: 102002's severe URI, drug withdrawn) that any competent review should surface.

INPUTS AND OUTPUTS — THE TEN-MODULE ITINERARY. These same rows will be: profiled raw in **M1** (column names, types, distinct values); matched to controlled terminology in **M2** (Moderate → MODERATE, No → N); embedded in an LLM prompt in **M3**; turned into a generated `AE_map.py` and executed into `ae.csv` / `ae.xpt` in **M4**; checked against SDTMIG conformance rules in **M5**; scored by the review harness in **M6**; and exported into spec documents, define.xml and bundles in **M7**. M8 puts them in study context alongside their DM and VS siblings, and M9 audits the whole trail. When a later module shows you a row, it will be one of these.

HOW IT WORKS. Nothing works yet — this is data, not code — but the transformation the whole book builds toward can be stated now. Raw `SUBJECT` 101001 becomes SDTM `USUBJID` (study-prefixed); `AETERM_VERBATIM` maps to `AETERM`; `AE_PT` and `AE_SOC` to `AEDECOD` and `AEBODSYS`; `14-APR-2024` becomes `2024-04-14` in `AESTDTC`. For the SAS programmer: this is the same raw-to-SDTM mapping you have written a hundred times; the novelty is who writes the program (Module 4) and how anyone knows it is right (Module 5). For the Python beginner: SDTM is the FDA-mandated standard shape for clinical trial data, and "mapping" means writing the program that reshapes a sponsor's arbitrary raw export into it.

THE PACKAGES. Only pandas touches this file directly. The load idiom used throughout the codebase deserves its one paragraph of ceremony: `pd.read_csv(path, dtype=str, keep_default_na=False)`. Without those arguments pandas would parse `AE_NUMBER` as integers, and — the sharper edge — silently convert empty strings to `NaN`, which then poisons every string comparison downstream. SAS has one missing value semantic; pandas has `NaN`, `None`, `pd.NA` and the empty string, and they are not interchangeable. Reading everything as text and converting on purpose is the codebase's answer.

TRY IT YOURSELF. The book's core transformation on two of the real rows, standalone:

```
import io
import pandas as pd

raw = io.StringIO(
    "SUBJECT,AETERM_VERBATIM,AE_START,AE_SEVERITY\n"
    "101001,Insomnia,14-APR-2024,Moderate\n"
    "102002,Rash on forearm,25-MAR-2024,Severe\n")
df = pd.read_csv(raw, dtype=str, keep_default_na=False)
ae = pd.DataFrame({
    "STUDYID": "SYNTH01",
    "DOMAIN": "AE",
    "USUBJID": "SYNTH01-" + df["SUBJECT"],
    "AETERM": df["AETERM_VERBATIM"],
    "AESTDTC": pd.to_datetime(df["AE_START"], format="%d-%b-%Y").dt.strftime("%Y-%m-%d"),
    "AESEV": df["AE_SEVERITY"].str.upper(),
})
assert list(ae["AESTDTC"]) == ["2024-04-14", "2024-03-25"]
print(ae.to_string(index=False))
```

Ten lines of pandas do what a DATA step with `input` / `format` statements would do — but note what is absent: no implicit loop, no PDV, no automatic retain. Each column is one whole-vector expression. The date line is the densest bridge for a SAS reader, so unpack it once: `14-APR-2024` is what you would read with the `date9.` informat (plus a hyphen); `format="%d-%b-%Y"` is Python's spelling of that informat, and `.dt.strftime("%Y-%m-%d")` is the `yymmdd10.` format on the way back out. Two SAS instincts fail here. First, there is no numeric date value underneath — `pd.to_datetime` produces a

datetime column, and once `strftime` runs you hold plain text again; if you need date arithmetic later you must keep the datetime column, not the string. Second, an unparseable date does not produce a note in a log and a missing value — it raises an exception and the whole program dies, unless the code passes `errors="coerce"` and then silently manufactures `NaT`. Both behaviours — die loudly or coerce silently — are choices the generated programs must make, and Module 4 shows which one the prompts demand and why.

LIMITS. Synthetic data is clean data. Every date in this file parses; every severity is one of three spellings; every subject has a well-formed id. Real EDC exports have `UNK` dates, partial dates (`--APR-2024`), free-text severities, and duplicated AE_NUMBERs — and this file exercises none of that, so a pipeline that scores perfectly here has proven much less than it appears to. The file also arrives pre-coded (AE_PT/AE_SOC are already MedDRA-shaped), meaning the hardest real-world step — dictionary coding — is already done in the raw data. We do not measure how the pipeline behaves on dirty inputs anywhere in this module.

HOW TO IMPROVE IT. A `synthetic_v5` with deliberate dirt — partial dates, a blank severity, one duplicate key — would let the validation and repair modules (M4, M5) demonstrate their value on inputs that actually need them.

LIFT AND REUSE. Lift the practice, not the file: any pipeline book or test suite should nominate one small, synthetic, legally clean dataset and thread it through every stage, so each stage's diff is legible. Leave behind any temptation to substitute real study data — the quarantine incident in this project's history (real unblinded data discovered mislabelled as mock) is exactly why "synthetic only" is a rule and not a preference.

M0.4 Three distinctions — golden versus generated, map versus exec, repo versus repo

You open `/root/sdtm_review/golden/AE/` and see `AE_map.py`. You open `/root/sdtm_review/output/8f4aae1888c9/` and see `AE_exec.py`. You open `/root/sdtm_mapper/golden/AE/` and see — apparently — the same files again. Three pairs of things that look interchangeable and are not. Every hour of confusion you will ever have with these tools traces back to blurring one of these three lines.

WHAT IT DOES. This chapter is a glossary with teeth. **Distinction 1:** `golden/` holds human-blessed reference artifacts — the programs, specs and datasets that a person examined and declared correct; `output/<job_id>/` holds machine-generated artifacts from past pipeline runs, one directory per job, named by a 12-character id minted by `uuid.uuid4().hex[:12]` in `app.py`. **Distinction 2:** `AE_map.py` is the mapping program the tool writes (the deliverable a customer would receive); `AE_exec.py` is the program the tool runs (the same code, materialised to disk at execution time, possibly rewritten by the repair loop). **Distinction 3:** `/root/sdtm_mapper` and `/root/sdtm_review` are two codebases that share ancestry, not code.

WHY IT EXISTS. Without distinction 1, you would treat a stale machine artifact as a reference — `output/` contains whatever past runs left behind, including failed ones, and nothing in a job directory certifies correctness. Without distinction 2, you would read `AE_exec.py`, find it differs from `AE_map.py`, and conclude the tool is lying about what it delivered — when in fact the exec copy may have been patched by up to `MAX_REPAIRS = 2` rounds of LLM self-repair (`services/executor.py`) while the delivered map file was not. Without distinction 3, you would fix a bug once and believe it fixed twice; the two `CLAUDE.md` files record this exact class of accident.

INPUTS AND OUTPUTS. Concretely, for AE. `/root/sdtm_review/golden/AE/` today contains eleven files: `AE_map.py`, `AE_map.R`, `AE_map.sas`, `AE_exec.py`, `AE_mapping_spec.xlsx`, `AE_mapping_spec.docx`, `AE_conformance_report.xlsx`, `ae.csv`, `ae.xpt`, `ae_raw.csv`, and a `.bak` of the R program. Note that golden keeps both the map and exec forms plus the raw input and the executed outputs — a complete, frozen, end-to-end witness. A live job directory such as `/root/sdtm_review/output/114c11547ed6/` has the same shape (`DM_map.py`, `DM_exec.py`, `dm.csv`, `dm.xpt`, specs) but no blessing attached: it is evidence of what one run did, nothing more.

HOW IT WORKS. The map/exec split is two lines of code apart. In `app.py`'s `/api/code` handler, the generated programs are written as deliverables:

```
for lang, ext in [("sas","sas"),("r","R"),("python","py")]:
    open(os.path.join(d, f"{domain}_map.{ext}"), "w").write(code[lang])
```

Then in `services/executor.py`, `_run_once()` writes the copy it will actually execute:

```
prog = os.path.join(out_dir, f"{domain}_exec.py")
open(prog, "w").write(py_code)
```

Between those two writes sits the repair loop: if execution fails, the traceback is fed back to the model, and the corrected program is what lands in `_exec.py` on the retry. So `*_map.py` is the first draft the customer sees and `*_exec.py` is the last draft that actually ran — usually identical, meaningfully different exactly when something went wrong. For the SAS programmer, the sharpest framing: **a generated program is not a macro expansion.** A macro expands deterministically — same inputs, same text, every time. These programs come from an LLM at temperature 0.0, and this project measured (on glm-5.2, recorded in its memory notes) that even temperature 0.0 does not guarantee identical output across runs. Two runs of `/api/code` on the same spec can produce two textually different, both-correct — or one-correct — programs. That is why the pipeline executes and compares rather than trusting generation, and why `golden/` must be frozen by a human rather than regenerated on demand.

A half-sibling of distinction 1 lives inside `output/` itself: demo-replay jobs. Both apps carry a demo mode (`/api/demo/<domain>/<step>`) that replays pre-computed snapshots from `demo_snapshots/` instantly, with no LLM call — built for live presentations. A demo job is registered in the same `JOBS` dict as a real job, but `/api/execute` refuses it with an explicit 400 ("demo replay jobs have no source dataset and cannot be re-executed") rather than crashing, because a replayed snapshot is not an upload and has nothing to execute. So when you browse `output/` you may be looking at three different kinds of directory — real run, gate run (`_gate_job`), or demo scaffolding — and nothing in the directory name distinguishes them.

The repo distinction has an operational face too, in the systemd units. `sdtm-mapper.service` runs from a promoted release — `WorkingDirectory=/srv/prod/sdtm_mapper/current`, with a comment recording that before 2026-07-17 it ran from `/root/sdtm_mapper`, meaning "every file saved during development was instantly live on a public URL." `sdtm-review.service` still runs directly from `/root/sdtm_review` with `Environment=SDTM_PORT=8813`. Both load secrets from `EnvironmentFile=/etc/clincoder/sdtm_mapper.env` (a root-owned mode-600 file; contents not shown here). So editing `/root/sdtm_mapper` changes nothing live, while editing `/root/sdtm_review` changes production on the next restart — the same edit has different blast radius depending on which tree you are in.

THE PACKAGES. No third-party library implements any of this; it is `os.path.join`, `open().write()`, `uuid` and `subprocess` (wrapped by `services/sandbox.py` in the review fork, which executes generated programs under containment rather than as root). That is the point: artifact discipline is a convention,

and conventions are exactly what a study pack must write down because no import statement will ever surface them.

TRY IT YOURSELF. The generate-then-execute pattern in miniature — write a program to disk, run it as a separate process, read back its artifact:

```
import csv, os, subprocess, sys, tempfile

d = tempfile.mkdtemp()
prog = os.path.join(d, "AE_exec.py")
open(prog, "w").write(
    "import csv\n"
    "rows = [['USUBJID','AETERM'], ['SYNTH01-101001','Insomnia']]\n"
    "csv.writer(open('ae.csv','w',newline='')).writerows(rows)\n"
)
r = subprocess.run([sys.executable, prog], cwd=d, capture_output=True, text=True, timeout=30)
assert r.returncode == 0, r.stderr
out = list(csv.reader(open(os.path.join(d, "ae.csv"))))
assert out[1] == ["SYNTH01-101001", "Insomnia"]
print("generated program ran; artifact:", out[1])
```

Notice what the parent process knows about the child: an exit code, stdout, stderr, and whatever files appeared. That narrow interface is precisely what the real executor gets, and why its repair loop feeds tracebacks — not intentions — back to the model.

LIMITS. Nothing enforces these distinctions mechanically. `golden/` is blessed by convention, not by checksum: anyone (or any script) can overwrite a golden file and no gate notices at write time — this project's global backup rule exists because a `--force` regeneration once destroyed 27 generated programs that git was not tracking. `output/` accumulates forever; there is no retention policy visible in `app.py`. And the two golden/AE directories being currently similar across repos is an observation about today, not an invariant — we did not diff them file-by-file this session, and given the forks' divergence you should assume nothing.

HOW TO IMPROVE IT. A `golden/MANIFEST.json` with per-file SHA-256 hashes, checked by the existing gate scripts, would turn "blessed by convention" into "blessed and verified" for roughly thirty lines of code.

LIFT AND REUSE. The reference/artifact/deliverable triad generalises to any generative pipeline: keep a frozen human-approved witness (`golden/`), keep per-run machine output segregated by job id (`output/<id>/`), and name the executed form differently from the delivered form so post-repair drift is visible in a directory listing. Leave behind the unenforced blessing — take the manifest idea with you.

M0.5 How to read this book — anatomy, audiences, and the map ahead

You are holding a book about a codebase that its own authors expect to change. The tools will gain domains, swap models, maybe migrate frameworks; some file named in this book will eventually not exist. That is not a defect of the book — it is its premise. The modules teach the ideas the files currently embody, so that when the files change, or when you rebuild these capabilities inside a different tool entirely, the ideas travel with you.

WHAT IT DOES. This chapter explains the book's mechanics: what every chapter contains, who it is written for, how to decide what to trust, and where each of the roughly seventy source files will be covered across Modules 1–9.

WHY IT EXISTS. A study pack read out of order, or trusted uncritically, is worse than no study pack — it manufactures confident misunderstanding. Three reading rules prevent that. Rule one: every chapter has the same skeleton, in order: a cold open (a concrete situation, no jargon), **What it does**, **Why it exists**, **Inputs and outputs** (always with an AE example), **How it works** (real function names, real control flow), **The packages**, **Try it yourself** (standalone, runnable, no ClinCoder imports), **Limits**, **How to improve it**, and **Lift and reuse**. Skim by reading only cold opens and Limits; study by running every Try-it block. Rule two: two audiences, one text. If you are the Python beginner, run the snippets and skip the SAS asides. If you are the SAS/R clinical programmer, the SDTM content is never explained to you — but every place where DATA-step thinking will mislead you is flagged inline: pandas is not a DATA step (M0.3), a generated program is not a macro expansion (M0.4), and an LLM call is not a deterministic function even at temperature 0.0 (M0.4, and at length in M3). Rule three: trust the code, not the comments — including this project's own documentation.

INPUTS AND OUTPUTS. Rule three deserves evidence, because this codebase demonstrates drift today, in ways we verified this session. Four concrete instances. (1) `/root/sdtm_review/app.py` opens with the docstring `"""ClinCoder SDTM Mapper — Flask application."""` — it is the review app, not the mapper; the docstring rode along in the fork. (2) `/root/sdtm_review/CLAUDE.md` is headed "SDTM Mapper", names root `/root/sdtm_mapper` and port 8812 — while the service it sits beside actually runs from `/root/sdtm_review` on port 8813. (3) Both repos' `requirements.txt` are byte-identical and both headed "SDTM Mapper". (4) In both `app.py` files, the `/api/auto` handler's docstring promises "auto-escalate to Opus only if it doesn't pass," but the code's escalation default is `STRONG = os.environ.get("SDTM_STRONG_MODEL", "minimax/minimax-m2.5")` in `services/accuracy_gate.py`, and the handler's own allowlist (`ALLOWED = {"deepseek/deepseek-v4-pro", "minimax/minimax-m2.5", "z-ai/glm-5.1", "qwen/qwen3-max"}`) contains no Anthropic model at all. The comment describes a spending policy that was deliberately abolished; the code enforces the new one. When this book and the tree disagree, the tree has changed — reread the tree.

HOW IT WORKS — THE MAP OF MODULES 1–9. Each module owns a stage of the AE rows' journey and the service files that implement it. File names below were checked against the trees today; files marked (review) exist only in `/root/sdtm_review`, (mapper) only in `/root/sdtm_mapper`, otherwise both forks carry a file of that name (not necessarily the same contents — M0.1).

Module	Stage (what happens to the AE rows)	Primary files covered
M1	Raw intake and profiling — the 78 rows become a column profile	`services/engine.py` (`profile_dataset`), `services/datasets.py` (review), `services/study_upload.py` (review)
M2	Domain knowledge and terminology — Moderate becomes MODERATE, the file is classified AE	`services/engine.py` (`classify_domain`, `DOMAINS`, `KP`), `services/knowledge.py` (review), `services/ct.py`, `services/ct_index.py`, `services/ig_reference.py` (all review)
M3	The LLM layer — the profile becomes a prompt, the prompt becomes a spec	`services/llm.py` (both forks — they differ), `services/model_tiers.py` (review), `generate_mapping_spec` in `services/engine.py`, `services/llm_mapper.py` (review)
M4	Code generation and execution — spec to `AE_map.py`, execution to `ae.csv` / `ae.xpt`, repair loop	`services/codegen.py`, `services/executor.py`, `services/sandbox.py` (review)
M5	Validation — the executed dataset meets SDTMIG rules	`services/validator.py`, `services/core_validator.py` (mapper), `services/core_engine.py` (review), `services/spec_enforce.py` (review)
M6	Scoring and human review — runs scored, variables adjudicated, TA group review	`services/review.py`, `services/mapping_review.py`, `services/ta_review.py`, `services/accuracy_gate.py`, `services/parity_harness.py`, `routes_review.py`, `routes_mapping.py`, `routes_ta.py` (all review except `accuracy_gate.py`)
M7	Export and packaging — specs, define.xml, bundles, TA packs	`services/docgen.py`, `services/define_xml.py`, `services/define_pipeline.py`, `services/bundle.py`, `services/dds_export.py`, `services/ta_pack.py`, `services/mapping_plan_export.py` (all review except `docgen.py`)
M8	Study-level orchestration — AE alongside DM, VS and siblings, bulk runs	`services/study_run.py`, `services/study_browser.py`, `services/study_context.py`, `routes_study.py`, `scripts/batch_run.py`, `scripts/fleet_map.py` (all review)
M9		

Module	Stage (what happens to the AE rows)	Primary files covered
	Gates, audit and operations — evidence, security, the CI gates	`services/audit.py`, `services/apikey.py`, `services/watermark.py`, `services/privacy.py`, `services/gates.py` (review), `scripts/ci_gate.py`, `scripts/parity_gate.py`, `scripts/iq_gate.py`, `scripts/oq_gate.py`, `scripts/sec_scan.py`, `scripts/visitor_soak.py`

The count worth internalising: the mapper's `services/` holds 12 Python modules; the review fork's holds 56. The delta — 44 files — is the review product, and Modules 6–8 live almost entirely inside it.

Two suggested paths through Modules 1–9, one per audience. The Python beginner should read in numeric order — each module's Try-it builds on the previous one's vocabulary — but may defer M6 (review workflows) and M8 (study orchestration) to a second pass, since both are organisational machinery on top of a pipeline you need to understand first. The SAS/R clinical programmer can move faster and less linearly: skim M1–M2 (profiling and terminology will feel familiar; read only the pandas-idiom asides), then slow right down for M3–M4, which contain the genuinely new material — what an LLM prompt for a mapping spec actually looks like, why the same prompt can yield different programs, and what a repair loop does that a syntax check cannot. M5 will feel like P21/CORE territory with a different engine; M9 is where the regulated-industry instincts (audit trails, gates, evidence) reappear in Python clothing and is worth reading closely precisely because it will feel familiar and is not quite.

THE PACKAGES. The book itself depends on nothing beyond what M0.2 installed. Every Try-it block in Modules 0–9 runs under `/usr/bin/python3` with the five libraries plus the standard library, against synthetic data only.

TRY IT YOURSELF. Confirm the territory matches this map before reading further — both services, live, from the standard library alone:

```
import json, urllib.request

for port, name in ((8812, "sdtm_mapper"), (8813, "sdtm_review")):
    with urllib.request.urlopen(f"http://127.0.0.1:{port}/api/health", timeout=5) as r:
        h = json.load(r)
    print(name, "port", port, "->", h["ig"], "|", len(h["domains"]), "domains")
```

Run on this host it prints `SDTMIG v3.4 (SDTM v2.0)` with 7 domains for the mapper and `SDTMIG v4.0 (SDTM v2.1)` with 21 domains for the review fork — the two-codebases fact of M0.1, measured in four lines. (If a service is down, `systemctl status sdtm-mapper sdtm-review` is the first look; note the review fork's domain list includes an oddly-named key `"NS--"`, unexplained at orientation level and flagged for M2.)

LIMITS. The module map is a plan, not a contract: Modules 1–9 are written after this one, and the file-to-module assignment may shift as their authors read deeper — treat the table as the intended itinerary. The map also under-represents the `scripts/` directories (17 scripts in the mapper, 37 in the review fork); many are one-shot patch or fix scripts (`patch_batch1.py` … `patch_batch6.py`, `fix_dm_lb_spec.py`) that record history rather than implement the product, and the book will cover them selectively. Finally, this book documents the trees at `/root/sdtm_mapper` and `/root/sdtm_review`; the mapper's

production copy lives at `/srv/prod/sdtm_mapper/current` and can lag or lead the tree — a divergence this book does not track release by release.

HOW TO IMPROVE IT. When Modules 1–9 land, regenerate this table from their actual headings; a stale map in the orientation module would violate its own rule three.

LIFT AND REUSE. The chapter anatomy itself is the reusable artifact: cold open, what/why, real I/O, real control flow, package rationale, runnable standalone snippet, honest limits, next version, extraction notes. Any codebase handover written to that skeleton — with every claim executed before it is asserted — will outlive the codebase it describes, which is the only success criterion a study pack has.

What this module still owes you

- **How anything actually gets mapped.** You have seen the AE rows and the pipeline's shape, but no real profile, spec, prompt, or generated program yet — that is Modules 1–4.
- **A file-by-file diff of the two forks.** We measured that `services/llm.py` differs (417 vs 515 lines) and that the domain knowledge differs (7 vs 21 domains, SDTMIG v3.4 vs v4.0), but the full divergence surface is unmapped.
- **Dirty-data behaviour.** Everything here used clean synthetic rows; how the pipeline handles partial dates, blanks and duplicates is untested in this module — we do not yet measure this.
- **Accuracy numbers.** No correctness percentage appears in this module because none was measured in this session; Module 6 owes you the scoring machinery and the honest way to read its outputs.
- **The `"NS--"` domain key** in the review fork's health response — noted, not explained; owed by Module 2.
- **The production/tree relationship** for the mapper (`/srv/prod/sdtm_mapper/current` vs `/root/sdtm_mapper`) — stated, but the promote/release mechanics are owed by Module 9.
- **The LLM backend cascade.** Both forks route every model call through `services/llm.py`, and the forks' copies differ; which backend actually answers, in what order, and what guards spending is Module 3's opening question — nothing in this module should be taken as an answer to it.

Module 1 — Ingest and study context

This module covers how clinical raw data physically enters the SDTM Mapper / SDTM Review pair on this box: a synthetic study is generated, files in four formats are read into DataFrames, every raw column is censused into an inventory, whole studies are uploaded and catalogued, and the cross-domain facts (reference dates, epochs, visit numbers) are extracted so later mapping steps can use them. The running example is one row of `data/synthetic_v4/raw/raw_ae.csv` — subject `101001`, an Insomnia adverse event starting `14-APR-2024` — which we watch arrive as EDC-shaped CSV, get profiled, get catalogued, and get its study day (`27`) and epoch (`TREATMENT`) computed. Every path, symbol and number below was read or executed on this host in this session; the two repos are `/root/sdtm_mapper` (single-dataset demo app, port 8812) and `/root/sdtm_review` (the batch/fleet fork where all the study-level machinery lives).

M1.1 `/root/sdtm_review/data/synthetic_v4/build_synthetic.py` — invents a study both sides trust

You want to grade a mapping tool, but every real dataset on the box is either unblinded sponsor data you must not touch or public data with no trusted answer key. The only clean way out is to invent a study yourself — and write down the answers before the tool ever sees the questions.

WHAT IT DOES. Generates a complete fake clinical study called SYNTH-001: 60 patients, 11 data domains, every visit, adverse event and questionnaire response fabricated from one random seed. For each domain it writes two files: `raw/raw_<dom>.csv` (messy, the way an EDC system would export it) and `expected/<dom>.csv` (clean SDTM, the answer key). It also writes `MANIFEST.json` describing exactly what is and is not guaranteed.

WHY IT EXISTS. Nine domains (CO, DV, FA, IE, PR, QS, SE, SU, SV) had no input data anywhere on this box, so the mapper could never be tested on them. Without this file there is no oracle: you could run the mapper and eyeball the output, but you could never say "correct" — which is the whole premise of the review harness (correctness = executed data compared cell-for-cell, not plausible-looking SAS text).

INPUTS AND OUTPUTS. Inputs: `knowledge/sdtm_ig_v4.json` (the SDTM IG variable definitions) and the real CDISC controlled-terminology CSV at `/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv`. Outputs: 11 raw CSVs, 11 expected CSVs, one manifest. The AE pair looks like this — same event, two shapes:

```
raw/raw_ae.csv       SUBJECT,AE_NUMBER,AETERM_VERBATIM,...,AE_START,...
                     101001,1,Insomnia,...,14-APR-2024,...
expected/AE.csv      STUDYID,DOMAIN,USUBJID,AESEQ,...,AETERM,...,AESTDTC,AESTDY,EPOCH
                     SYNTH-001,AE,SYNTH-001-101001,...,Insomnia,...,2024-04-14,27,TREATMENT
```

`raw_ae.csv` has 78 data rows and 12 columns (measured this session).

HOW IT WORKS. `main()` loads the IG pack and CT, seeds one `random.Random(SEED)` (`SEED = 20260719`), then calls three functions in order. `build_study(rng, ct)` creates the single in-memory study model: 60 subject dicts with staggered consent dates, an arm, a visit list built from `VISIT_SCHEDULE`, and disposition (one subject, index 23, dies). `_attach_events` hangs AE/IE/DV/PR/SU/QS/CO/FA/SE records off each subject, always bounded by that subject's own date window. Then the split that makes it an oracle: `emit_raw(study)` writes the EDC shape (coded sex `1`/`2`, Title-Case severity, `DD-MMM-YYYY` dates via `edc()`), while `derive_expected(study, ig)` writes the SDTM shape (ISO dates via `iso()`, `--SEQ` numbering, `study_day()` for `--DY`). Two code paths, one model — the docstring calls it THE CIRCULARITY RULE:

```
# This module must NEVER import or invoke ``services.engine``,
# ``services.codegen``, ``services.executor`` or any LLM. If the
# reference came from the thing under test, the oracle would prove nothing.
```

That claim holds: there are no `services` imports in the file.

THE PACKAGES. stdlib only — `csv`, `datetime`, `json`, `os`, `random`. Deliberate: pulling in pandas or numpy would add nothing (the writer is `csv.writer` with `lineterminator="\n"` for byte-stable output) and would make the determinism claim ("two runs are byte-identical", asserted in `tests/test_synthetic.py`) hostage to library version changes. `random.Random(SEED)` as an explicitly passed instance, never the module-level `random.*` functions, is what makes the draw order auditable.

SAS BRIDGE. This is the piece a SAS shop usually doesn't have: you have test data, but rarely test data with a machine-checkable answer key generated independently of the mapping code. The nearest analogy is writing your production DATA step and your QC double-programming from one shared spec — except here the "QC side" (`derive_expected`) is written first and frozen, and the thing being graded is generated code. Where the analogy misleads: `expected/` is not a second independent programmer; it is one deliberate reading of the IG (the manifest's `reading_of_the_ig` section says so), so a disagreement can mean the mapper is wrong or the reading differs.

TRY IT YOURSELF. A 20-line version of the raw/expected split (stdlib only):

```
import csv, datetime as dt, io, random

rng = random.Random(42)                       # fixed seed = byte-identical reruns
subjects = [f"101{n:03d}" for n in range(1, 4)]
model = []                                    # ONE in-memory study model
for subj in subjects:
    start = dt.date(2024, 3, 1) + dt.timedelta(days=rng.randint(0, 30))
    model.append({"subj": subj, "term": "Headache", "start": start})

def edc(d):                                   # raw side: 21-MAR-2024
    return d.strftime("%d-%b-%Y").upper()

raw, sdtm = io.StringIO(), io.StringIO()
csv.writer(raw).writerows([["SUBJECT", "AETERM_VERBATIM", "AE_START"]] +
                          [[m["subj"], m["term"], edc(m["start"])] for m in model])
csv.writer(sdtm).writerows([["USUBJID", "AETERM", "AESTDTC"]] +
                           [[f"SYNTH-{m['subj']}", m["term"],
                             m["start"].isoformat()] for m in model])
print(raw.getvalue())                         # what the mapper receives
print(sdtm.getvalue())                        # what it must produce
```

LIMITS. No clinical realism — AE rates and dropout are arbitrary, the manifest says never to validate an analysis against it. `AEDECOD` / `AEBODSYS` are placeholders, not MedDRA (MedDRA is licensed; do not score dictionary coding here). No SUPPQUAL, no define.xml, no Trial Design domains. The known 39-row EPOCH disagreement between the generator's `_visit_at` rule and the SE-window rule is documented in M1.7. `CT_PATH` is an absolute path outside the repo — regeneration fails on a box without `/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv`.

HOW TO IMPROVE IT. Ship the CT CSV (or a trimmed extract) inside the repo so the corpus regenerates anywhere; add partial-date and duplicate-record injections as labelled difficulty tiers; emit a second header variant per domain so shape-matching (M1.5) has a synthetic negative case.

LIFT AND REUSE. The lift-worthy idea is the pattern, not the study: one seeded model, two emitters, a manifest that states guarantees and non-guarantees explicitly, and a hard rule that the generator may

not import the system under test. Genericise `build_study` / `emit_raw` / `derive_expected` behind "model → (raw_writer, expected_writer)". Leave behind the EQ-5D specifics and the hard-coded CT path; take the `MANIFEST_NOTES` discipline with you.

M1.2 `/root/sdtm_mapper/app.py` — one raw dataset enters, any format

A visitor lands on the demo site, drags a file into the browser, and expects columns and sample values on screen a second later. That file might be a CSV from Excel, a `.sas7bdat` straight out of a SAS library, a `.xpt` transport file, or a spreadsheet — and the server has to read all four without asking questions.

WHAT IT DOES. The Flask app (served by `sdtm-mapper.service` on port 8812) accepts one uploaded dataset at `/api/profile`, saves it to disk, reads it into a pandas DataFrame regardless of format, and returns a column profile — names, fill rates, sample values — plus an LLM-assisted guess at which SDTM domain it is. The actual format handling lives in `profile_dataset()` in `/root/sdtm_mapper/services/engine.py`.

WHY IT EXISTS. This is the front door of the single-dataset Mapper. Without it nothing enters the system at all; without the multi-format branch, every SAS shop's first file (`ae.sas7bdat`) would bounce and the demo would be CSV-only.

INPUTS AND OUTPUTS. Input: a multipart POST with a `file` field, or `{"library": "ae_raw.csv"}` naming a bundled sample. Output: JSON with `job_id`, `profile` and `classification`. Feeding it our running example, the profile of `raw_ae.csv` contains an entry like `{"name": "AE_START", "pct_filled": 100.0, "samples": ["14-APR-2024", ...]}`. The saved file lands **flat** as `data/uploads/<job_id>.<ext>` where `job_id = uuid.uuid4().hex[:12]` — e.g. `data/uploads/ae725509f03f.csv` exists on disk from a past visitor.

HOW IT WORKS. In `api_profile` (app.py):

```
data = request.get_json(silent=True) or {}
if "file" in request.files:
    f = request.files["file"]; jid = uuid.uuid4().hex[:12]
    ext = f.filename.rsplit(".",1)[-1].lower()
    src = os.path.join(UPLOADS, f"{jid}.{ext}"); f.save(src); raw_name = f.filename
```

then `df, profile = profile_dataset(src)`, `classify_domain(...)`, and the job is parked in the in-memory `JOBS` dict keyed by `jid` (a restart forgets every job — files survive, jobs don't). The comment above `data = ...` records a real production bug: `data` used to be bound only in the else-branch, so every file upload — the primary visitor action — raised `UnboundLocalError`. The format dispatch in `services/engine.py`:

```
ext = path.lower().rsplit(".", 1)[-1]
if ext in ("csv", "txt"):
    df = pd.read_csv(path, dtype=str, keep_default_na=False, nrows=5000)
elif ext in ("sas7bdat", "xpt"):
    import pyreadstat
    reader = pyreadstat.read_sas7bdat if ext == "sas7bdat" else pyreadstat.read_xport
    df, meta = reader(path)
    df = df.astype(str)
elif ext in ("xlsx", "xls"):
    df = pd.read_excel(path, dtype=str).fillna("")
```

WHAT METADATA SURVIVES, PER FORMAT. This is the load-bearing table for a SAS reader:

- **CSV** — nothing but names and values. No labels, no formats, no lengths, no types. `dtype=str, keep_default_na=False` keeps blanks as `""` (pandas would otherwise turn them into `NaN` — the equivalent of SAS silently converting `''` to `.`), and `nrows=5000` caps the read.
- **sas7bdat / xpt** — pyreadstat returns `(df, meta)` where `meta` carries `column_names_to_labels`, `original_variable_types` (SAS formats), lengths and encodings. **But the code drops `meta` on the floor:** `df = df.astype(str)` and only the DataFrame goes forward. Labels survive the reader and die at the next line. Also measured this session: `/root/sdtm_review/golden/AE/ae.xpt` has every label `None` — an XPT only carries labels if whoever wrote it set them, and that file was written label-less by this pipeline's own executor.
- **xlsx/xls** — values only; cell formatting, column widths and any second sheet are ignored (`pd.read_excel` reads the first sheet by default).

THE PACKAGES. `pyreadstat` over `pandas.read_sas`, and the difference is measurable: `pd.read_sas` on that same `ae.xpt` returns byte strings (`b'CDISCPILOT01-701-101'`) and **no metadata object at all**, so you would be decoding bytes by hand and labels would be unreachable. pyreadstat (a wrapper over the C library ReadStat) returns clean strings plus the `meta` object, and — used elsewhere in this repo — can also write XPT (`pyreadstat.write_xport`, see `services/executor.py`), which pandas cannot. `openpyxl` is here as pandas' xlsx engine and for writing the spec workbook in `services/docgen.py`. Swapping to `pd.read_sas` would save a dependency and cost you labels, string decoding and XPT writing — a bad trade in a clinical tool.

SAS BRIDGE. `profile_dataset` is `PROC IMPORT` + `PROC CONTENTS` + `PROC FREQ` in one function. But the libname mental model misleads twice. First, there is no persistent library: every read is one file by path, and the "session" (`JOBS`) lives in process memory, not on disk like a `WORK` library. Second, SAS carries labels/formats/lengths as first-class attributes on every dataset forever; pandas has no native slot for them, which is why the `meta` object exists as a separate return value — and why dropping it is a real loss, not a cosmetic one.

TRY IT YOURSELF. (Run with the system `python3` that has pyreadstat; paths exist on this box.)

```
import pandas as pd

# CSV: everything as text, blanks stay "" -- never NaN.
df = pd.read_csv("/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv",
                 dtype=str, keep_default_na=False)
print(df.shape)                          # (78, 12)
print(df.loc[0, ["SUBJECT", "AETERM_VERBATIM", "AE_START"]].to_dict())
# {'SUBJECT': '101001', 'AETERM_VERBATIM': 'Insomnia', 'AE_START': '14-APR-2024'}

# XPT: pyreadstat returns data AND a metadata object.
import pyreadstat
xdf, meta = pyreadstat.read_xport("/root/sdtm_review/golden/AE/ae.xpt")
print(xdf.shape, list(meta.column_names_to_labels.items())[:2])
# (1191, 24) [('STUDYID', None), ('DOMAIN', None)]  <- labels lost at write time
```

LIMITS. `ext` comes from the client's filename, so a mislabelled file hits the wrong reader and 500s into `_err`. `nrows=5000` means the profile of a bigger file silently describes only its head. `JOBS` is unbounded in-memory state. The `meta` discard means domain classification never sees SAS labels — often the single most informative field a `sas7bdat` carries ("Adverse Events" as a dataset label answers the classification question outright). One environment note: on this box the default `python3` (a tooling install, Python 3.14) does **not** have pyreadstat; `/usr/bin/python3` — which the systemd unit runs — has pyreadstat 1.3.5, openpyxl 3.1.5, pandas 3.0.3, flask 3.1.3 (all verified by import this session).

HOW TO IMPROVE IT. Sniff format from magic bytes instead of extension; keep `meta` and pass `column_names_to_labels` into the classifier prompt; persist `JOBS` to sqlite so a restart doesn't orphan uploaded files.

LIFT AND REUSE. Lift `profile_dataset` as a standalone "read anything clinical" function — its interface (`path -> (df, profile_dict)`) is right. Take the `dtype=str, keep_default_na=False` discipline verbatim; it is the single most important pandas idiom in this codebase (blank-vs-NaN bugs are this stack's equivalent of SAS's numeric-missing propagation). Leave behind the Flask job plumbing and the classification call.

M1.3 `/root/sdtm_review/scripts/build_raw_variable_inventory.py` — census every raw column before mapping

Before this script existed, mapping was done one domain at a time against one study, and nobody could answer "what raw variables actually exist across all therapeutic areas, and what SDTM target does each feed?" Every spec was written blind to the others. This script answers the question mechanically, once, in one workbook.

WHAT IT DOES. Reads every raw CSV in the tree — four corpora, hundreds of files — records every column with its real population statistics (fill rate, distinct values, inferred type, sample values), cross-references the SDTM IG v4 knowledge pack for each domain's target variables, and writes one Excel workbook: `evidence/raw_inventory/RAW_VARIABLE_MASTER.xlsx` (present on disk). It suggests candidate mappings from an explicit synonym table but decides nothing.

WHY IT EXISTS. An inventory precedes mapping for the same reason `PROC CONTENTS` precedes a DATA step: you cannot write `AESTDTC = input(AE_START, ...)` until you know `AE_START` exists, how it is formatted, and whether it is even populated. At fleet scale there is a second reason — the inventory's `HEADER_SHAPES` sheet is where "210 raw files collapse to 25 header shapes" is discovered, and that collapse is the unit of work for everything downstream (M1.5, M1.6). Without it you'd map 21 byte-identical Cardiology/Oncology/... AE files 21 times.

INPUTS AND OUTPUTS. Input: the filesystem plus `knowledge/sdtm_ig_v4.json`. Output: the workbook, with sheets `README`, `MASTER_VARIABLES`, `RAW_VAR_INDEX`, `HEADER_SHAPES`, `SDTM_TARGETS`, `MAPPING_WORKSHEET`, `UNMAPPED_RAW`, `COVERAGE`, plus `FLEET_MAPPING` / `FLEET_CONSENSUS` and per-shape `MAP_<domain>_<corpus>` sheets only when a fleet run's JSON exists beside it. Running the real `profile_file` on `raw_ae.csv` this session produced, per column, rows like:

```
raw_var          pos  n_rows  pct  n_distinct  inferred_type    codelist?  samples
SUBJECT            1      78  100.0        39  numeric          N          101001 | 102002 | ...
AETERM_VERBATIM    3      78  100.0        10  text             Y          Insomnia | Diarrhoea
| ...
AE_START           6      78  100.0        72  date_ddmmmyyyy   N          03-JUN-2024 | 24-MAY-2024
| ...
```

`date_ddmmmyyyy` on `AE_START` is the inventory telling the spec writer "this is a `--DTC` candidate that needs reformatting" — exactly the fact the mapper must later act on.

HOW IT WORKS. `main()` → `build(out_path)`. First `discover_sources()` walks four fixed corpus locations under the repo root (`corpus/tierB_robustness`, `corpus/tierA_oracle/*/raw`, `data/synthetic_v4/raw`, `data/raw_library`) and returns one dict per CSV with corpus, study, domain (derived from the filename), path, and — crucially — an `oracle` field stating what that corpus can prove (`"cell-for-cell"` for tier A and SYNTH-001; `"none (robustness only - values NOT gradeable)"` for tier

B). Then `profile_file(source)` reads each CSV with the stdlib `csv` module and emits one dict per column; `infer_type` classifies values against four regexes (`_ISO_DATE`, `_DDMMMYYYY`, `_TIME`, `_NUMERIC`) plus a Y/N set, sampling at most 2000 non-blank values. `candidate_target(raw_var, domain, target_names)` proposes a target three ways, in order: the raw name already is the SDTM name; the normalised name hits the hand-written `SYNONYMS` dict (with `--` expanded to the domain prefix); or the name minus its domain prefix is a target. Everything else lands in `UNMAPPED_RAW` for a human to disposition as SDTM / SUPPQUAL / DROP. `_write_sheet` does the openpyxl styling (bold header, freeze panes, autofilter, width heuristics).

A CODE-VS-COMMENT CONTRADICTION, MEASURED. The docstring of rule 3 reads `e.g. AE_SEVERITY -> AESEV`. It does not do that. `AE_SEVERITY` normalises to `AESEVERITY`; stripping the `AE` prefix leaves `SEVERITY`, which is not an IG variable name (the variable is `AESEV`), so the function returns `(None, '', '')` — verified by calling it this session. Same for `AETERM_VERBATIM` and `AE_START`: all three of our running example's most obvious columns get **no** candidate, because the synonym table keys (`verbatim`, `startdate`, `severity`) don't match the concatenated normalised forms (`aetermverbatim`, `aestart`, `aeseverity`). Only `SUBJECT → USUBJID (medium)` fires. The comment describes an intention; the code implements something narrower. Document the code: prefix-stripping only works when the remainder is a complete SDTM name.

THE PACKAGES. stdlib for all reading (deliberate — the profiling loop is a plain `csv.reader`, no pandas, so a 400-file scan holds one file's columns in memory at a time), openpyxl only at write time, imported inside `_write_sheet` / `build` so `discover_sources` and `profile_file` are importable without it — which is exactly how `study_browser._discover` consumes this script (M1.6). Swapping openpyxl for xlsxwriter would work (write-only here) but openpyxl is already a dependency of the mapper's docgen.

SAS BRIDGE. This workbook is `PROC CONTENTS` + `PROC FREQ` across an entire portfolio, landed in the exact artefact a clinical programmer already uses: a mapping-spec workbook with a `MAPPING_WORKSHEET` to fill in. The misleading part of the analogy: `inferred_type` is not a real type. CSV has no types, so `SUBJECT` — a site-prefixed subject ID — profiles as `numeric` (measured above). In SAS the informat decided; here a regex over string values guesses, and the guess exists to prompt a human, not to drive code.

TRY IT YOURSELF.

```
import csv, re
from collections import Counter

DDMMMYYYY = re.compile(r"^\d{2}-[A-Za-z]{3}-\d{4}$")

def inventory(path):
    with open(path, newline="", encoding="utf-8-sig") as fh:
        rows = list(csv.reader(fh))
    header, body = rows[0], rows[1:]
    out = []
    for i, name in enumerate(header):
        vals = [r[i].strip() for r in body if i < len(r) and r[i].strip()]
        kind = ("date_ddmmmyyyy" if vals and all(DDMMMYYYY.match(v) for v in vals)
                else "text")
        out.append({"raw_var": name, "pct_populated": round(100 * len(vals) / len(body), 1),
                    "n_distinct": len(set(vals)), "inferred_type": kind,
                    "samples": [v for v, _ in Counter(vals).most_common(3)]})
    return out

for row in inventory("/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv")[:4]:
    print(row)
```

LIMITS. CSV only — a corpus delivered as sas7bdat is invisible to it. Domain comes from the filename (`fname.split("_")[0].upper()`), so `adverse_events.csv` would be filed under domain `ADVERSE`. `infer_type` samples the first 2000 non-blank values, so a column that switches format at row 2001 is misclassified. The synonym misses shown above mean `MAPPING_WORKSHEET` candidates are sparse for realistically-named EDC columns — by design it under-claims rather than fuzzy-matches, but the README's framing ("SUGGESTIONS with a stated confidence") is doing heavy lifting. And nothing here is evidence of correctness; the README sheet itself says only an executed, oracle-graded run is that.

HOW TO IMPROVE IT. Route file reading through M1.2's `profile_dataset` so sas7bdat/xpt corpora join the census (and labels become a profiling column); make `candidate_target` also try the synonym table on the prefix-stripped remainder (that one change would catch `AE_SEVERITY` via `severity`); emit the inventory as JSON beside the xlsx so downstream code stops depending on a spreadsheet.

LIFT AND REUSE. `discover_sources` / `profile_file` / `infer_type` / `candidate_target` are already clean, dependency-free functions with dict-in/dict-out interfaces — lift them as-is. The right generic interface is `list[source_dict] -> list[column_profile_dict]`; the workbook writer is presentation and can be left behind. Keep the `oracle` provenance field on every row — carrying "what this data can prove" through every artefact is the best idea in the file.

M1.4 `/root/sdtm_review/services/datasets.py` — one catalogue of every selectable dataset

The review UI has a dropdown: pick a therapeutic area, pick a domain, run. Behind that dropdown sit five unrelated directory trees on two repos — some CSV, some XPT, some with answer keys, most without — and something has to walk them all and present one honest list.

WHAT IT DOES. Enumerates every candidate source dataset across five corpora, normalises each into one flat dict (`id`, `ta`, `domain`, `path`, `format`, `rows`, `tier`, `has_oracle`, `available`), and caches the list to `data/dataset_catalog.json`. It knows nothing about mapping or SDTM semantics — the docstring's claim "filesystem only" is accurate.

WHY IT EXISTS. Without it every consumer would re-implement "what data exists" with its own glob, and the flags that keep the numbers honest — `has_oracle` in particular — would be re-derived (or

forgotten) in each copy. `scripts/review_gate.py`, `app.py` and three test files all import it (grepped this session).

INPUTS AND OUTPUTS. Input: six hard-coded roots — `BENCHMARK_ROOT = "/root/cdisc_benchmark"`, `TIER_A_ROOT = "/root/sdtm_mapper/corpus/tierA_oracle/CDISCPILOT01"`, `TIER_B_ROOT = "/root/sdtm_mapper/corpus/tierB_robustness"`, `ONCO_ROOT = "/root/onco-cdisc-demo/data/sdtm"`, plus `data/raw_library` and `data/synthetic_v4` under the app root. Note the cross-repo reach: tier A/B here point at `/root/sdtm_mapper`'s corpus, while M1.3's `discover_sources` walks `/root/sdtm_review`'s own `corpus/` copy — two scanners, two corpus locations, both directories exist. Output, measured this session via `catalog_summary()`: 375 entries — `tierA 5, synth 11, bench 158, tierB 189, onco 7, raw 5`. The running example's entry:

```
{"id": "synth:SYNTH-001:AE", "ta": "SYNTH-001", "domain": "AE",
 "path": "/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv",
 "format": "csv", "rows": 78, "tier": "synth", "has_oracle": true, "available": true}
```

HOW IT WORKS. `build_catalog(force=False)` takes a `threading.Lock`, returns the in-process cache, else the JSON cache, else runs the six scanners in `_SCANNERS` order (`_scan_tier_a`, `_scan_synth`, `_scan_benchmark`, `_scan_tier_b`, `_scan_onco`, `_scan_raw_library`), de-duplicates by id, sorts by `(tier, ta, domain)` and writes the cache atomically (`os.replace` of a `.tmp` file; an `OSError` is swallowed so a read-only data dir degrades to scan-per-process rather than crashing). `_collect` picks exactly one file per domain when several formats exist, ranked by `FORMAT_PREFERENCE = ("csv", "xpt", "json")` — CSV wins so the `id` stays stable regardless of what formats are lying around. `_count_rows` counts newlines in binary chunks, capped at 8 MB (`rows=None` above that, and always `None` for xpt/json — counting would require parsing). Two provenance rules matter: `has_oracle` is `True` only for tier A and synth, set by provenance, never inferred from filenames, and `_scan_synth` only emits a domain if its `expected/<DOM>.csv` counterpart is actually on disk — no reference file, not gradeable, not listed.

THE PACKAGES. stdlib only (`json`, `os`, `threading`). `os.scandir` instead of `os.walk` because the benchmark tree is 788 MB / ~878 files per its docstring and only two fixed levels are relevant. Nothing to swap; pandas here would be pure weight.

SAS BRIDGE. This is the closest thing in the stack to a set of `libname` statements plus a metadata view of `sashelp.vtable` — a stable two-part name (`tier:ta:domain` instead of `libref.dataset`) resolving to a physical path. The analogy misleads on freshness: a libname resolves live at every reference, but this catalogue is a snapshot cached to JSON. Drop a new file into a corpus and it does not exist until someone calls `build_catalog(force=True)` (the `__main__` block does exactly that and prints the summary and scan seconds).

TRY IT YOURSELF.

```
import os

def count_rows(path, cap=8 * 1024 * 1024):
    if os.path.getsize(path) > cap:
        return None                      # too big to count cheaply: say so
    with open(path, "rb") as fh:
        data = fh.read()
    n = data.count(b"\n") + (0 if data.endswith(b"\n") else 1)
    return max(n - 1, 0)                 # minus the header

def catalogue(directory, tier, ta, has_oracle):
    entries = []
    for name in sorted(os.listdir(directory)):
        if not name.endswith(".csv"):
            continue
        domain = name.removeprefix("raw_").removesuffix(".csv").upper()
        path = os.path.join(directory, name)
        entries.append({"id": f"{tier}:{ta}:{domain}", "path": path,
                        "rows": count_rows(path), "has_oracle": has_oracle})
    return entries

cat = catalogue("/root/sdtm_review/data/synthetic_v4/raw", "synth", "SYNTH-001", True)
print(len(cat), cat[0])   # 11 {'id': 'synth:SYNTH-001:AE', ...}
```

LIMITS. The stale-cache trap above is the big one — the 375 figure I measured came from the cached JSON, not a fresh scan. Hard-coded absolute roots make the module box-specific; on any other machine three of six scanners silently return `[]` (each catches `OSError` per directory), so a thin catalogue looks like a small corpus, not a broken path. `_count_rows`'s newline count is documented as approximate: a quoted field containing an embedded newline inflates it. `_domain_of` trusts filenames, same ceiling as M1.3.

HOW TO IMPROVE IT. Move the six roots into one dict-of-roots constant (or env-var overrides) so porting is a config change; stamp the cache with a scan timestamp and root mtimes so staleness is at least detectable; unify with M1.3's `discover_sources` — two independent definitions of "what raw data exists", pointed at different corpus copies, is exactly the drift `_discover`'s docstring warns about, one module over.

LIFT AND REUSE. The entry schema is the reusable asset: `{id, ta, domain, path, format, rows, tier, has_oracle, available}` is a good universal descriptor for "a dataset something might run on", and `has_oracle`-by-provenance is a discipline worth exporting to any evaluation harness. Genericise by injecting the scanner list; leave the specific `_scan_*` functions behind.

M1.5 `/root/sdtm_review/services/study_upload.py` — a whole study arrives at once

A sponsor never hands over one dataset. They hand over a study — nine or eleven raw extracts from one therapeutic area — and want them mapped together. The Mapper page (M1.2) could already take one file; uploading eleven files one at a time means the same work done N times and N chances to lose track of which file got which spec.

WHAT IT DOES. Accepts a batch of raw CSVs as one uploaded study, writes them to `data/study_uploads/<token>/` with a `manifest.json`, and — the interesting part — decides per file whether an already-human-reviewed mapping spec applies to it, by comparing the file's header byte-for-byte against the 25 adjudicated header shapes. Exact match: the file is that shape and inherits its spec. No match: `shape_id: None`, honestly reported as having no spec.

WHY IT EXISTS. It closes the gap between "the corpus we adjudicated" and "a file a user just uploaded" without an LLM call and without fuzzy matching. The module docstring is explicit about what is deliberately absent: no fuzzy or subset header matching ("'Close enough' is how a column silently maps wrong"), no domain guessed from the filename when a header already answered it, no LLM ("Deciding whether two headers are equal is arithmetic").

INPUTS AND OUTPUTS. Input to `create_upload(files, label=None, uploaded_by=None)`: an iterable of `(filename, bytes)` pairs plus an optional label. Output: the manifest dict, also persisted. If our `raw_ae.csv` were uploaded, its header — 12 columns starting `SUBJECT,AE_NUMBER,AETERM_VERBATIM` — would be compared against each canonical shape header; an entry would record `{"file": "raw_ae.csv", "domain": ..., "shape_id": ..., "matched_by": "header"|"filename"|"none", "n_columns": 12, "n_bytes": ...}`. The upload directory `data/study_uploads/` does not exist on disk yet (no study has been uploaded on this box); it is created on first use with `os.makedirs(dest, exist_ok=True)`.

HOW IT WORKS. Per file: `safe_name()` flattens the client filename to a filesystem-safe basename (directory components discarded — measured: `safe_name("../../etc/passwd")` returns `passwd`); files rejected unless `.csv` (`ALLOWED_EXT`) and ≤ 25 MB (`MAX_FILE_BYTES`); `_read_header()` decodes the bytes as `utf-8-sig` (falling back to latin-1) and parses exactly the first line with `csv.reader`. Then the match:

```
shape_id = next((sid for sid, hdr in canonical.items() if hdr == header), None)
```

where `canonical` comes from `_canonical_headers()` — shape_id → header, read from disk via `mapping_review.headers_by_shape()` and `study_browser._shape_ids()` (the shapes index at `evidence/raw_inventory/specs/INDEX.json`, 25 shapes, verified this session). Only when no header matches does `infer_domain()` scan filename tokens against real IG domain codes — and a filename match still carries no spec, "so nothing rides on the guess". The token is `uuid.uuid4().hex[:12]`; `read_manifest(token)` refuses anything not matching `^[0-9a-f]{12}$` before the token is ever joined onto a path, so a traversal attempt is refused rather than sanitised into something plausible. `upload_sources()` re-emits every uploaded file in exactly the shape of M1.3's `discover_sources()` entries — `corpus: "upload"`, `oracle: "none (uploaded raw data - no expected output)"` — which is what lets uploads flow through the study browser and batch runner unchanged.

WHY CSV-ONLY IS A FEATURE HERE. The comment above `ALLOWED_EXT` gives the real reason: the header is the entire basis for matching a file to a reviewed spec, so it must be read exactly; accepting formats whose readers may coerce or reorder columns (xlsx via pandas, sas7bdat via pyreadstat) would make the "byte-for-byte" claim a lie. Also note the separate root: uploads live in `data/study_uploads/`, deliberately not `data/uploads/`, because the Mapper's app.py already owns that directory for flat single-file uploads and sharing it would mean one feature's cleanup deleting another feature's data.

THE PACKAGES. stdlib only: `csv`, `io`, `json`, `os`, `re`, `uuid`, `datetime`. Header equality needs none of pandas; `uuid4().hex[:12]` instead of a database sequence keeps upload identity filesystem-native (the manifest is the record, readable after restart).

SAS BRIDGE. The header-shape idea has no direct SAS ancestor, but the closest habit is checking `PROC COMPARE` on `CONTENTS` output before reusing a validated program on a new transfer: same columns, same order → the old program applies. This module makes that check the only admission rule and makes it exact. Where SAS instinct misleads: a SAS programmer would reach for the dataset label or filename to identify a domain first; here the filename is explicitly the last resort and never sufficient for a spec.

TRY IT YOURSELF.

```
import csv, io

known_shapes = {  # shape_id -> the exact header a human already reviewed
    "demo:AE": ["SUBJECT", "AE_NUMBER", "AETERM_VERBATIM", "AE_PT", "AE_SOC",
                "AE_START", "AE_END", "AE_SEVERITY", "AE_SERIOUS",
                "AE_OUTCOME", "AE_ACTION", "AE_ONGOING"],
}

def match(raw_bytes):
    text = raw_bytes.decode("utf-8-sig", errors="strict")
    header = next(csv.reader([io.StringIO(text).readline().rstrip("\r\n")]))
    for shape_id, canonical in known_shapes.items():
        if canonical == header:          # exact, or nothing -- never fuzzy
            return shape_id
    return None

blob = open("/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv", "rb").read()
print(match(blob))                       # demo:AE -- raw_ae.csv IS this shape
print(match(b"SUBJECT,AE_TERM\n1,x"))    # None -- close is not equal
```

LIMITS. Exactness cuts both ways: one renamed column, one reordered pair, or a BOM-less file re-exported with different quoting and the study arrives spec-less — correct behaviour by this module's philosophy, but the user experience is "no spec" with no explanation of how near the miss was. 25 MB per file is a real ceiling for large LB extracts. `list_uploads()` does an `os.listdir` per call — fine for tens of uploads, unindexed beyond that. No delete/expiry: uploads accumulate forever.

HOW TO IMPROVE IT. Report the near-miss diagnostics without acting on them ("matches shape `X` except column 7 renamed") so a human can adjudicate the delta into a new shape; add an expiry sweep keyed on `uploaded_utc`; store a content hash per file in the manifest so re-uploads of identical studies deduplicate.

LIFT AND REUSE. Lift the whole admission pattern: exact-fingerprint match against adjudicated artefacts, explicit `matched_by` provenance, guessing allowed only where nothing rides on it, tokens validated by pattern before touching paths. The right interface is exactly `create_upload(files: iterable[(name, bytes)]) -> manifest` plus `upload_sources() -> list[source_dict]` conforming to the discovery schema. Leave behind the coupling to `mapping_review`/`study_browser` internals (`mr._header_for`, `sb._shape_ids` — private-by-underscore functions imported across modules; it works, but genericise behind a `get_canonical_headers()` callable).

M1.6 `/root/sdtm_review/services/study_browser.py` — translate header shapes back into studies

The fleet did its mapping work in header shapes — 25 of them — because that is the honest unit of work. But no human thinks "show me shape `tierB_robustness:AE`". They think "show me Cardiology". This module is the translation layer between the two, and it refuses to hide the join.

WHAT IT DOES. Assembles the study-level view the review UI shows: every study in the tree with its datasets, each dataset resolved to its adjudicated header shape (or honestly unshaped), which other studies share that shape, and the master mapping spec for any (study, domain) pair. Read-only; the docstring's closing claim — "No LLM call happens in this module" — is true of the code.

WHY IT EXISTS. All 21 tier-B therapeutic areas share a byte-identical header per domain, so one adjudicated spec covers all 21. Shown naively, the UI would display 21 identical Cardiology/

Oncology/... specs "and imply 21 independent reviews existed". The `shared_with` field on every dataset row exists so the screen has to say one review covers them all.

INPUTS AND OUTPUTS. Inputs: the discovery scan (M1.3's `discover_sources`), uploads (M1.5's `upload_sources`), the shapes index `evidence/raw_inventory/specs/INDEX.json` (25 shapes; first entry `{"shape_id": "raw_library+tierA_oracle:AE", "domain": "AE", ..., "n_mapped": 59, "ok": true}`), and `services.spec_bridge` for spec rows. Outputs: `studies()` returns one dict per study (`study_id`, `corpus`, `oracle`, `n_datasets`, `n_with_spec`, `domains`); `datasets(sid)` one dict per file including `shape_id`, `header_matches_shape`, `shared_with`, `raw_columns`; `master_spec(sid, domain)` the full spec document; `spec_csv` the same as CSV text. For the running example, SYNTH-001's AE dataset appears under `study_id = "synthetic_v4::SYNTH-001"` (the id format is `f"{corpus}::{study}"` from `study_id(src)`).

HOW IT WORKS. `_discover()` is the whole ingest story in one function: it loads M1.3's script **by file path** — `importlib.util.spec_from_file_location("_inv", .../scripts/build_raw_variable_inventory.py)` — because `scripts/` is not a package, and caches the corpus scan in a module global, then appends `su.upload_sources()` uncached on every call. The comment gives the reason: a user who uploads a study expects it in the dropdown on the next request, not after a restart; the corpus doesn't change under a running process, uploads do. Shape resolution is two-layered. `_shape_of(src)` matches by name: an upload already carries its `shape_id` from upload time; a corpus file resolves via `shape_for(corpus, domain)`, which splits joined shape ids like `raw_library+tierA_oracle:AE` on `+` and matches the corpus against the parts — substring matching would make `tierA_oracle` match the joined id wrongly. Then `_verified_shape_of` re-reads the file's own header from disk and compares it to the shape's canonical header:

```
canonical = mr._header_for(shape, mr.headers_by_shape())
return shape if list(canonical) == _header(src["path"]) else None
```

A name-level match with a different real header is unshaped — no spec — never "shaped but flagged". The docstring calls this the premise of the whole shape collapse, "verified per file, not assumed". `master_spec` finally delegates row assembly to `spec_bridge.build_spec(shape_id, domain, header=...)` so viewing a spec and generating code from it go through one assembler with one precedence order: human decision > adjudicated fleet mapping > honest placeholder. Traversal safety is structural: `datasets(sid)` only ever compares the id against the discovered list, so a hostile `sid` resolves to "unknown study", not to a path.

THE PACKAGES. stdlib plus two sibling services. The one exotic move is the `importlib.util` file-path import; the alternative (make `scripts/` a package, or duplicate the glob) was rejected in-code: "there is exactly one definition of 'what raw files exist'; a second copy would drift the moment a corpus is added." Note the friction against M1.4: that promise holds within this module's world, but `datasets.py` is a second definition, pointed at a different corpus copy.

SAS BRIDGE. Think of `shared_with` as the output of `PROC SORT NODUPKEY` on program-input signatures across studies: the moment you see that 21 studies share one input signature, "21 mapping programs" collapses into one program with 21 librefs. The misleading part: in SAS you'd trust that collapse once established; here the identical-header premise is re-verified from disk on every call, because a new study can be dropped into a corpus directory at any time with the right name and the wrong columns.

TRY IT YOURSELF.

```
import csv, glob, os
from collections import defaultdict

def header(path):
    with open(path, newline="", encoding="utf-8-sig") as fh:
        return tuple(next(csv.reader(fh)))

# Group raw files by exact header shape -- the browser's core join.
by_shape = defaultdict(list)
for path in sorted(glob.glob("/root/sdtm_review/data/synthetic_v4/raw/*.csv")):
    by_shape[header(path)].append(os.path.basename(path))

for shape, files in by_shape.items():
    shared = "shared by " + ", ".join(files) if len(files) > 1 else "unique"
    print(f"{len(shape):2d} cols  {files[0]:<14s} {shared}")
# All 11 SYNTH-001 files print "unique": one study, 11 distinct shapes.
# Point the glob at corpus/tierB_robustness/*/ and domains collapse 21-to-1.
```

LIMITS. `studies()` re-reads every file's header (through `_verified_shape_of`) for every study on every call — correct-first, cache-never; fine at 25 shapes and a few hundred files, measurable pain at thousands. The `_sources_cache` global means a corpus file added mid-process never appears until restart (uploads deliberately excepted). Cross-module use of `mr._header_for` and the underscored `_shape_ids` couples three modules to private names. Header comparison is order- and spelling-exact by design, so a benign column reorder demotes a dataset to spec-less.

HOW TO IMPROVE IT. Cache `(path, mtime) -> header` so verification stays per-call-fresh without re-reading unchanged files; promote `_shape_ids` / `_header_for` to public functions with the contract the three consumers already rely on; surface why `header_matches_shape` is false (first differing column) for the UI.

LIFT AND REUSE. The generalisable design is "work units are content fingerprints; presentation re-joins fingerprints to human categories, and every row must disclose the sharing" — that transfers to any dedup-heavy pipeline. Lift `shape_for`'s careful corpus matching and the verify-on-read discipline. Leave behind the file-path import hack; in a fresh repo, put discovery in an importable module from day one.

M1.7 `/root/sdtm_review/services/study_context.py` — cross-domain facts one file cannot see

Map `raw_ae.csv` in isolation and three SDTM columns are impossible, not merely hard: `AESTDY` needs the subject's reference start date (which lives in DM), `EPOCH` needs the subject's element windows (which live in SE), and visit numbering needs the visit map (which lives in SV). No amount of cleverness on the AE file alone produces them — a model asked to try will guess.

WHAT IT DOES. Extracts exactly three cross-domain lookup tables from already-produced SDTM datasets in a study directory — `rfstdtc` (USUBJID → RFSTDTC from DM), `se_windows` (USUBJID → ordered `(SESTDTC, SEENDTC, EPOCH)` triples from SE), `visit_map` (VISIT → `(VISITNUM, VISITDY)` from SV) — bundles them in a `StudyContext` dataclass, and provides two pure derivations: `study_day(date_str, rfstdtc_str)` and `epoch_for(date_str, windows)`.

WHY IT EXISTS. Phase P1's `spec_enforce` deliberately leaves `--DY`, `EPOCH` and `VISITNUM` / `VISITDY` blank rather than let a generated program guess them (see `docs/STUDY_CONTEXT_LIMITATIONS.md`). This module supplies the missing facts "and only those facts". Without it, every run scores blank cells on three columns per domain forever; with a guessing version of it, wrong values would be indistinguishable from mapped ones — the worse failure.

INPUTS AND OUTPUTS. Input to `build_study_context(study_dir)`: a directory containing `DM.csv`/`SE.csv`/`SV.csv` (any case, located by `_locate`; missing files yield empty slices, not errors). Output, measured this session against `data/synthetic_v4/expected/`:

```
ctx.rfstdtc["SYNTH-001-101001"]      -> "2024-03-19"          (60 subjects total)
ctx.se_windows["SYNTH-001-101001"]   -> [("2024-03-07","2024-03-18","SCREENING"),
                                         ("2024-03-19","2024-06-09","TREATMENT"),
                                         ("2024-06-10","2024-07-10","FOLLOW-UP")]
ctx.visit_map["BASELINE"]            -> (2.0, 1);  "UNSCHEDULED" -> (4.1, None)
ctx.study_day_for(u, "2024-04-14")   -> 27
ctx.epoch_for_subject(u, "2024-04-14") -> "TREATMENT"
```

That closes the running example's loop: the raw row `101001,1,Insomnia,...,14-APR-2024` becomes `2024-04-14`, and the oracle row in `expected/AE.csv` for that subject/date reads `AESTDY=27, EPOCH=TREATMENT` — the derived values match the answer key exactly (checked this session).

HOW IT WORKS. `parse_iso_date` accepts only a complete `YYYY-MM-DD` (an optional `T...` time part is split off) and returns `None` for everything else — partial dates (`2024`, `2024-04`), explicit-unknowns (`2024-UN-11`), EDC formats (`11-APR-2024`), and calendar impossibilities (`2024-02-30`, caught by the `datetime.date` constructor). `study_day` implements the no-day-zero rule:

```
diff = (when - reference).days
return diff + 1 if diff >= 0 else diff
```

`epoch_for` walks the ordered windows and returns the epoch of the first window containing the date, inclusive on both ends, skipping malformed windows. The extractors are defensively plain: `extract_rfstdtc` omits subjects with blank RFSTDTC so `.get()` → `None` → blank, never a value derived from `""`; `extract_se_windows` drops half-open elements ("a half-open element cannot answer a containment question"); `extract_visit_map` keeps VISITNUM as float (4.1 is a legal unscheduled visit number) and VISITDY as int-or-`None` ("`None` says that where `0` would lie"). Three design constraints are stated as load-bearing in the module docstring: the derivations are **pure** (no file access, no globals — so the harness can inline the context into a generated R program that must reproduce the dataset without reading another domain's file); unusable input yields `None`, never a guess and never an exception (a partial date is normal in clinical data; an exception here would take a whole domain's run down); and **stdlib only**, because the module is read by the harness, the generated-code path and the tests.

THE MEASURED HONESTY. The docstrings carry their own verification numbers (theirs, from a prior session, not re-measured here): `study_day` reproduces 2477/2477 `--DY` cells of the SYNTH-001 oracle; `epoch_for` reproduces **1204/1243 (96.9%), not 100%**. All 39 residuals share one cause, documented in full: the corpus generator (M1.1) assigns EPOCH from the most recent attended visit (`_visit_at`), while it builds the SE follow-up element as `[last treatment visit + 1, follow-up visit date]` — a record dated strictly inside that span is inside the FUP element but its last attended visit is still a treatment visit. The rule here "is left exactly as SDTMIG states it. Nothing here is widened to absorb the 39 rows" — the shortfall lives in `tests/test_study_context.py` as a strict xfail. This is a genuine code-vs-code disagreement, deliberately preserved rather than papered over.

THE PACKAGES. stdlib (`csv`, `datetime`, `os`, `dataclasses`) — constraint 3 above. Swapping `csv.DictReader` for pandas would gain nothing and force the dependency into the generated-code path and tests.

SAS BRIDGE. `study_day` is the classic `AESTDY = AESTDT - RFSTDT + (AESTDT >= RFSTDT);` — the conditional `+1` because SDTM has no day 0, which `INTCK('day', ...)` alone gets wrong on and after the reference date. The whole module is a MERGE replaced by injected lookups: in SAS you would `PROC SORT; MERGE`

`ae(in=a) dm(keep=usubjid rfstdtc);` — here the generated program is forbidden from reading DM (the input-contract rule), so the harness computes the lookup once and inlines it as data. Where SAS instinct misleads most: dates stay ISO **strings** end to end, compared and stored unparsed until the last moment; there is no numeric date value, no informat, and a partial date is not a special missing value — it is just a string that fails `parse_iso_date` and propagates `None`.

TRY IT YOURSELF.

```
import datetime as dt

def parse(s):
    try:
        return dt.date.fromisoformat(s[:10]) if isinstance(s, str) else None
    except ValueError:
        return None

def study_day(date_str, rfstdtc):
    d, ref = parse(date_str), parse(rfstdtc)
    if d is None or ref is None:
        return None                    # partial date -> honest blank, not 0
    diff = (d - ref).days
    return diff + 1 if diff >= 0 else diff   # no day zero

def epoch_for(date_str, windows):
    d = parse(date_str)
    for start, end, epoch in windows:
        if parse(start) <= d <= parse(end):
            return epoch
    return None

windows = [("2024-03-07", "2024-03-18", "SCREENING"),
           ("2024-03-19", "2024-06-09", "TREATMENT")]
assert study_day("2024-04-14", "2024-03-19") == 27       # matches expected/AE.csv
assert epoch_for("2024-04-14", windows) == "TREATMENT"
assert study_day("2024-04", "2024-03-19") is None        # partial date
print("all good")
```

(Note: `date.fromisoformat` is looser than the module's hand-rolled parser on some malformed strings; the real `parse_iso_date` length/dash checks are stricter on inputs like `2024-4-1`.)

LIMITS. The 96.9% EPOCH ceiling is structural, not a bug to fix here — fixing it means choosing whose definition of EPOCH wins, generator or SDTMIG. `visit_map` is study-global, so two visits with the same name but different numbers across arms would collide (first row wins). `build_study_context` reads produced SDTM outputs, so context quality is downstream of DM/SE/SV mapping quality — the P1 dependency order (SV → DM → SE → everything else) exists precisely for this, and running early gives honest blanks. Partial-date arithmetic is simply refused; imputation is out of scope by design.

HOW TO IMPROVE IT. Add an imputation layer as a separate, flagged module (never silently inside these functions); carry per-subject visit maps to survive name collisions; expose a `context_coverage(ctx, domain_rows)` report so a run can state up front how many rows will get blanks and why.

LIFT AND REUSE. This is the most liftable file in the module: stdlib-only, pure derivations, dataclass container, `None`-propagating semantics — copy `parse_iso_date`, `study_day`, `epoch_for` into any SDTM tooling as-is. The right interface is exactly what it has: `build_study_context(dir) -> StudyContext` for I/O, pure functions for math. Leave behind nothing except the SYNTH-001-specific verification claims, which you must re-measure against your own oracle.

What this module still owes you

- **The spec side.** You have seen files arrive, get profiled, and get matched to shapes — but not what a mapping spec row actually contains or how `spec_bridge.build_spec` merges human decisions over fleet output. That precedence engine is the next module's core.
- **Code generation and execution.** The running example stops at "AESTDY would be 27". How a generated Python/R program is prompted, executed against `raw_ae.csv`, repaired on failure, and compared cell-for-cell to `expected/AE.csv` is Modules 2–4 territory.
- **The fleet and adjudication.** The 25 shapes and `INDEX.json` were treated as given here; how two independent models proposed mappings, how consensus/`review`/`one_model_only` verdicts were adjudicated, and how the learning loop feeds back is not covered.
- **`mapping_review` internals.** M1.5 and M1.6 both lean on `mr.load_ig()`, `mr.headers_by_shape()` and `mr._header_for()` without opening that module.
- **The two-repo split.** Noted but not resolved: `/root/sdtm_review/CLAUDE.md` still titles itself "SDTM Mapper", `datasets.py` reaches into `/root/sdtm_mapper`'s corpus while `discover_sources` walks the local copy, and the knowledge packs differ between forks (`sdtm_ig_v4.json` here vs `sdtm_ig34.json` there). Treat nothing as shared between the trees without checking.
- **Unmeasured here.** The 2477/2477 and 1204/1243 verification counts, the "210 files → 25 shapes" collapse, and the 788 MB benchmark size are the repos' own documented numbers; this session re-verified only single spot values (AESTDY 27, EPOCH TREATMENT, 78 rows, 375 catalogue entries, 25 shapes in INDEX.json). Everything runs against synthetic or public pilot data only.

Module 2 — Knowledge and terminology

Every mapping the pipeline produces is only as good as what it knows: which variables a domain has, which values those variables may legally hold, and where those facts come from. This module walks the knowledge layer — the compiled "knowledge pack" the mapper reads, the CDISC controlled-terminology package both apps share, and the reference sources (CDISC Library cache, pinned upstream repos, accumulated human decisions) that the review app layers on top. The running thread is one file, `/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv`, whose `AE_SEVERITY` column says `Mild` while the standard demands `MILD` — and the machinery that notices the difference.

M2.1 `/root/sdtm_mapper/knowledge/build_kp.py` — compiles the SDTM rulebook into JSON

A SAS programmer opening the SDTMIG keeps three facts in their head per variable: its label, whether it is Required, and which codelist governs it. Ask a program to do the same and those facts have to live somewhere a program can read — not in a 400-page PDF. This 106-line script is where the mapper's copy of those facts was typed in.

WHAT IT DOES. `build_kp.py` is a run-once generator: it holds a hand-written Python dictionary describing 7 SDTM domains (DM, AE, VS, EX, DS, LB, CM — 136 variables total, per the pack it emits) and dumps it to `knowledge/sdtm_ig34.json`. The JSON, not the script, is what the mapper loads at runtime. Think of the script as source code for a data file.

WHY IT EXISTS. The alternative is either parsing the SDTMIG PDF (fragile) or calling the CDISC Library API at runtime (network dependency, licence key). Freezing a curated subset into JSON makes the mapper deterministic and offline. Without it, `services/engine.py`, `services/codegen.py` and `services/validator.py` in the mapper all fail on import — each one opens the JSON at module load:

```
#| skip-check
# sdtm_mapper/services/engine.py, line 8 — and validator.py:9, codegen.py:13 do the same
KP = json.load(open(os.path.join(os.path.dirname(__file__), "..", "knowledge", "sdtm_ig34.json")))
```

INPUTS AND OUTPUTS. Input: nothing external — the IG content is transcribed by hand into the `KP` literal. Output: `sdtm_ig34.json` (21,678 bytes in the mapper copy), shaped `{"ig_version": "SDTMIG v3.4 (SDTM v2.0)", "domains": {"AE": {"class", "label", "structure", "keys", "signature_hints", "variables"}, ...}}`. For the running example, the AE entry carries `"keys": ["STUDYID","USUBJID","AEDECOD","AESTDTC"]`, signature hints like `"adverse event"`, `"severity"`, `"meddra"` (used by domain detection in Module 1), and per-variable rows such as:

```
{"name": "AESEV", "label": "Severity/Intensity", "type": "Char", "core": "Perm", "ct": "MILD/
MODERATE/SEVERE"}
```

HOW IT WORKS. The whole file is one dict literal plus two statements. Variables are typed compactly as five-element lists to keep the source readable, then expanded to dicts just before dumping:

```
#| skip-check
# tail of build_kp.py — the only executable logic in the file
for dom,info in KP["domains"].items():
    info["variables"]=[{"name":v[0],"label":v[1],"type":v[2],"core":v[3],"ct":v[4]} for v in
info["variables"]]
json.dump(KP, open("/root/sdtm_mapper/knowledge/sdtm_ig34.json","w"), indent=1)
```

Note the hardcoded absolute output path — this script writes into the mapper tree no matter where you run it from.

THE SAS BRIDGE. The nearest SAS analogue is a metadata spec sheet plus a format catalog rolled into one dataset — the thing you'd build with `PROC IMPORT` from the sponsor's define spec and query with `PROC SQL`. The analogy misleads in one important way: a SAS format applies itself (`FORMAT AESEV $sev.;` changes what you see), while this pack is inert data. Nothing here converts anything; every consumer must read the `ct` field and act on it — or, as M2.3 shows, fail to.

THE PACKAGES. `json` from the stdlib, nothing else. Alternatives — YAML for human editing, SQLite for querying — buy nothing at this size; a 21 KB JSON loads in microseconds and diffs cleanly in git.

TRY IT YOURSELF. The compact-row expansion pattern, end to end:

```
import json, tempfile, os

# Compact rows: [name, label, type, core, ct] -- five strings, no keys.
KP = {
    "ig_version": "demo",
    "domains": {
        "AE": {"class": "Events", "label": "Adverse Events",
               "keys": ["STUDYID", "USUBJID", "AEDECOD", "AESTDTC"],
               "variables": [
                   ["AETERM", "Reported Term for Adverse Event", "Char", "Req", ""],
                   ["AESEV",  "Severity/Intensity", "Char", "Perm", "AESEV"],
                   ["AEOUT",  "Outcome of Adverse Event", "Char", "Perm", "OUT"],
               ]}}}

# The expansion step build_kp.py runs before dumping:
for dom, info in KP["domains"].items():
    info["variables"] = [{"name": v[0], "label": v[1], "type": v[2],
                          "core": v[3], "ct": v[4]} for v in info["variables"]]

path = os.path.join(tempfile.gettempdir(), "demo_kp.json")
json.dump(KP, open(path, "w"), indent=1)
pack = json.load(open(path))
assert pack["domains"]["AE"]["variables"][1]["ct"] == "AESEV"
print("pack written:", path, "| AE vars:", [v["name"] for v in pack["domains"]["AE"]["variables"]])
```

LIMITS. Hand-transcribed, so it inherits every transcription choice. The most consequential one: the `ct` field mixes codelist names (`"OUT"` for AEOUT), literal value enumerations (`"MILD/MODERATE/SEVERE"` for AESEV), dictionary references (`"MedDRA PT"` for AEDECOD) and format markers (`"ISO8601"`) in one string field with no type tag. Every consumer must re-guess which kind it is looking at — the mapper's validator does exactly that (M2.3). Seven domains also means a raw file for any other domain simply has no home. Both limits are what the review app's v4 pack fixed: `sdtm_review/knowledge/sdtm_ig_v4.json` (built by `build_kp_v4.py` from `SDTMIG_v4_Tabulations.xlsx`, 316 KB) covers 21 domains and 693 variables and splits the field into `ct`, `codelist`, `allowed_terms`, `format`, plus `role`, `c_code` and `definition`. There, AESEV's codelist is properly named `"AESEV"`, not spelled out as values.

HOW TO IMPROVE IT. Stop hand-transcribing: derive the pack from the CDISC Library cache that `ig_reference.py` (M2.7) already reads — 63 domains, 1,917 variables, authored by CDISC rather than retyped. Keep the JSON as the runtime artifact; change only its provenance. And make the output path relative to the script.

LIFT AND REUSE. The pattern to keep is "compile the standard into a versioned JSON artifact; load it, never fetch it." The interface is just the JSON shape: `domains -> {class, label, keys, variables[{name,label,type,core,ct}]}`. Leave behind the hand-typed literal and the hardcoded path.

If you rebuild elsewhere, generate the same shape from an authoritative source and keep `ig_version` inside the file — provenance travels with the data.

M2.2 `/root/sdtm_review/services/knowledge.py` — one place chooses which pack loads

Two knowledge packs sit in `sdtm_review/knowledge/`: the 7-domain v3.4 pack and the 21-domain v4 pack. Three different modules need one of them. If each module picked for itself, a validator checking against v3.4 while codegen wrote against v4 would disagree about what AE even contains — and the disagreement would surface as mystery findings, not as an error.

WHAT IT DOES. `services/knowledge.py` is the single loader for the review app's knowledge pack. It decides which JSON file to use (environment override, then v4, then v3.4 as fallback), loads it once, caches it, and hands every caller the same dict.

WHY IT EXISTS. The docstring states the contract plainly: "engine.py, codegen.py and validator.py all consume the pack. They MUST import it from here so the pack path is chosen in exactly one place." A grep confirms the discipline spread further — `ta_ct.py`, `value_domains.py`, `ta_mapper.py`, `ta_compliance.py`, `ta_registry.py`, `dds_export.py` and `parity_harness.py` all import from it. Without it you get the mapper's situation (M2.1), where three files each open the JSON by their own relative path.

INPUTS AND OUTPUTS. Inputs: the env var `SDTM_KNOWLEDGE_PACK` (absolute path or bare filename resolved inside `knowledge/`), else `sdtm_ig_v4.json`, else `sdtm_ig34.json`. Output: the pack dict. For the running example, `load_pack()["domains"]["AE"]` returns 59 variables under the default v4 pack, where AESEV carries `"codelist": "AESEV"`, `"core": "Perm"`, `"role": "Qualifier"` and a real definition ("The degree of something undesirable.").

HOW IT WORKS. Four small functions in a straight line. `_candidates()` builds the ordered path list; `load_pack()` walks it:

```
def load_pack(force=False):
    global _cache, _loaded_path
    if _cache is not None and not force:
        return _cache
    errors = []
    for path in _candidates():
        try:
            with open(path) as f:
                pack = _validate(json.load(f), path)
        except Exception as e:  # missing, unparseable, or structurally wrong -> next candidate
            errors.append(f"{path}: {e}")
            continue
        _cache, _loaded_path = pack, path
        return _cache
    raise RuntimeError("No loadable SDTM knowledge pack. Tried:\n  " + "\n  ".join(errors))
```

`_validate()` rejects any file without a non-empty `domains` dict, so a truncated JSON degrades to the fallback instead of poisoning every consumer. `pack_path()` reports which file actually won — useful in reports, and honest when the fallback fired. Helpers `domains()` and `ig_version()` save callers a key lookup.

THE SAS BRIDGE. This is `OPTIONS FMTSEARCH=(work library)` — an ordered search path for the catalog that resolves the first hit and stops. The analogy holds unusually well, including the failure mode it

prevents: two programmers with different `FMTSEARCH` orders producing different outputs from identical code. One place sets the order; everyone inherits it.

THE PACKAGES. `json` and `os` only. A config framework would be the classic over-engineering here — the entire "configuration system" is one env var and a two-item fallback list.

TRY IT YOURSELF. The candidate-walk-with-validation pattern, self-contained:

```
import json, os, tempfile

d = tempfile.mkdtemp()
json.dump({"ig_version": "v4-demo", "domains": {"AE": {}}},
          open(os.path.join(d, "pack_v4.json"), "w"))
open(os.path.join(d, "pack_broken.json"), "w").write("{not json")

def load_pack(candidates):
    errors = []
    for path in candidates:
        try:
            pack = json.load(open(path))
            if not pack.get("domains"):
                raise ValueError("no usable 'domains' mapping")
            return pack, path
        except Exception as e:
            errors.append(f"{path}: {e}")
    raise RuntimeError("No loadable pack:\n  " + "\n  ".join(errors))

pack, used = load_pack([os.path.join(d, "pack_broken.json"),
                        os.path.join(d, "pack_v4.json")])
assert used.endswith("pack_v4.json"), "fallback should have skipped the broken file"
print("loaded", pack["ig_version"], "from", os.path.basename(used))
```

LIMITS. The fallback is silent from a caller's perspective: if v4 goes missing, everything keeps running against the 7-domain v3.4 pack, and only a caller who thinks to compare `pack_path()` notices that 14 domains just vanished. The cache is process-global with no invalidation on file change (`force=True` exists but nothing calls it in the services). And the two packs have different per-variable schemas (`ct` only vs `ct` + `codelist` + `allowed_terms`), which is why downstream code hedges with `v.get("codelist") or v.get("ct")` — a seam you will see again in M2.4 and M2.5.

HOW TO IMPROVE IT. Log a warning naming the pack actually loaded whenever it is not the first candidate; that single line converts the silent-degrade limit into a visible event. Normalising both packs to one schema at load time would delete the `codelist`-or-`ct` hedge from every consumer.

LIFT AND REUSE. Lift the whole file — it is already generic: an ordered-candidates JSON loader with structural validation and a cache. The interface to preserve: `load_pack()`, `pack_path()`, and the rule that no consumer opens the file itself. Leave behind the specific filenames; they are this repo's history.

M2.3 `/root/sdtm_review/services/ct.py` + `ct_index.py` — the codelists, loaded once, answered two ways

The raw AE file says a subject's outcome was `RECOVERED/RESOLVED`. Is that a legal value for AEOUT, or just a plausible-looking one? The answer is not a matter of judgment — CDISC publishes the exact list — but only if something actually opens the published file. These two small modules are the only code in the review app that does.

WHAT IT DOES. Both modules read the same file: `/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv`, a 13 MB export of the CDISC SDTM controlled-terminology package (path overridable via the `SDTM_CT_PATH` env var). `ct.py` answers one question — "which submission values does codelist X contain?" — as `{codelist name: [terms]}`, 1,158 codelists on this box. `ct_index.py` answers the richer per-term questions Define-XML needs: NCI concept code, human decode, synonyms, and whether the codelist is extensible.

WHY IT EXISTS — THE SCAR. This is where the project's most instructive failure lives, so it gets told in full. In the earlier mapper, controlled terminology entered the model prompt as a name only. The prompt builder in `sdtm_mapper/services/engine.py` (line 91) formats each variable as:

```
#| skip-check
f"  {v['name']} | {v['label']} | {v['type']} | core={v['core']}" + (f" | CT={v['ct']}" if v['ct']
else "")
```

So the model saw `AEOUT | Outcome Adverse Event | Char | core=Perm | CT=OUT` — the codelist's name, never its terms. No codelist was ever supplied to the model. Asked to produce values for a variable whose legal list it had never seen, each model did the only thing it could: it emitted the human-readable side of the terminology it knew from training. The repo's own code documents the concrete instances. `value_domains.py` records `CMDOSFRQ = "AS NEEDED"` where the codelist says `PRN` — and the CT package shows `AS NEEDED` is not a term in codelist FREQ while `PRN`'s published decode is literally `"As needed"` (NCI code C64499): the model emitted the decode where the submission value belonged. The same docstrings record EGTESTCD emitted as `HR, PR, QRS, QT, QTC, RHYTHM` — six readable abbreviations, none of them submission values; the published terms are `EGHRMN, PRAG, QRSAG, QTAG, RHYNOS, INTP`. And the check that should have caught it was structurally unable to: the mapper's `validator.py` value-checks CT only "where a value set is known", and its `_ct_set()` deliberately returns `None` for any codelist name — conservative by design, so `CT=OUT` and `CT=ACN` were never value-checked at all. Where the specifics above are not in the code, treat this as the class of failure the current design guards against: a pipeline can pass its own checks with every coded value wrong, because supplying the codelist's name is not supplying the codelist. `ct.py`'s docstring encodes the lesson as policy: names that resolve to no codelist "are simply not CT-checkable — callers must treat that as 'unchecked', never as 'passed'."

INPUTS AND OUTPUTS. Input: the CT CSV, whose structure is positional-by-reference — a codelist row has a blank `Codelist Code` column; each term row points back to its parent via that column. Output of `ct.terms_for("AESEV", "AESEV")`: `['MILD', 'MODERATE', 'SEVERE']`. Output of `ct_index.codelist("AESEV")`: code `C66769`, name "Severity/Intensity Scale for Adverse Events", `extensible: False`, and per-term entries like `term("OUT", "RECOVERED/RESOLVED")` → `{"code": "C49498", "decode": "Recovered from Adverse Event", ...}` — which settles the cold-open question: yes, the raw value is already a legal submission value.

HOW IT WORKS. `ct.load_codelists()` is one pass with a module-level cache:

```
code_to_name = {r["Code"]: r["CDISC Submission Value"]
                for r in rows if not r["Codelist Code"]}
out = {}
for r in rows:
    parent = r["Codelist Code"]
    if parent and parent in code_to_name:
        out.setdefault(code_to_name[parent], []).append(r["CDISC Submission Value"])
```

A missing file returns `{}` — surfaced by callers as "unchecked", never as a pass. `codelist_for(var, pack_ct)` resolves a variable's pack annotation to a real codelist name, with a `_FALLBACK` restoring

`RACE` and `ETHNIC` (dropped by the v4 pack scrape but published by CT). `ct_index.index()` (an `@lru_cache(maxsize=1)` build over the same rows) adds the four Define-XML facts and computes each term's `decode` with preference order synonym → NCI preferred term → the submission value itself. Its `resolve(names)` splits multi-codelist fields on whitespace — `"EGTESTCD HETESTCD"` is one IG field naming two codelists, "the bug that made `gates.py` pass EG while 3,500 records were in violation" per its docstring. `term(codelist, value)` is the single-term lookup.

THE SAS BRIDGE. A codelist is exactly a format's domain, and the CT CSV is the source you'd feed `CNTLIN=` to build the catalog: `MILD` ↔ `Grade 1` is a coded value and its label. But the analogy inverts at the crucial point, and the scar above is what the inversion costs. In reporting, you apply formats code → label so humans read `Recovered from Adverse Event`. In SDTM you submit the code side: `RECOVERED/RESOLVED` goes in the dataset, the decode goes in Define-XML. A programmer (or a language model) whose instincts run format-forward will emit labels where codes belong — which is precisely what the models did.

THE PACKAGES. `csv` and `os` from the stdlib, plus `functools.lru_cache` in `ct_index`. Pandas could load the CSV in one call, but a 13 MB file with `csv.DictReader` takes a couple of seconds once per process and adds no dependency; the by-hand two-pass build is also the clearest possible statement of the file's parent/child structure.

TRY IT YOURSELF. Parse the real CT package and look up the AE codelists (the file exists on this box; swap the path for your own CT export elsewhere):

```
import csv

CT = "/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv"

with open(CT, encoding="utf-8-sig", newline="") as fh:
    rows = list(csv.DictReader(fh))

# Codelist rows have a blank "Codelist Code"; term rows point back to them.
code_to_name = {r["Code"]: r["CDISC Submission Value"]
                for r in rows if not r["Codelist Code"]}
codelists = {}
for r in rows:
    parent = r["Codelist Code"]
    if parent in code_to_name:
        codelists.setdefault(code_to_name[parent], []).append(
            (r["CDISC Submission Value"], r["CDISC Synonym(s)"]))

for name in ("AESEV", "OUT"):
    print(name, "->", [t for t, _ in codelists[name]])

# Submission value vs. the human-readable synonym CDISC also publishes:
for term, syn in codelists["AESEV"]:
    print(f"  submit {term!r:12} synonyms: {syn}")
```

Output on this box: `AESEV -> ['MILD', 'MODERATE', 'SEVERE']`, `OUT -> ['FATAL', 'NOT RECOVERED/NOT RESOLVED', 'RECOVERED/RESOLVED WITH SEQUELAE', 'RECOVERED/RESOLVED', 'RECOVERING/RESOLVING', 'UNKNOWN']`, and AESEV's synonyms `1; Grade 1` / `2; Grade 2` / `3; Grade 3`.

LIMITS. The synonym-first decode preference backfires exactly where you just saw it: `ct_index` reports MILD's decode as `"1"` (its first synonym) rather than anything readable — fine for LBTESTCD where the synonym is "Hemoglobin", unhelpful for AESEV. The whole layer is pinned to one dated CT release (2025-03-28); newer terms simply don't exist here, and nothing warns you the package is stale. Both caches (`ct._cache` by path, `ct_index`'s `lru_cache`) live until process death, so replacing the CSV under

a running app changes nothing. And `ct.py` returning `{}` for a missing file is safe only because every caller honours the "unchecked ≠ passed" contract — a new caller that treats an empty list as "no constraints" silently reopens the scar.

HOW TO IMPROVE IT. Give `ct_index` a decode preference that skips synonyms which are bare digits or "Grade N". Record the CT package's date in every report that used it (the manifest already names it; reports should too). Consider one shared row-loading function — today `ct.py`, `ct_index.py`, `ta_ct.extensible_codelists` and `value_domains.synonyms` each re-read the same 13 MB CSV into their own cache.

LIFT AND REUSE. This pair is the most portable code in the module: any CDISC shop can point `SDTM_CT_PATH` at their own quarterly CT export and get `load_codelists` / `terms_for` / `codelist` / `term` unchanged. The interface worth preserving is three-valued honesty: a lookup returns terms, or `None` for "not CT-checkable" — never an empty "pass". Leave behind `_FALLBACK`; it patches one specific pack's scrape gap.

M2.4 `/root/sdtm_review/services/ta_ct.py` — decide what terminology governs each variable

Run a CT check naively over all 693 variables of the v4 pack and most of the output is noise: dates flagged for not being severity terms, MedDRA columns flagged because we hold no MedDRA licence, numeric results flagged against a findings codelist. The check that matters — is `AE_SEVERITY`'s `Mild` legal for AESEV? — drowns. This module exists to sort the 693 into "checkable" and five distinct flavours of "not checkable" before any value is examined.

WHAT IT DOES. `ta_ct.py` classifies every variable's terminology annotation into one of six kinds, then checks column values only where a check is meaningful. Its docstring tabulates the split; measured today against the v4 pack the counts are: `terms` 204, `iso8601` 66, `external` 13, `multiple` 4, `sponsor` 2, `none` 404. (The docstring says 5 multiple and 401 none — mild drift, the code and pack have moved since the table was written; the 204 checkable figure still holds.)

WHY IT EXISTS. Two design commitments, both learned the hard way. First, "not checked is not the same as valid" — an unlicensed MedDRA column is reported as unchecked, never silently passed. Second, near misses are separated from unknowns: `Mild` where the codelist says `MILD` is a case defect with a mechanical fix; `Slightly sore` is a mapping problem needing a human. Per the docstring, case defects are "the largest single source of the value disagreements between the R and Python arms" — burying the cheap fix under the expensive one would misdirect all review effort.

INPUTS AND OUTPUTS. `classify("AESEV", "AESEV")` (variable name, pack annotation) returns:

```
{"kind": "terms", "codelist": "AESEV", "terms": ["MILD", "MODERATE", "SEVERE"],
 "extensible": false, "numeric_exempt": false,
 "note": "Not extensible: only the published terms are legal."}
```

`check_values(values, spec)` then takes one column's values and returns counts plus `violations` and `near_misses` lists. Fed the 78 `AE_SEVERITY` values from `raw_ae.csv`: 0 conformant, 0 violations, and three near-misses — `Mild` (40 rows), `Severe` (22), `Moderate` (16) — each carrying `"expected": "MILD"` etc. `assess_dataset(domain, csv_path)` wraps this over every column of a produced dataset and totals it.

HOW IT WORKS. `classify` is a guard ladder: blank → `none`; `ISO8601` → format-check kind; `*` / `SPONSOR` → sponsor-defined; `MEDDRA` / `LOINC` / `SNOMED` / `WHODRUG` / `WHODD` / `UNII` → external; otherwise split the annotation on whitespace and keep the names that resolve in `ct.load_codelists()` (one resolved →

`terms`, several → `multiple`, none → `none` with an honest note). Extensibility is read from the CT package by `extensible_codelists()` so a novel term on an extensible list becomes a notice, not an error. `check_values` never counts blanks (absence is a core/required question belonging to the validator, and double-reporting one defect under two rules is explicitly avoided), validates ISO 8601 with a regex that accepts legal partial dates (`2024`, `2024-04`), and splits failures by case:

```
bad = Counter(v for v in present if v not in legal)
base["n_conformant"] = len(present) - sum(bad.values())
for value, n in bad.most_common(100):
    entry = {"value": value, "count": n}
    # A case-only difference is a different, and much cheaper, problem.
    if value.lower() in lower:
        entry["expected"] = lower[value.lower()]
        base["near_misses"].append(entry)
    else:
        base["violations"].append(entry)
```

One surgical exemption: variables ending `STRESC` hold standardised results in character form, which for a quantitative test is the number itself; numeric strings are exempted there only, because checking `"76"` against 253 findings terms once reported a correct dataset as 1,646 violations (the comment records the incident).

THE SAS BRIDGE. This is the triage you do implicitly when deciding which columns get a `$fmt.`-based edit check at all: you'd never validate a date column against a severity format, and you know AEDECOD is checked by the coder against MedDRA, not by you against a catalog. What the analogy misses is that SAS gives you nowhere to record "deliberately unchecked" — a variable without a format is just unformatted. Here every unchecked variable carries a `kind` and a note saying why, and that reported classification is the difference between "passed" and "never looked", the exact distinction the M2.3 scar turned on. Also note `scope` in `assess_dataset`'s output: terminology checking confirms a value is in the codelist, not that it is the right term for the record — "a row coded MILD when the subject was severe conforms perfectly."

THE PACKAGES. Stdlib only: `csv`, `re`, `collections.Counter`, plus `services.ct` and the pack loader. A validation framework (pandera, great_expectations) could express the membership rule but not the six-kind classification or the near-miss split, which is where all the value is.

TRY IT YOURSELF. The near-miss split against the real raw file:

```
import csv
from collections import Counter

RAW = "/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv"
LEGAL = {"MILD", "MODERATE", "SEVERE"}          # codelist AESEV, C66769

with open(RAW, newline="", encoding="utf-8") as fh:
    values = [row["AE_SEVERITY"] for row in csv.DictReader(fh)]

lower = {t.lower(): t for t in LEGAL}
near, bad, ok = Counter(), Counter(), 0
for v in values:
    v = v.strip()
    if not v:
        continue
    if v in LEGAL:
        ok += 1
    elif v.lower() in lower:
        near[v] += 1        # case-only defect: cheap fix
    else:
        bad[v] += 1         # real mapping problem
print(f"{ok} conformant, near misses: {dict(near)}, violations: {dict(bad)}")
```

Prints `0 conformant, near misses: {'Moderate': 16, 'Mild': 40, 'Severe': 22}, violations: {}` on this box.

LIMITS. The docstring's count table has drifted (measured 404/4 vs written 401/5) — trust `classify` run over the pack, not the prose. Near-miss detection is case-only; `MILD` with a trailing space inside the raw value is caught (values are stripped), but `M1LD` or `Moderate-Severe` land in `violations` with no "did you mean". The external-dictionary set is a hardcoded frozenset of six names; a pack annotating `"WHO-DD"` with a hyphen would fall through to `none`. And extensibility downgrades novel terms to notices — correct per the standard, but it means an extensible codelist can never fail, only murmur.

HOW TO IMPROVE IT. Add an edit-distance "did you mean" to violations (stdlib `difflib.get_close_matches` suffices). Regenerate the docstring table from the code in a test so it can't drift again. Fold the four separate CT-CSV readers (noted in M2.3) into one.

LIFT AND REUSE. Lift `classify` + `check_values` as a pair; they are the general engine for "membership checking with honest abstentions". The interface: a spec dict with `kind/terms/extensible`, a result dict where `checkable: False` is loud. Leave behind the pack-schema hedge (`v.get("codelist") or v.get("ct")`) and the STRESC exemption unless your data has the same shape — but keep the idea that every exemption names the incident that justified it.

M2.5 `/root/sdtm_review/services/value_domains.py` — show real values, then catch guessing

A generated LB program once mapped `"White Blood Cells"` to WBC — reasonable, except the raw column actually said `"White Blood Count"`, so the branch never fired and `TRUE ~ NA_character_` turned every miss into a silent blank. Half of 3,000 rows lost their LBTESTCD; all 3,000 lost LBSPEC. The R code wasn't broken. It was confident about data it had never been shown.

WHAT IT DOES. The largest file in this module (896 lines) closes both ends of that defect. Before generation, `prompt_block()` injects into the codegen prompt the observed distinct values of every source column and the legal CDISC submission values of every target — "these are facts, not examples". After generation, `coverage_report()` compares the produced dataset back against the raw

file and names every source value whose target came out blank on every occurrence. A third layer, `spec_ct_conflicts()`, checks the mapping spec's own prose for instructed values the codelist forbids.

WHY IT EXISTS. The module docstring names the root cause: "the codegen prompt names the source column and names the codelist, and shows the model neither one's contents. So it invents the domain it is reading from and the domain it is writing to." This is the M2.3 scar made operational — and the docstring adds the design judgment that matters: "the detector matters more than the prompt… the generator is not deterministic even at temperature 0, so a prompt improves the odds of a draw while a check rejects a bad one." Every structural rule passes on a dataset built from an invented lookup; only comparison against the source can see it.

INPUTS AND OUTPUTS. For the running example — spec rows mapping `AE_SEVERITY → AESEV` and `AE_OUTCOME → AEOUT` against `raw_ae.csv` — `prompt_block("AE", spec, raw_csv)` returns exactly this text (real output from this box):

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

Everything the model needs to write `Mild → MILD` is now in front of it. `observed_values()` returns per-column `{free_text, n_distinct, values[{value, n}]}` — `AETERM_VERBATIM` (10 distinct here) would be enumerated, but a real verbatim column past 60 distinct values is reported as free text instead of pasted. `coverage_report()` returns `{aligned, findings[{target, source, unmapped_values, ...}]}`.

HOW IT WORKS. Layer by layer, with the constants first: `MAX_VALUES = 60`, `FREE_TEXT_THRESHOLD = 60`, `MAX_TERMS = 60` — and a comment warning that MAX_VALUES must never drop below the threshold, because the block says "contains EXACTLY these N value(s)" and a lower cap would make the prompt state a falsehood (six real columns once sat in that gap). `normalise_row()` absorbs both spec shapes in play (`sdtm_var`/`raw_var` from the TA pack, `target`/`source` from codegen, plus `derived_from` lists), refusing to treat classification words like `DERIVED` as column names. `logic_sources()` regex-scans a rule's English prose for real column names on word boundaries, because a Derived variable arrives with `source: ""` and its true input named only in a sentence — LBTESTCD being the case that mattered. `codelist_terms()` delegates legality to `ta_ct.classify` (so prompt and checker cannot disagree) and then `_rank_terms()` reorders large codelists by resemblance to the observed values — LBTESTCD holds over a thousand terms and the first 60 alphabetically are useless; a synonym match ("Hemoglobin" → HGB via the CT synonym column) outranks every text heuristic. For wide Findings domains, `candidate_terms()`/`pivot_block()` propose `--TESTCD` terms per raw column name, returning nothing on a score tie ("a tie at the top is genuine ambiguity… let it reach the review queue"). The detector end is the most battle-scarred code: `coverage_report()` refuses positional comparison unless it can prove row order, because "every generated program sorts by the natural key before writing" — measured on AE, 114 rows in, 114 out, and 43 AETERM values not on the raw line they came from. `_alignment_witness()` probes with a verbatim-copied column (requiring ≥5 distinct values, since constant STUDYID agrees under any permutation, and taking the worst surviving witness), and `_row_pairing()` recovers a content-based join from verbatim columns when order was lost — "what recovers AE, CM, LB, MH and PE." Only then does it count, per source value, whether the target was blank on every occurrence:

```
dead = sorted((v for v in seen if blanked[v] == seen[v]),
              key=lambda v: -seen[v])
```

`is_not_collected()` exempts variables the spec deliberately leaves blank — before that check, 21 of 29 findings on the Cardiology pack were false positives (a measured 72% rate, recorded in the docstring). Finally `spec_ct_conflicts()` extracts values a spec's prose instructs (`_instructed_literals` parses arrows positionally: literals before `->` are inputs, first literal after is the output) and reports any the codelist forbids to a human — because "an explicit instruction must outrank a suggestion", so a bad spec must be fixed, not prompt-overridden.

THE SAS BRIDGE. `prompt_block` is what a careful clinical programmer does before writing a `PROC FORMAT`-driven recode: run `PROC FREQ` on the source column, open the IG page for the target's codelist, and only then write the mapping — with the discipline that every level in the FREQ output appears in the code. The `OTHERWISE ' '` in a SAS `SELECT` is exactly the `TRUE ~ NA_character_` that caused the LB disaster; both turn an unhandled level into a silent blank. What has no SAS analogue is the detector: because generation here is a nondeterministic draw, the FREQ-before-coding step is re-run mechanically after coding, against the produced dataset. If you remember one thing: the prompt is `PROC FREQ` for the model; the coverage report is `PROC COMPARE` against the source, run because the model cannot be trusted to have read the FREQ.

THE PACKAGES. Stdlib throughout — `csv`, `re`, `os`, `Counter`. Pandas would shorten `_read`/`_column` but the file's complexity is all in the decisions (what counts as a source, a witness, a dead value), which no library supplies.

TRY IT YOURSELF. The prompt-block core — observe the source, state the legal targets:

```
import csv
from collections import Counter

RAW = "/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv"
FREE_TEXT_THRESHOLD = 60
legal = {"AESEV": ["MILD", "MODERATE", "SEVERE"],
         "AEOUT": ["FATAL", "NOT RECOVERED/NOT RESOLVED", "RECOVERED/RESOLVED",
                   "RECOVERED/RESOLVED WITH SEQUELAE", "RECOVERING/RESOLVING", "UNKNOWN"]}
mappings = [("AESEV", "AE_SEVERITY"), ("AEOUT", "AE_OUTCOME")]

with open(RAW, newline="", encoding="utf-8") as fh:
    rows = list(csv.DictReader(fh))

for target, source in mappings:
    counts = Counter(r[source].strip() for r in rows if r[source].strip())
    print(f"* {target}")
    if len(counts) > FREE_TEXT_THRESHOLD:
        print(f"    SOURCE {source} is free text ({len(counts)} distinct) -- copy, don't
enumerate.")
    else:
        vals = ", ".join(f'"{v}"' for v in counts)
        print(f"    SOURCE {source} contains EXACTLY these {len(counts)} value(s): {vals}")
    print(f"    TARGET must be a CDISC submission value from {target}: {', '.join(legal[target])}")
```

LIMITS. Almost every limit here is measured and written next to its fix, which is the file's real signature. Still open: `coverage_report` is defeated by pivoting domains (unequal row counts → `aligned: False`, checked: nothing) — a refusal, not a solution. `_instructed_literals` is a hand-rolled English parser whose comment history (`possessive apostrophes opened phantom quotes`, `= is deliberately not an arrow`) tells you it will meet a sentence shape it mishandles again. Term ranking is "deliberately crude" by its own admission — it only needs to pull candidates into a 60-term window. And the free-text threshold means a coded column with 61 levels gets no enumeration at all.

HOW TO IMPROVE IT. For pivoting domains, compare sets per subject-visit key instead of refusing. Replace the prose-literal parser with a rule: specs must carry structured `maps_to` pairs, and prose-only rules get flagged as unparseable rather than half-parsed. Log each prompt block alongside the run so a bad draw can be audited against exactly what the model was shown.

LIFT AND REUSE. Three genuinely portable ideas: (1) enumerate the source's observed domain into the prompt with an exactness guarantee, capped honestly; (2) rank a large legal-value list by resemblance to observed data, synonyms first; (3) never positionally compare datasets without an alignment witness, and prefer the worst witness. The interface is plain: `prompt_block(domain, spec, raw_csv) -> str` and `coverage_report(produced, raw, spec, domain) -> dict`. Leave behind the dual spec-shape handling and the negation heuristics borrowed from `ta_pack` — those are this repo's sediment.

M2.6 `/root/sdtm_review/services/ta_knowledge.py` — reviewer decisions become reusable mapping rules

Five reviewers spent days annotating mappings in the review app, and for a while all of it fed exactly nothing: remarks went into one database table, the exportable rule base read from a different one, and no code joined them — `knowledge/mapping_rules.json` held 0 decisions and 0 rules. This module is the join, and the thesis it implements is bigger than the bug: a decision a human makes once should never need a model again.

WHAT IT DOES. `ta_knowledge.py` harvests accepted reviewer decisions from both ledgers in `data/review.db` (the append-only `ta_events` table and the `mapping_decisions` table), aggregates them into `knowledge/mapping_rules.json`, and then uses that file deterministically: `propose_from_rules()` builds a mapping spec from rules alone — "No model is called" — and `rule_coverage()` reports what fraction of a raw file's columns the accumulated rules already handle. The docstring names that percentage "the product's real progress metric": at 0% every study needs the model fleet; at 100% a new study is a reproducible Python run.

WHY IT EXISTS. The 390 spec rows in the tool were produced by models (two per domain, a third adjudicating disagreements) — "generation was a guess; only the checking was deterministic." Human review is the only step that converts a guess into knowledge, and without this module that knowledge evaporated per-study. With it, "`severity` maps to `AESEV`" is recorded once and applies to every future sponsor's file.

INPUTS AND OUTPUTS. Inputs: rows from the two SQLite ledgers — an event-ledger decision is an accepted proposal (proposed by one named reviewer, resolved by another; comments deliberately don't count, and declined proposals are kept as `accepted: False` so a rejected idea can't be re-proposed forever). Output: the rules JSON, whose `rules[]` entries are keyed on `(domain, normalised raw column name)` with per-target provenance (`decided_by`, `n_sources`, `last_decided`) and a `contested` flag. For the running example, an accepted decision that `AE_SEVERITY` maps to `AESEV` becomes a rule keyed `("AE", "severity")` — which then also matches a next sponsor's `IT.severity` or bare `Severity`.

HOW IT WORKS. `normalise()` is the load-bearing function and is deliberately narrow:

```
text = str(name or "").strip().lower()
text = re.sub(r"^[a-z0-9]{1,8}\.", "", text)          # dotted EDC prefix
text = re.sub(r"[^a-z0-9]+", "_", text).strip("_")
if domain:
    prefix = domain.strip().lower() + "_"
    if text.startswith(prefix) and len(text) > len(prefix):
        text = text[len(prefix):]
```

Only two prefixes are stripped — a dotted EDC prefix and the domain code — because an earlier, greedier version turned `event_term` into `term`, a key generic enough to collide across the domain: "a missed match asks a human, a false match answers wrongly." `build_rules()` merges both ledgers time-ordered (single writer since 2026-07-27; `mapping_review.export_rules` now delegates here), splits accepted from rejected, groups RAW-sourced decisions by normalised column and everything else into `derivations`, counts `n_sources` as distinct origin×domain pairs (one decision is a data point; the same decision reached independently is a convention), keeps contested rules unresolved ("two studies legitimately mapping a column differently is information"), and writes atomically via a temp file and `os.replace`. `rules_index()` serves the deterministic lookup with an mtime-keyed cache, indexing only uncontested single-target rules. `propose_from_rules()` walks a raw header through that index, emitting decided mappings plus explicit `unresolved_columns` and `contested_columns` — the unresolved list doubling as "the list of questions a reviewer should be asked next."

THE SAS BRIDGE. This is your company macro library and specification archive, made queryable: the accumulated "how we always map this" knowledge that in a SAS shop lives in senior programmers' heads and last study's define.xml. The analogy breaks on provenance and dissent: a copied SAS spec tells you nothing about who decided or whether anyone disagreed, while every rule here names its decider, counts independent confirmations, and preserves contested mappings instead of letting the loudest copy win. Note also what the JSON honestly disclaims: these are "DECISIONS, not validated correctness — only an executed run graded against an oracle establishes that."

THE PACKAGES. `json`, `os`, `re`, `sqlite3`, `collections.defaultdict` — stdlib. SQLite is read directly rather than through an ORM; two SELECTs don't justify one.

TRY IT YOURSELF. Normalisation carrying one decision across three sponsors' spellings:

```
import re

def normalise(name, domain=None):
    text = str(name or "").strip().lower()
    text = re.sub(r"^[a-z0-9]{1,8}\.", "", text)      # dotted EDC prefix
    text = re.sub(r"[^a-z0-9]+", "_", text).strip("_")
    if domain:
        prefix = domain.strip().lower() + "_"
        if text.startswith(prefix) and len(text) > len(prefix):
            text = text[len(prefix):]
    return text

# One accepted decision becomes a rule keyed on the normalised column name...
rules = {("AE", "severity"): {"sdtm_var": "AESEV", "rule": "Uppercase to CT term"}}

# ...and three sponsors' spellings of the same column all reach it:
for raw in ("AE_SEVERITY", "IT.severity", "Severity"):
    key = ("AE", normalise(raw, "AE"))
    hit = rules.get(key)
    print(f"{raw:14} -> {key[1]:10} -> {hit['sdtm_var'] if hit else 'UNRESOLVED'}")
```

LIMITS. The rule key is a column name; a sponsor whose severity column is called `intensity` won't match `severity` however it's normalised — names, not meanings, are what recur, and the module bets on that. Coverage counts columns resolved, not values handled: a rule can match the column and still carry a `rule` string whose logic no longer fits the new sponsor's values (the value-domain machinery of M2.5 is what would catch that downstream). `decisions_from_table` trusts `mapping_decisions` rows as accepted by construction. And the current rule base is small — `mapping_rules.json` on this box is 7.4 KB; the flywheel is designed, not yet spun up.

HOW TO IMPROVE IT. Add synonym-aware matching as a proposal tier (name match = decided; synonym match = suggested, reviewer confirms) so `intensity` can at least be asked about. Track per-rule hit counts in use, so dead rules age visibly. Surface `rule_coverage` on the review UI's front page — a progress metric nobody sees moves nobody.

LIFT AND REUSE. The whole pattern generalises to any human-in-the-loop mapping product: append-only decision ledger → aggregated, provenance-carrying rule file → deterministic proposer + coverage metric. The interface is the rules JSON schema and `normalise`. Leave behind the two-ledger merge — that's the scar of this repo's history; a fresh build should have one ledger from day one.

M2.7 `/root/sdtm_review/services/ig_reference.py` + `cdisc_repos.py` — CDISC's own metadata, found and pinned

The 21-domain pack took real effort to build, and the whole time a richer version of the same facts sat unused on this box: the CDISC Library's own variable metadata, cached by the CORE conformance engine at `/opt/cosa` — 63 domains, 1,917 variables. These two modules are about using what CDISC already published, and knowing exactly which copy of it you are using.

WHAT IT DOES. `ig_reference.py` loads two pickles from the CORE cache (`variables_metadata.pkl`, `variable_codelist_maps.pkl`, root default `/opt/cosa/cdisc-rules-engine/resources/cache`, overridable via env var `CORE_CACHE`) and serves SDTMIG v3.4 metadata read-only: per-variable `label`, `description`, `core`, `role`, `type`, `ordinal`, `codelist_codes`, `sdtm_class`. `cdisc_repos.py` is the registry of the three CDISC 360i upstream repositories cloned at `/root/cdisc360i_p2` (env `CDISC360I_ROOT`), each pinned to a recorded commit under tag `clincoder-pinned-20260726`, with `verify()` reporting drift and `attribution()` emitting the MIT notice.

WHY THEY EXIST. For `ig_reference`: the difference from the local pack "is not size, it is kind." The pack says AETERM is Char/Req/Topic; the Library adds "Verbatim name of the event." — and a description is a matchable fact, how a raw column called `event_term` can be routed to AETERM by a Python function instead of a language model. It also covers domains the 21-domain pack cannot express (CV, TU, TR, RS, MB, MS, IS, CE — the oncology/infectious-disease gap). For `cdisc_repos`: on 2026-07-26 the three CDISC repos existed in three places on this box at five different commits, one copy six months stale — and this project had already paid for a stale second copy once (TLF Studio, 226 commits behind, holding unbacked-up work). One canonical location, asked for by key, never hardcoded again.

INPUTS AND OUTPUTS. `ig_reference.variable("AE", "AETERM")` on this box returns core `Req`, role `Topic`, description "Verbatim name of the event."; `topic_variable("AE")` → `AETERM`; `domain_class("AE")` → `Events`; `summary()` reports 63 domains / 1,917 variables and, importantly, `"authored_here": False`. `cdisc_repos.verify()` today reports all three repos `pinned` (dds `47a3e0835`, dde `096b2bfe3`, 360i `24bd98bdc`) and lists four known-stale copies it deliberately does not delete ("other tools on this box may still point at them").

HOW IT WORKS. `ig_reference` is load-and-index, nothing more — "Nothing in this module infers, guesses or derives," and callers needing judgment "build it on top and say so." `_raw()` unpickles once behind `lru_cache(maxsize=1)`, keyed into the cache dict by the pinned IG version:

```
IG_VERSION = "3-4"
_VAR_KEY = f"library_variables_metadata/sdtmig/{IG_VERSION}"
```

— pinned rather than "latest", because "a mapping plan that silently changed IG version between runs would be unreproducible, which is the whole failure this codebase is trying to leave behind." `variables()` sorts by the IG's own `ordinal` (variable order is itself a conformance rule, CORE-000852). `_class_of()` reads Events/Findings/Interventions from the metadata's model-class hyperlink; `domain_class()` takes a majority vote of the variables' classes, with an explicit table for the classless domains — DM returning `""` once "made DM look like a routing failure when it was a correct answer" — so Special Purpose, Trial Design and Relationship are named. On the security question the module is unusually candid: these are pickles, unpickling executes arbitrary code, and they are trusted only because they are an operator-installed dependency that the CORE engine itself unpickles on every run — with the caveat that `CORE_CACHE` "must never be made settable from a request." `cdisc_repos` keeps a dict of three repo records (commit, upstream URL, licence, key paths); `path(key, *parts)` resolves files inside a repo and raises `RepoUnavailable` if it isn't cloned; `verify()` shells out to git (`rev-parse HEAD`, `describe --tags --exact-match`, `status --porcelain`) and grades each repo `pinned` / `DRIFTED` / `DIRTY` / `UNPINNED` / `MISSING` — "reports; never repairs."

THE SAS BRIDGE. `ig_reference` is the closest thing this codebase has to SAS dictionary tables — a queryable, authoritative catalog you interrogate (`topic_variable`, `required`, `variables`) instead of retyping the IG. The mismatch to keep in mind: `DICTIONARY.COLUMNS` describes your data; this describes the standard, so it can judge your data. `cdisc_repos` has no SAS-culture analogue at all, and that absence is the lesson for the SAS-trained half of the audience: validated SAS environments version the installation, but study code that `%include`s from a shared drive has exactly the five-copies-three-commits problem this module kills — a dependency you can't name the version of is a dependency you can't trust an artifact built from.

THE PACKAGES. Stdlib again: `pickle`, `os`, `functools.lru_cache`, `subprocess` for git. No GitPython — three read-only git queries don't justify a dependency.

TRY IT YOURSELF. A minimal pin-verifier over the real canonical root (adjust `ROOT` / `PINS` on another box):

```
import os, subprocess

ROOT = "/root/cdisc360i_p2"
PINS = {"DataExchange-DDS": "47a3e0835",
        "data-definition-engine": "096b2bfe3",
        "360i": "24bd98bdc"}

def git(repo, *args):
    out = subprocess.run(["git", "-C", repo, *args],
                         capture_output=True, text=True, timeout=20)
    return out.stdout.strip() if out.returncode == 0 else ""

for name, expected in PINS.items():
    d = os.path.join(ROOT, name)
    if not os.path.isdir(d):
        print(f"{name:24} MISSING"); continue
    head = git(d, "rev-parse", "HEAD")
    tag = git(d, "describe", "--tags", "--exact-match")
    status = "pinned" if head.startswith(expected) else f"DRIFTED to {head[:9]}"
    print(f"{name:24} {status}  tag={tag}")
```

Prints three `pinned tag=clincoder-pinned-20260726` lines on this box today.

LIMITS. `ig_reference` is machine-local by construction: no CORE cache, no reference (`available()` says so cleanly, but nothing rebuilds it — installing CORE at `/opt/cosa` is an operator task outside this repo). It serves v3.4 while the review app's default pack is v4-labelled; two IG vintages coexist in one system, and nothing reconciles them. `domain_class` by majority vote is a heuristic with a hardcoded escape table — a genuinely novel special-purpose domain would come back `""`. In `cdisc_repos`, the pinned commits and `KNOWN_STALE` list are literals maintained by hand; `verify()` detects drift of the clones but nothing detects drift of the registry. And `_git` swallowing every exception into `""` means "git not installed" and "repo corrupt" both read as empty string.

HOW TO IMPROVE IT. Run `cdisc_repos.verify()` in CI or app startup and surface non-`pinned` loudly — a check nobody runs is documentation. Give `ig_reference` a JSON export step so downstream boxes can carry the reference without CORE or pickles. Reconcile pack vs reference versions in one visible place (`summary()` of each already exists; diff them).

LIFT AND REUSE. From `ig_reference`, lift the stance: prefer the standard body's own metadata over your transcription, pin its version, and report `authored_here: False`. From `cdisc_repos`, lift the whole module — registry-of-pinned-dependencies with `path()`, `verify()`, `attribution()` is generic to any project that vendors upstream repos. The interface is `path(key, *parts)` and the `verify()` report shape. Leave behind the pickle format (an accident of CORE's cache) and this box's literal paths.

What this module still owes you

Honest gaps, so you know what reading this did not buy:

1. **The v4 pack builder is undocumented here.** `sdtm_review/knowledge/build_kp_v4.py` (which parses `SDTMIG_v4_Tabulations.xlsx` into the 693-variable pack the review app actually runs on) was outside this module's file list; M2.1 documents its v3.4 predecessor and the output schema, not the xlsx-parsing itself.
2. **CT checking stops at membership.** Every layer here answers "is this value in the codelist" — none answers "is it the right term for this record." A dataset coded `MILD` for a severe event passes everything in this module; `ta_ct.assess_dataset` says so itself in its `scope` field.

3. **External dictionaries are a hole, honestly labelled.** MedDRA and WHODrug columns (AEDECOD, AEBODSYS, CMDECOD) are classified `external` and never checked; 13 of 693 variables are unverifiable on this box, and no amount of code changes that without licences.
4. **The rule-base flywheel is designed, not demonstrated.** `mapping_rules.json` is 7.4 KB; `rule_coverage` is a metric with almost no history yet. The claim that coverage climbs toward model-free mapping is architecture, not evidence.
5. **Counts you can trust vs counts you inherited.** Measured in this write-up: 1,158 codelists, 204/66/13/4/2/404 classification split, 78 AE rows all case-near-misses, 63 domains/1,917 variables in the reference, three repos pinned. Inherited from docstrings (plausible, incident-derived, but not re-measured here): the 3,500-record EG violation, the 72% Cardiology false-positive rate, the 43-of-114 AE misalignment.
6. **Two IG vintages coexist.** The review app's default pack self-describes as SDTMIG v4.0 while `ig_reference` pins v3.4, and the mapper still runs its own 7-domain v3.4 pack. Which vintage governs which check is answerable only per-call-site, not globally.

Module 3 — The AI layer

Modules 1 and 2 covered the deterministic machinery: profiling raw data, knowledge packs, spec skeletons. This module covers the one part of the pipeline that is not deterministic — the LLM calls — and the four layers wrapped around them to make an unreliable component usable in a regulated context: a fallback cascade with a spend guard, a usage ledger, a privacy filter that decides what may leave the box, and a validation gate that filters what comes back. The single most important idea, for a reader coming from SAS: a macro invocation expands the same way every time; a model call does not, **even at temperature 0** — this project measured that, and the whole architecture is shaped by it.

M3.1 `/root/sdtm_mapper/services/llm.py` — one function, four fallback brains

It is 2 a.m., a batch run is mapping eleven domains, and the primary model gateway starts refusing calls because a monthly spend limit tripped. In most tools that is the end of the run. Here it is one line in an error list, because `call_llm()` was built on the assumption that every backend will eventually fail mid-run — and the job of this file is to make sure the caller never notices.

WHAT IT DOES. This file is the only door to any language model in the SDTM Mapper. Every classify, spec-generation, code-generation and repair call goes through one function, `call_llm(prompt, ...)`, which tries up to four backends in a fixed cheapest-first order and returns the first answer it gets, along with the id of the model that produced it. It also polices money: a "spend guard" blocks any call to the one metered backend that would use an expensive model, and sends a Telegram alert instead.

WHY IT EXISTS. Two problems. First, availability: each backend has its own way of dying (CLI not installed, OAuth spend limit, HTTP 402, rate limits), and a mapping run that hard-fails on the first hiccup is useless. Second, cost: this project had a real credit-drain incident on 2026-07-19 (an expensive model was reachable on the per-token OpenRouter route; the file's own comments describe callers being able to pick "the most expensive model available and bill it to us"), after which the guard code was added. Without this file, every service would carry its own HTTP code, its own retry logic, and its own opportunity to leak money.

INPUTS AND OUTPUTS. Input: a prompt string, optional system string, and a `tier` — one of `"classify"`, `"spec"`, `"code"`, `"repair"` — which selects how strong a model each backend should use, without the caller ever naming a model. For the AE running example, the classify call from `services/engine.py` passes a prompt containing the 12 profiled `raw_ae.csv` columns and `tier="classify"`. Output: a 2-tuple `(text, model_id)`, e.g. `('{"domain":"AE","confidence":92,...}', "glm-4.7")`. On total failure it raises `RuntimeError("All LLM backends failed: ...")` with one entry per backend tried.

HOW IT WORKS. The cascade in `call_llm()` (line 276 in this file) is a chain of `try/except` blocks, each appending to an `errors` list and falling through:

```
def call_llm(prompt, system="", max_tokens=4096, temperature=0.0,
             retries=3, tier="spec", override_model=None):
    errors = []
    if override_model:
        if OPENROUTER_KEY:
            try:
                return _call_openrouter(override_model, prompt, system,
                                        max_tokens, temperature, retries)
            except Exception as e:
                errors.append(f"override:{e}")
    if _commandcode_available():
        try:
            return _call_commandcode(CC_MODELS.get(tier, CC_MODELS["spec"]),
                                     prompt, system)
        except Exception as e:
            errors.append(f"commandcode:{e}")
```

The order after that: `_oauth_available()` → `_call_claude_cli(...)` (the `claude` CLI under a Max subscription, on when `SDTM_USE_CLAUDE_OAUTH=1`), then `_call_zai(...)` (z.ai GLM, a fixed-price subscription), then `_call_openrouter(...)` (metered per token, deliberately last). Each tier resolves to a different model per backend via four dicts — `CC_MODELS`, `OAUTH_MODELS`, `ZAI_MODELS`, `MODELS` — all overridable by env vars (`SDTM_CC_SPEC`, `SDTM_OAUTH_SPEC`, `SDTM_MODEL_SPEC`, `ZAI_STRONG_MODEL`, …).

Three details are worth a career's attention:

The docstring lies; the code is right. The `call_llm` docstring still says "Claude OAuth (best) -> OpenRouter (Opus 4.8) -> z.ai GLM" — the pre-2026-07-19 order. The code puts GLM ahead of OpenRouter. Both CLAUDE.md files flag this drift explicitly ("trust the code, not either comment"). This book documents the code.

Key resolution has a scrape fallback and therefore a kill switch. `_openrouter_key()` first reads `OPENROUTER_API_KEY`, then — if unset — opens `/root/clinassist_v2/config.py` (a different app's config) and regex-extracts a live `sk-or-v1-…` key. So unsetting the env var does not disconnect OpenRouter. The only reliable off switch is `SDTM_DISABLE_OPENROUTER=1`, which makes `_openrouter_key()` return `""` before either lookup. And because `OPENROUTER_KEY = _openrouter_key()` runs at module import, the switch must be set before `services.llm` is imported — `scripts/batch_run.py` does exactly that.

The spend guard gates on list price, not call cost. `_or_spend_guard(model, in_chars, max_tokens)` blocks an OpenRouter call unless the model is allowlisted (`OR_ALLOWED_MODELS`, default deepseek/minimax/gemini-flash), its list price is ≤ `OR_MAX_PRICE_PER_MTOK` ($3/1M default), and the worst-case call cost is ≤ `OR_MAX_CALL_USD` ($3 default). The comment explains why price-per-Mtok is the binding constraint: a single premium call on a short prompt costs ~$0.11, so a per-call cap alone "would never have stopped the incident." Unknown models price as `(5.0, 25.0)` — fail-expensive. `:free`-suffixed models bypass the allowlist by construction (a free model cannot spend). A second, independent guard, `_assert_openrouter_allowed()`, hard-denies vendor prefixes in `PREMIUM_ON_OPENROUTER` (`anthropic/`, `openai/`, …) unless `SDTM_ALLOW_PREMIUM_OPENROUTER=1`. Blocked calls fire `_or_tg_alert()` to a Telegram bot; break-glass is `OR_ALLOW_EXPENSIVE=1`, which alerts loudly instead of blocking.

THE SAS BRIDGE — WHERE THE MACRO ANALOGY BREAKS. A SAS programmer's nearest mental model for `call_llm()` is a macro: text in, text out, called from many places. The analogy holds for the interface and fails for the semantics. `%mymacro(AE)` produces byte-identical expansion on every invocation, forever; that is why a validated SAS program stays validated. `call_llm(prompt, temperature=0.0)` does not. The file's own header claims "temperature 0 deterministic, reproducible output (regulatory requirement)" — and this project measured that claim to be false on the backend actually in use: on 2026-07-26, the same codegen prompt for the same AE domain produced 308-, 260- and 288-line R

programs across three runs in one hour on glm-5.2, each with different runtime defects; a 9-domain regeneration went from 7/9 running programs to 2/9 on identical prompts. Sending `temperature: 0.0` (which every call here does) reduces sampling randomness; it does not obligate the provider's serving stack to be bit-reproducible. The consequence for architecture: nothing downstream trusts a single generation. Generated code is executed and graded (Module 4's executor and data-equivalence gate); a draw that fails is regenerated, and a claim like "the fix improved codegen" is unsupported from one before/after pair. If you remember one thing from this module: **a macro expansion is a fact, a model call is a draw.**

A SCAR: THE EMPTY-ANSWER FAILURE CLASS. Project history includes an incident where a thinking-mode default caused calls to silently return empty content while running 5–13× slower — the model spent its budget on hidden reasoning and the wrapper treated the empty string as an answer. The current trees contain no thinking-mode configuration at all (grep for "thinking" across both repos returns nothing), so the incident's exact mechanics cannot be shown from code here; what can be shown is that every CLI-shaped backend now guards the exact failure shape. `_call_commandcode()` raises on empty stdout ("commandcode returned empty output"), and `_call_claude_cli()` treats both empty output and refusal text as errors — because the `claude` CLI prints refusals on stdout with exit code 0:

```
CLI_REFUSALS = ("not logged in", "spend limit", "usage limit", "rate limit",
                "quota exceeded", "please run /login", "credit balance is too low")

def _guard_cli_output(out):
    # the shape of the check inside _call_claude_cli()
    if not out or any(r in out.lower() for r in CLI_REFUSALS):
        raise RuntimeError(
            f"claude CLI returned no usable output: {out[:120] or '(empty)'}")
```

The comment above the real check records the documented instance of this class: on 2026-07-27 a run in a sibling tool hit a monthly spend limit and four sections of two generated SAP documents read "You've hit your monthly spend limit" — refusal text written into the artefact as if it were content. The lesson generalises: **ANY SUBPROCESS- OR HTTP-WRAPPED MODEL MUST TREAT "EMPTY" AND "POLITE REFUSAL" AS HARD ERRORS**, or the cascade silently ships garbage.

THE PACKAGES. Standard library only: `urllib.request` for HTTP (no `requests`, no vendor SDK), `subprocess` for the two CLI backends, `json`, `re`, `time`. That is a deliberate choice: zero dependencies means this file works in any Python 3 environment the Flask app runs in, and there is no SDK version to drift. Cost of the choice: no streaming, no typed responses, hand-rolled retry loops. Swapping to a vendor SDK would buy typed usage objects and retries but bind each backend to a package and its release cadence — for four heterogeneous backends (two CLIs, two HTTP APIs in different wire formats: OpenAI-style for OpenRouter, Anthropic-style for z.ai), stdlib is the honest floor.

TRY IT YOURSELF. The cascade mechanic, with all four backends stubbed — note the OAuth stub failing the way the real CLI fails, on stdout with a refusal:

```
def commandcode(prompt):
    raise RuntimeError("commandcode: binary not on PATH")

def claude_oauth(prompt):
    out = "You've hit your monthly spend limit"   # stdout, exit code 0
    if "spend limit" in out.lower():
        raise RuntimeError("claude CLI returned no usable output: " + out[:40])
    return out, "oauth:model"

def zai_glm(prompt):
    return '{"domain": "AE", "confidence": 92}', "glm-stub"

def openrouter(prompt):
    raise AssertionError("should never be reached: GLM answered first")

def call_llm(prompt):
    errors = []
    for name, backend in [("commandcode", commandcode), ("oauth", claude_oauth),
                          ("glm", zai_glm), ("openrouter", openrouter)]:
        try:
            return backend(prompt)
        except Exception as e:
            errors.append(f"{name}:{e}")
    raise RuntimeError("All LLM backends failed: " + " | ".join(errors))

text, model = call_llm("Classify this dataset.")
assert model == "glm-stub" and "AE" in text
print("answered by", model, "after 2 fall-throughs")
```

LIMITS. (1) The determinism claim in the header is false in practice — measured, above. (2) Backend availability probes are cached per process (`_oauth_ok`, `_cc_ok` are module globals set once), so a backend that logs in mid-run is not picked up until restart. (3) A hardcoded z.ai key fallback exists in source (`ZAI_KEY = os.environ.get("ZAI_API_KEY") or "408d…nVi7"` — redacted here); a secret in a source file survives every env audit. (4) The Telegram alert is fire-and-forget over `urllib` with all exceptions swallowed — a blocked call whose alert also fails is invisible except in the raised `RuntimeError`. (5) Retries are per-backend (`retries=3` inside `_call_openrouter` and `_call_zai`), so a worst case walks ~8 attempts before failing — `llm_mapper.py`'s docstring counts this honestly.

HOW TO IMPROVE IT. Make the availability probes re-checkable with a TTL instead of process-lifetime caching; move the hardcoded z.ai fallback key into the same file-based secret pattern the Telegram token uses; and log every cascade fall-through (which backend failed, why) to the usage ledger the sdtm_review fork added — today the error list dies with the exception unless all four backends fail.

LIFT AND REUSE. The reusable interface is exactly `call_llm(prompt, system, tier) -> (text, model_id)` plus the two-guard pattern for any metered backend: an allowlist + list-price gate at the single choke point the money flows through, not in config. Genericise by making the backend list a data structure of `(available_fn, call_fn)` pairs rather than inline blocks; keep the rule that every backend's failure is a string in a list, never a crash.

M3.2 `/root/sdtm_review/services/llm.py` — the same client, now it counts tokens

A batch run finishes and someone asks the only question that matters commercially: what did that cost? In the sdtm_mapper fork the honest answer is "we don't know — reconstruct it from provider dashboards." In the sdtm_review fork the answer is `usage_summary()`, because this file — same name, same cascade, different code — appends one JSON line per model call to a ledger on disk.

WHAT IT DOES. This is the sdtm_review fork's copy of the LLM client. The cascade, spend guard, tier dicts, kill switch and CLI-refusal handling are line-for-line the code you just read in M3.1. On top of that it adds a usage-accounting layer: every backend call records `(backend, model, tier, prompt_tokens, completion_tokens, estimated)` to `/root/sdtm_review/data/llm_usage.jsonl`, and two helper functions let a batch run report only its own consumption.

WHY IT EXISTS. The sdtm_mapper file returns text and forgets the transaction. That was fine until batch runs (17-domain fleets, repeated regeneration under a nondeterministic backend) made "how many tokens did model X actually consume" a question someone had to answer from data, not from list-price arithmetic after the fact. The docstring of `_record_usage` states the design goal: "One choke point for every backend so a batch run (or anything else) can reconstruct real per-model token counts instead of estimating from list prices after the fact."

THE DELTA, EXPLICITLY. Diffing the two files (120 changed lines) yields exactly three changes — nothing else differs:

1. **Usage recording.** New module constants `ROOT` and `USAGE_LOG_PATH`, and three new functions: `_record_usage(...)` (append one line, never raise — logging failure is swallowed so it can never break the call it decorates), `usage_offset()` (current byte size of the log), `usage_summary(since_offset=0)` (aggregate by model from a byte offset onward).
2. **Tier threading.** Every private backend function gains a `tier=""` keyword (`_call_commandcode`, `_call_claude_cli`, `_call_openrouter`, `_call_zai`) so the ledger can record which routing bucket a call belonged to; `call_llm` passes `tier=tier` at each call site.
3. **A better `extract_json` failure.** When a model answers in prose with no JSON at all, the mapper fork's `extract_json` reaches `min(cands)` on an empty list and dies with a bare `ValueError("min() iterable argument is empty")` — the comment records that this "killed a whole domain mid-batch" while naming neither the model nor the caller. The review fork raises `ValueError("model returned no JSON object or array; first 200 chars: ...")` instead. The mapper fork still has the bare-`min()` bug today.

INPUTS AND OUTPUTS. Same call surface as M3.1. The ledger line is the new output. A real line from `/root/sdtm_review/data/llm_usage.jsonl` (78 lines on disk as of 2026-08-03; this file exists only in the review fork — `/root/sdtm_mapper/data/` has no such file):

```
# a real line from the ledger, shown as a Python literal (JSON false -> False)
usage_line = {"ts": "2026-07-27T18:21:06Z", "backend": "zai", "model": "glm-4.7",
              "tier": "classify", "prompt_tokens": 991, "completion_tokens": 127,
              "estimated": False}
```

That specific line is the AE-scale classify call shape: a ~1,000-token prompt (the 12-column profile plus candidate domains), ~130 tokens of JSON back.

HOW IT WORKS. The interesting design choice is the `estimated` flag. The two HTTP backends return real token counts in their response bodies, so `_call_openrouter` reads `usage.prompt_tokens`/`usage.completion_tokens` (OpenAI wire shape) and `_call_zai` reads `usage.input_tokens`/`usage.output_tokens` (Anthropic wire shape) and both record `estimated=False`. The two CLI backends print only the answer text — no usage block — so they estimate `len(text) // 4` tokens per side and record `estimated=True`. A comment in `_call_claude_cli` documents the deliberate refusal to switch the CLI to `--output-format json` just for accounting: "switching to it changes what stdout contains and risks the parsing above — not worth it for a log line." `usage_summary` surfaces the split as `estimated_calls` per model, so a cost report can say "N calls measured, M estimated" instead of

blending them — the same honesty rule as this project's Wilson-CI convention: never present an estimate as a measurement.

The batch integration is byte-offset bracketing, visible in `/root/sdtm_review/scripts/batch_run.py` (lines 946 and 1017): snapshot `usage_offset()` before the run, call `usage_summary(usage_offset_start)` after, and the report covers only calls made in between — with no run id threaded through the LLM layer at all.

```
# the bracketing pattern batch_run.py uses
# start = llm.usage_offset()
# ... run the batch ...
# mine = llm.usage_summary(start)   # only THIS run's calls
```

THE SAS BRIDGE. The nearest SAS habit is reading the log after a run: `NOTE: There were 78 observations read...` — accounting produced as a side effect, parsed after the fact. This ledger is that idea done deliberately: structured, append-only, one line per unit of work. The byte-offset trick is `firstobs=` thinking applied to a log file. Where the analogy misleads: a SAS log line is authoritative; here, `estimated: true` lines are `len/4` guesses, and a chars-to-tokens ratio of 4 is a folk constant, not a measurement — fine for relative comparisons across a run, wrong for invoicing.

THE PACKAGES. Stdlib only, again: `json`, `os`, `time`. The append uses a `ponytail:`-annotated shortcut — it relies on `open("a")` + a single `write()` under `O_APPEND` being atomic for lines shorter than `PIPE_BUF` (~4 KB) instead of taking a real lock, and the comment names both the ceiling and the upgrade path (`fcntl.flock`). JSONL beats SQLite here because writers never block, partial files are still parseable line-by-line, and `grep` works.

TRY IT YOURSELF. The full ledger mechanic — record, snapshot, summarize-only-mine:

```
import json, os, time, tempfile

LOG = os.path.join(tempfile.mkdtemp(), "llm_usage.jsonl")

def record_usage(backend, model, tier, pt, ct, estimated):
    line = json.dumps({"ts": time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime()),
                       "backend": backend, "model": model, "tier": tier,
                       "prompt_tokens": pt, "completion_tokens": ct,
                       "estimated": bool(estimated)})
    with open(LOG, "a", encoding="utf-8") as fh:
        fh.write(line + "\n")

def usage_offset():
    return os.path.getsize(LOG) if os.path.exists(LOG) else 0

def usage_summary(since=0):
    out = {}
    with open(LOG) as fh:
        fh.seek(since)
        for line in fh:
            rec = json.loads(line)
            m = out.setdefault(rec["model"], {"calls": 0, "prompt_tokens": 0})
            m["calls"] += 1
            m["prompt_tokens"] += rec["prompt_tokens"]
    return out

record_usage("zai", "glm-4.7", "classify", 991, 127, False)   # someone else's call
mark = usage_offset()                                          # my run starts here
record_usage("zai", "glm-5.2", "code", 1094, 143, False)      # my run's call
mine = usage_summary(since=mark)
assert list(mine) == ["glm-5.2"] and mine["glm-5.2"]["calls"] == 1
print("batch saw only its own calls:", mine)
```

LIMITS. (1) Tokens are logged; dollars are not — turning the ledger into cost requires a price table, and for the fixed-subscription backends (GLM, OAuth) a per-call dollar figure is category error anyway (the scarce resource on GLM is prompts, ~80/5hr, not tokens — per the routing comments). (2) The byte-offset bracketing assumes one batch at a time; two concurrent runs each see the other's lines. (3) CLI estimates use `len//4` uniformly, which under-counts for code-heavy text. (4) The two forks have drifted and only this one accounts — a call made through the mapper fork is invisible to any summary. All four are design ceilings, not bugs; the first ponytail comment in the file names its own upgrade path.

HOW TO IMPROVE IT. Add an optional `run_id` field to `_record_usage` (threaded like `tier` was) to replace offset bracketing; join `usage_summary` output against a small price table for the metered backend only; and backport the whole delta to `/root/sdtm_mapper/services/llm.py` — including the `extract_json` guard, which is a straight bug fix the mapper fork is still missing.

LIFT AND REUSE. This is the most liftable file in the module: a ~60-line, zero-dependency usage ledger (`_record_usage` / `usage_offset` / `usage_summary`) that bolts onto any function-shaped LLM client. The interface to keep: logging must never raise into the call path, every record carries an `estimated` flag, and aggregation takes a start offset so callers can scope reports without ids.

M3.3 `/root/sdtm_review/services/privacy.py` — the shape leaves, the values stay

Until 2026-07-17, every classify and spec call in this tool transmitted real cell values to a third-party model endpoint. The file's own header quotes what that looked like on a raw AE dataset: subject

identifiers (`701-1015`), verbatim adverse event terms (`Application Site Erythema`), outcomes (`Fatal`). Flagged July 12; still live July 17. This file is the fix — and the reason the audit trail can now print `privacy_mode: metadata-only`.

A path note first: this chapter's file exists identically in both trees at `/root/sdtm_review/services/privacy.py` and `/root/sdtm_mapper/services/privacy.py` — `diff` confirms byte-identity, so one chapter covers both. (There is no `scripts/privacy.py` in either tree; if you have seen that path in older notes, it does not exist today.)

WHAT IT DOES. It renders one column of a profiled dataset into one line of prompt text, and by default that line contains no data values — only the column name, fill percentage, optionally cardinality, and a derived "shape" like `ISO-date(YYYY-MM-DD)` or `categorical(3 levels)`. Sending real sample values is an explicit, off-by-default opt-in via the env var `AI_SEND_SAMPLE_VALUES` (accepted truthy spellings: `1`, `true`, `yes`, `on`).

WHY IT EXISTS. The product's core sales claim is "your data never leaves your environment," and the July incident proved that claim was being violated by the prompt builder itself. The design bet, stated in the header: column names are metadata, and names are "where nearly all the mapping signal lives anyway (the heuristic classifier works on names alone)." The header is equally explicit about the cost: metadata-only weakens controlled-terminology mapping — a model that cannot see `Not Recovered/not Resolved` cannot propose the exact AEOUT CT value — and it forbids asserting a number for that cost: "How much accuracy this costs is a MEASURABLE question, not a guess — scripts/accuracy_bench.py measures both modes. Do not assert a number here." That referenced script does not exist in either tree today — the header promises a measurement tool that was never built. This book follows the same rule.

INPUTS AND OUTPUTS. Input: one column dict from `profile_dataset()` (Module 1's profiler) — `{"name", "n_unique", "pct_filled", "samples"}` — plus optional row count. Output: one prompt line. Running the real functions against the real synthetic file `/root/sdtm_review/data/synthetic_v4/raw/raw_ae.csv` (78 rows, 12 columns) in default mode produces exactly these lines:

```
AE_PROFILE_LINES = """\
  SUBJECT | filled 100.0% | unique 39 | shape numeric(int, 6 digits)
  AE_NUMBER | filled 100.0% | unique 3 | shape numeric(int, 1 digits)
  AETERM_VERBATIM | filled 100.0% | unique 10 | shape categorical(10 levels)
  AE_PT | filled 100.0% | unique 10 | shape categorical(10 levels)
  AE_SOC | filled 100.0% | unique 7 | shape categorical(7 levels)
  AE_START | filled 100.0% | unique 72 | shape text(len 11)
  AE_END | filled 84.6% | unique 59 | shape text(len 11)
  AE_SEVERITY | filled 100.0% | unique 3 | shape categorical(3 levels)
  AE_SERIOUS | filled 100.0% | unique 2 | shape categorical(2 levels)
  AE_OUTCOME | filled 100.0% | unique 3 | shape categorical(3 levels)
  AE_ACTION | filled 100.0% | unique 2 | shape categorical(2 levels)
  AE_ONGOING | filled 100.0% | unique 2 | shape categorical(2 levels)
"""
```

Note what the model does not see: not one subject number, not one AE term, not one date. And note one real gap this render exposes: `AE_START` holds values like `14-APR-2024`, but `_DATE_PATTERNS` has no `DD-MMM-YYYY`-with-dashes pattern (it knows `YYYY-MM-DD`, ISO datetime, `DDMMMYYYY` and `MM/DD/YYYY`), so the dates degrade to `text(len 11)` — measured above, not assumed.

THE RUNNING EXAMPLE: THE ACTUAL AE CLASSIFY PROMPT. `services/engine.py` (`classify_domain()`, both forks) assembles the prompt in three blocks, in this order: the column profile (privacy-filtered, capped at the first 40 columns), then the top-4 heuristic candidate domains pulled from the

knowledge pack (`KP = load_pack()` at module import), then the output-format contract. For `raw_ae.csv` the heuristic scores come out `[('AE', 39), ('EX', 4), ('SU', 3), ('DM', 2)]` — measured by running `_heuristic_scores` on the real profile — so the assembled prompt is, truncated:

```
AE_CLASSIFY_PROMPT = """\
You are a CDISC SDTM expert mapping raw clinical data to SDTMIG v3.4 domains.

Raw dataset profile (78 rows, 12 columns):
  SUBJECT | filled 100.0% | shape numeric(int, 6 digits)
  AETERM_VERBATIM | filled 100.0% | shape categorical(10 levels)
  ...  (all 12 columns, rendered by privacy.describe_column)

Candidate domains (by heuristic):
- AE: Adverse Events (...)
- EX: ...   (labels/structures come from the knowledge pack)

Decide the single most likely SDTM domain. Respond ONLY with JSON:
{"domain":"XX","confidence":0-100,"rationale":"one sentence","alternatives":["YY"]}
"""
```

The spec-generation prompt (`generate_mapping_spec()`) uses the same `describe_column` lines (now with `include_unique=True`) as its SOURCE COLUMNS block, then lists only the open target variables — the ones the deterministic skeleton could not decide — with label/type/core/CT from the knowledge pack, plus a `KNOWN STUDY IDENTIFIER` block when the caller supplies a study id (so the model can never invent `STUDY123`; Module 2 covers `apply_study_id`). Token budget is handled bluntly rather than cleverly: classify caps at 40 columns and `max_tokens=400`; spec sends all columns with `max_tokens=8000`. There is no token counting on the input side — the privacy filter is what keeps prompts small, since a shape string is shorter than five sample values.

HOW IT WORKS. Three public functions. `send_sample_values()` reads the env var. `characterize(samples, n_unique, n_rows)` is a cascade of cheap tests — date regexes first, then all-int (with digit-width reporting), then all-decimal, then the categorical heuristic, then a length-range fallback:

```
def _categorical_test(n_unique, n_rows):
    # the categorical test, quoted from the body of characterize():
    # "Low cardinality relative to the data = a coded/categorical field."
    if n_unique is not None and n_unique <= 12 and (n_rows is None or n_rows > n_unique * 2):
        return "categorical(%d levels)" % n_unique
```

`describe_column(c, n_rows, include_unique)` glues name, fill%, optional cardinality, and either `shape ...` (default) or `e.g. [real values]` (opt-in) into the prompt line. `privacy_mode()` returns `"values-opt-in"` or `"metadata-only"` for the audit trail — `classify_domain()` stamps it into every classification result.

THE SAS BRIDGE. This is `PROC CONTENTS` versus `PROC PRINT`. Sending a vendor your CONTENTS output (names, types, lengths, nobs) is a routine act; sending PRINT output is a data transfer agreement. `characterize()` is CONTENTS plus a little FREQ-shaped inference (cardinality → "this is coded"). Where the analogy misleads: CONTENTS output is defined to be metadata, while `characterize` merely derives metadata from values — and the derivation itself can leak in edge cases. `categorical(2 levels)` on a column named `AE_FATAL` tells a reader something about the data; a digit-width of 6 on `SUBJECT` reveals the id scheme's shape. Metadata-only is a huge reduction of exposure, not an anonymization proof.

THE PACKAGES. `os` and `re` — nothing else. The date patterns are four compiled regexes; the whole file is 109 lines. An alternative would be a profiling library or a PII-detection model; both would add a

dependency and, ironically, another place data flows through. For a filter whose job is to keep data in, small and auditable beats capable.

TRY IT YOURSELF. A miniature `characterize` on synthetic AE values, proving no value escapes into the prompt line — and reproducing the date-format gap:

```
import re

def characterize(samples, n_unique=None, n_rows=None):
    vals = [str(s) for s in samples if str(s).strip()]
    if not vals:
        return "empty"
    if all(re.fullmatch(r"\d{4}-\d{2}-\d{2}", v) for v in vals):
        return "ISO-date(YYYY-MM-DD)"
    if all(re.fullmatch(r"-?\d+", v) for v in vals):
        return "numeric(int)"
    if n_unique is not None and n_unique <= 12 and (n_rows is None or n_rows > n_unique * 2):
        return "categorical(%d levels)" % n_unique
    lens = [len(v) for v in vals]
    return "text(len %d-%d)" % (min(lens), max(lens))

col = {"name": "AE_OUTCOME", "pct_filled": 100.0, "n_unique": 3,
       "samples": ["RECOVERED/RESOLVED", "RECOVERING/RESOLVING", "NOT RECOVERED"]}
shape = characterize(col["samples"], col["n_unique"], n_rows=78)
prompt_line = "  %s | filled %s%% | shape %s" % (col["name"], col["pct_filled"], shape)
assert "RECOVERED" not in prompt_line          # no cell value escapes
print(prompt_line)                             # AE_OUTCOME | filled 100.0% | shape categorical(3
levels)
print(characterize(["2024-04-14", "2024-05-09"]))   # ISO-date(YYYY-MM-DD)
print(characterize(["14-APR-2024", "09-MAY-2024"])) # text(...) -- the real gap
```

LIMITS. (1) The opt-in is a process-wide env var, not per-request or per-customer — one `AI_SEND_SAMPLE_VALUES=true` in a systemd unit flips every upload on the box to value-sending. (2) `characterize` sees only the profiler's first 5 unique non-blank samples, so a column that is 99% ISO dates and 1% free text can be mislabeled if the sample is clean. (3) Column names still leave the box by design; a raw file with a column literally named after a site or investigator would transmit that name. (4) The accuracy cost of metadata-only mode is deliberately unquantified here; the header points at `scripts/accuracy_bench.py` to measure it, but that script does not exist in either tree today, so the cost is not just unmeasured — the instrument to measure it is unbuilt. Treat any claimed figure as unproven.

HOW TO IMPROVE IT. Add the `DD-MMM-YYYY` (dashed SAS-style) date pattern — the running example shows real dates degrading to `text(len 11)` today; make the opt-in per-request (a flag on the upload, logged into the audit trail per dataset) rather than per-process; and have `describe_column` warn into the profile when `samples` were truncated, so `characterize`'s verdict carries its evidence size.

LIFT AND REUSE. Lift the trio as-is — `send_sample_values` / `characterize` / `describe_column` have no imports beyond `os` / `re` and no project types. The interface worth keeping is the default direction: the redacting path is the code path that runs when nothing is configured, and the revealing path requires a visible act. Any privacy layer where those are swapped will eventually ship an incident like the July one.

M3.4 `/root/sdtm_review/services/llm_mapper.py` — the model proposes, a human decides

A reviewer is staring at a residual — a raw column the deterministic mapper could not place. The diagnosis engine has already said why it is stuck ("ambiguous", "near-miss", two candidates with scores). The reviewer clicks "ask the model." What comes back is either a validated candidate mapping with a confidence number — or nothing at all, because the model hallucinated an SDTM variable and this file threw the answer away before anyone saw it.

WHAT IT DOES. `propose(plan, domain, residual)` asks an LLM for exactly one candidate mapping for one unmapped raw column, validates the answer against ground truth (the SDTM IG variable list for the domain, and the real header of the uploaded file), logs every attempt — kept or dropped — to `/root/sdtm_review/evidence/llm_mapper/proposals.jsonl`, and returns either a clean proposal dict or `None`. It exists only in the sdtm_review fork; sdtm_mapper has no equivalent.

WHY IT EXISTS. This is the review tool's one gated LLM feature (the annotation-engine state notes call it "gated LLM proposer ON"). The mapping ledger is an attributed, regulated record: a mapping decision must trace to a named human. The docstring states the contract in its first two lines: an LLM "PROPOSES mapping one residual column. It never decides one." Without this file, the alternative designs are both bad: no AI assist at all (reviewer stares at 40 near-misses by hand), or AI writing into the decision ledger (unattributable, unauditable, and — given M3.1's nondeterminism — unrepeatable).

INPUTS AND OUTPUTS. Input: the review `plan` dict (domains with their IG variable lists, datasets with their real column headers), a domain code like `"AE"`, and a `residual` dict like `{"column": "AE_SEVERITY", "file": "raw_ae.csv", "status": "near_miss", "candidates": [{"variable": "AESEV", "score": 0.82, ...}], "examples": [...]}`. Output: either `None` or a proposal dict:

```
proposal_shape = {"column": "AE_SEVERITY", "domain": "AE", "sdtm_var": "AESEV",
                  "source": "RAW", "raw_var": "AE_SEVERITY",
                  "rule": "Map severity text to AESEV controlled terminology.",
                  "confidence": 0.9, "reasoning": "...", "model": "glm-4.7"}
```

HOW IT WORKS. Five stages, in `propose()`:

1. **Prompt** — `_build_prompt()` packs the domain's full IG variable list (name, type, core, label), the deterministic diagnosis already computed (status, reason, near-miss candidates with scores), up to 5 example raw values, and the list of still-unmapped targets, then demands strict JSON against a fixed schema and ends with the load-bearing sentence: "Never invent an sdtm_var not in list above." The system prompt repeats the governance: "You PROPOSE only — named human decides." (Note the tension with M3.3: this prompt sends up to 5 real example values from the residual dict — the residual pipeline, not `describe_column`, controls what those contain.)
2. **Call** — by default through `services.llm.call_llm` at `tier="classify"` (the cheap bucket); the route can override the tier via a `model_tiers` alias. `call` is injectable, which is how `tests/test_llm_mapper.py` tests everything with no env var and no live model.
3. **Parse** — `extract_json` on the reply; malformed JSON or a non-dict is a drop, not an error.
4. **Validate against ground truth** — the decisive stage. `mapping_review.check(domain, sdtm_var, source, raw_var, header)` is the same check a human's save goes through; the header comes from `_header_for_domain(plan, domain, residual.get("file"))`, which matches on domain and file because two raw files can feed one domain (central and local labs both land in LB) and validating against a sibling file's header would wrongly keep hallucinations that happen to exist elsewhere. Any error → dropped, never surfaced.

5. **Log** — `_append()` writes one JSONL record either way, with the raw model response truncated to `RAW_TRUNCATE = 500` chars ("enough to debug the prompt without the jsonl growing unbounded on a chatty model").

One asymmetry matters: a backend failure (the whole M3.1 cascade exhausted) is logged and then **re-raised**, not returned as `None` — because "the model was down" and "the model proposed nothing sensible" are different facts, and the route counts them separately (`errors` vs `dropped`) so an outage does not read on screen as model incompetence.

WHERE THE GATE LIVES. Not here. `propose()` is callable and testable unconditionally; the on/off switch is in the route (`/root/sdtm_review/routes_ta.py`, around line 1151): unless the server environment sets `SDTM_MAPPER_LLM=1`, the endpoint refuses before this module is even imported. (On this host the flag is armed: `SDTM_MAPPER_LLM=1` sits in `/etc/clincoder/sdtm_mapper.env`, the `EnvironmentFile` of the `sdtm-mapper` systemd unit — verified today.) The route also resolves the client-facing tier alias, masks the model id via `model_tiers.alias_for`, and scrubs vendor names out of free-text fields — next chapter.

THE SAS BRIDGE. This is the statistical programmer / study lead relationship encoded in software. A junior programmer may draft a spec row; it becomes real when the lead signs it. The drop-don't-fix rule mirrors QC discipline: when a draft fails review you reject it, you don't quietly patch it — a "helpful" auto-correction of a hallucinated `AESEVCD` to `AESEV` would be the tool making a mapping decision, exactly what the contract forbids. Where the analogy misleads: a junior programmer who invents a variable name gets told, and learns. The model gets nothing back; every `propose()` is stateless, and yesterday's dropped hallucination is equally likely today. The learning loop in this project lives outside the model (prompt hardening, plus the proposals log as evidence), never inside it.

THE PACKAGES. Stdlib (`json`, `os`, `datetime`) plus two project services (`mapping_review` for ground truth, `services.llm` lazily for the call). Nothing else. The append-a-JSONL-line evidence pattern is the same one as M3.2's usage ledger.

TRY IT YOURSELF. The propose→validate→drop mechanic with a stubbed model that hallucinates plausibly:

```
import json

AE_VARS = {"AETERM", "AEDECOD", "AESEV", "AEOUT", "AESER"}   # stand-in for the IG pack
RAW_HEADER = ["SUBJECT", "AETERM_VERBATIM", "AE_SEVERITY"]    # real file columns

def check(sdtm_var, source, raw_var, header):
    errors = []
    if sdtm_var not in AE_VARS:
        errors.append(f"{sdtm_var!r} is not an AE variable in the IG")
    if source == "RAW" and raw_var not in header:
        errors.append(f"raw column {raw_var!r} not in the uploaded file")
    return errors

def fake_model(prompt):
    # A plausible-looking hallucination: AESEVCD does not exist in AE.
    return json.dumps({"sdtm_var": "AESEVCD", "source": "RAW",
                       "raw_var": "AE_SEVERITY", "confidence": 0.9})

def propose(prompt):
    obj = json.loads(fake_model(prompt))
    errors = check(obj["sdtm_var"].upper(), obj["source"], obj["raw_var"], RAW_HEADER)
    if errors:
        print("DROPPED, never shown to reviewer:", "; ".join(errors))
        return None
    return obj

assert propose("map AE_SEVERITY") is None    # hallucination filtered, not surfaced
print("reviewer sees nothing rather than something wrong")
```

LIMITS. (1) Cost per click is bounded in dollars only on the metered leg — the docstring itself counts a worst case of ~8 backend attempts (CommandCode, OAuth, GLM×3 retries, OpenRouter×3 retries) for one `propose()`. (2) Validation checks legality, not correctness: `AESEV` sourced from the wrong-but-present raw column passes `check()` and reaches the reviewer with a confident rationale — the human is the correctness gate, which is the design, but only as strong as the human. (3) Confidence is the model's self-report clamped to [0, 1]; per this project's judge-noise memory (scores of 21→19→17/35 across identical runs), treat it as a sortation hint, never a probability. (4) `proposals.jsonl` grows without rotation; 500-char truncation bounds line size, not line count.

HOW TO IMPROVE IT. Log the requested tier alias alongside the model id (the `model_tiers` ponytail note asks for exactly this); add a per-plan proposal budget so a reviewer clicking through 60 residuals cannot silently spend 60 cascades; and consider a second `propose()` draw on drop — under measured nondeterminism, one retry after a hallucination is cheap and often lands, but measure the kept-rate before claiming it helps.

LIFT AND REUSE. The liftable asset is the pattern, five stages in order: gate outside the module → prompt with an explicit closed vocabulary → strict-JSON parse where malformed means drop → ground-truth validation with the same check a human's action gets → append-only evidence log that records drops as faithfully as keeps. Any "AI suggests, human approves" feature in any domain can copy this file's skeleton with only `check()` swapped out.

M3.5 `/root/sdtm_review/services/model_tiers.py` — vendor names never reach clients

An API response is about to leave the server carrying `"model": "glm-4.7"`. To the operator that is routing detail; to a customer it is a vendor disclosure ("so my data went to z.ai?") and an anchoring problem ("why is a $18/month model reviewing my regulated submission?"). This 117-line file exists

so that what the customer sees instead is `"tier": "fast"` — and so that even the model talking about itself inside its own reasoning text gets masked.

WHAT IT DOES. It defines four client-safe aliases and maps them onto `services.llm`'s internal tier names: `TIERS = {"auto": None, "fast": "classify", "balanced": "code", "deep": "spec"}`. It converts in both directions: a client's requested alias becomes an internal tier before `llm_mapper.propose()` is called, and the model id that comes back (`"oauth:claude-haiku-4-5"`, `"glm-4.7"`) is reverse-mapped to an alias by `alias_for()` for display. A third function, `scrub()`, regex-replaces any of twelve vendor/model words in free text with "the model". Only the sdtm_review fork has this file.

WHY IT EXISTS. The module docstring names the policy: "Bhanu's policy: clients never see a model or vendor name." The subtle half of the problem is the reason `scrub()` exists: masking the `model` field is trivial, but a model can self-identify in prose — the docstring's example is a reasoning string beginning "As GLM-4.7 I think..." — so the vendor name leaks through the content, not the metadata. Without this file, every route handler would reinvent masking, inconsistently, and the anonymized-tier stance (also recorded in the project's annotation-engine notes) would be one forgotten field away from broken.

INPUTS AND OUTPUTS. `alias_for("glm-4.7")` → `"fast"`; `alias_for("")` or any unknown id → `"auto"`; `scrub("Per Claude's judgment, AESEV fits")` → `"Per the model's judgment, AESEV fits"`. The consumer is `routes_ta.py`'s propose endpoint: it validates the request's `tier` against `TIERS` (unknown alias → HTTP 400 listing valid choices), calls `llm_mapper.propose(plan, domain, res, tier=llm_tier)`, then does `p["tier"] = model_tiers.alias_for(p.pop("model", ""))` — note the `pop`: the real id is removed, not merely accompanied — and scrubs the free-text fields.

HOW IT WORKS. `alias_for()` strips the backend prefixes `call_llm` attaches (`"commandcode:"`, `"oauth:"`; z.ai and OpenRouter return bare ids), then asks, for each alias in declaration order, whether any of the four backend config dicts (`CC_MODELS`, `OAUTH_MODELS`, `ZAI_MODELS`, `MODELS` — imported lazily so `TIERS`-only callers never touch `services.llm`) would resolve that tier to this id. First match wins, and the file is honest about the resulting ambiguity in a `ponytail:` comment: CommandCode configures the same model for every tier, and glm-5.2 backs both `"code"` and `"spec"`, so `alias_for` is "a display hint reconstructed after the fact, not a promise that this exact alias was requested" — the named upgrade path is to log the requested alias rather than infer it. The file carries its own runnable check: `python3 /root/sdtm_review/services/model_tiers.py` runs the asserts in `__main__` and prints `ok`, including the deliberately pinned tie-break `alias_for("glm-5.2") == "balanced"`.

THE SAS BRIDGE. This is blinding. A blinded study report says "Treatment A"; the randomization ledger that resolves A to a compound stays inside the wall. Here the API response says `fast`; `proposals.jsonl` — server-side evidence, per `scrub()`'s docstring — keeps the original model id and unscrubbed text, because the operator must be able to audit which model said what even though the client must not. Where the analogy misleads: unblinding is ceremonial and controlled; `alias_for` is lossy and reconstructive — on a tie it cannot tell you which alias was actually requested, only a first-declared plausible one. The ledger, not the alias, is the audit truth.

THE PACKAGES. `re` and a lazy project import. The scrubber is one compiled alternation over `VENDOR_WORDS` (`glm`, `deepseek`, `grok`, `claude`, `zai`, `openrouter`, `commandcode`, `anthropic`, `opus`, `haiku`, `gemini`, `gpt`) with word boundaries and `re.IGNORECASE`.

TRY IT YOURSELF.

```
import re

TIERS = {"auto": None, "fast": "classify", "balanced": "code", "deep": "spec"}
TIER_MODELS = {"classify": {"glm-4.7"}, "code": {"glm-5.2"}, "spec": {"glm-5.2"}}

def alias_for(model_id):
    bare = model_id.split(":", 1)[-1] if model_id.startswith(("oauth:",)) else model_id
    for alias, tier in TIERS.items():
        if tier and bare in TIER_MODELS[tier]:
            return alias          # first declared match wins on ties
    return "auto"

VENDOR_RE = re.compile(r"\b(?:glm|deepseek|claude|opus|gemini|gpt)\b", re.IGNORECASE)

def scrub(text):
    return VENDOR_RE.sub("the model", text) if text else text

assert alias_for("glm-4.7") == "fast"
assert alias_for("glm-5.2") == "balanced"     # code and spec tie; first wins
assert alias_for("mystery-model") == "auto"
assert "Claude" not in scrub("Per Claude's judgment, AESEV maps here")
print(scrub("As GLM-4.7 I think AE_SEVERITY -> AESEV"))
```

LIMITS. (1) Word-boundary scrubbing has edge artifacts — the file's own `__main__` pins one: `"As GLM-4.7 I think..."` scrubs to `"As the model-4.7 I think..."`, leaving a version number dangling. (2) The word list is an enumeration; a new backend ("qwen", "mistral") leaks until someone adds the word. (3) `alias_for` reads live config, so changing `ZAI_STRONG_MODEL` retroactively changes how old stored model ids would alias. (4) Scrubbing catches names, not fingerprints — distinctive phrasing can still hint at a vendor; unmeasured and probably unmeasurable.

HOW TO IMPROVE IT. Do what the ponytail comment says: record the requested alias at request time and stop inferring. Derive `VENDOR_WORDS` from the model ids actually present in the four config dicts plus a manual extras list, so a newly configured backend cannot leak by omission.

LIFT AND REUSE. The generic asset is "public capability names, private implementation names" with a two-layer mask: map the id field, scrub the prose. The right interface is exactly these three names — a `TIERS` dict, `alias_for(id) -> alias`, `scrub(text) -> text` — plus the rule that server-side evidence keeps originals.

M3.6 `/root/sdtm_review/services/apikey.py` + `/root/sdtm_mapper/api_keys.json` — a JSON file is the customer database

Six days after this gate was written, six `pro`-tier keys existed in `api_keys.json` — issued to real evaluators. There is no database server, no auth framework, no JWT library anywhere in it: the entire commercial access layer for the mapper's API is 60 lines of stdlib Python, one JSON file of keys, and one JSON file of daily counters.

WHAT IT DOES. `apikey.py` (byte-identical in both trees — `diff` confirms; both copies point at the same key files under `/root/sdtm_mapper/`) is a Flask `before_request` gate for the six proprietary compute endpoints (`/api/profile`, `/api/spec`, `/api/code`, `/api/execute`, `/api/validate`, `/api/auto`). It checks an `X-API-Key` header against `api_keys.json`, enforces a per-key daily quota, and — crucially — does nothing at all unless the env var `API_KEY_ENFORCE=1` is set. Health, index and demo endpoints are never gated.

WHY IT EXISTS. The docstring calls it "IP Layer 2A": the moment the mapper became a product demo on a public URL, its expensive endpoints (each one triggers the M3.1 cascade — real compute, real money) needed a gate that could be armed without breaking the live demo. "SAFE BY DEFAULT: enforcement is OFF unless env API_KEY_ENFORCE=1, so deploying this never breaks the live UI." That deploy-dark-then-arm pattern is the file's main design idea. A terminology note for readers arriving with the phrase "BYOK": these are not bring-your-own-LLM-key credentials — the LLM keys are the operator's (M3.1). These are keys the tool issues to customers for API access. The BYOK stance this project talks about elsewhere (privacy.py's header, the Enterprise Build Plan) is the adjacent idea — AI off by default, customer opts in under their own agreements — but this file is plain API metering.

INPUTS AND OUTPUTS. `issue_key("cro@x.com", "pro")` returns a fresh key and persists its record. The stored shape, redacted to shape (never reproduce real keys):

```
# api_keys.json shape as a Python literal (JSON false -> False); key redacted
api_keys_shape = {
    "sdtmm_53...ab77": {
        "email": "bh***",
        "tier": "pro",
        "daily_limit": 1000,
        "created": "2026-07-26T00:08:53",
        "revoked": False,
    }
}
```

Six such entries exist today, all `pro`. `gate()` returns `None` to allow, or a `(jsonify(...), status)` tuple: 401 for missing/invalid/revoked keys, 429 when the day's counter reaches the tier limit (`TIER_LIMIT = {"free": 50, "pro": 1000, "ent": 100000000}`).

HOW IT WORKS. Both apps wire it identically (`/root/sdtm_mapper/app.py` line 37, `/root/sdtm_review/app.py` line 36):

```
from services.apikey import gate as _apikey_gate

@app.before_request
def _apikey_before():
    return _apikey_gate()
```

Flask's contract makes this clean: a `before_request` hook returning `None` lets the request proceed; returning a response short-circuits it. Inside `gate()`: not enforcing → allow; path not in `PROTECTED` → allow; then key lookup (the raw key string is the dict key — an O(1) plaintext lookup), then the quota read-increment-write against `api_quota.json` keyed by key and `YYYY-MM-DD`. Key generation in `issue_key()` is `"sdtmm_" + sha256(email + timestamp)[:40]` — the hash is used as a random-looking generator, not as storage protection; what lands in the file is the usable secret itself. Revocation is manual and instant: edit the JSON, set `"revoked": true`.

Two facts about live state, checked on this host today: `API_KEY_ENFORCE` appears nowhere in the `sdtm-mapper` unit's environment (`systemctl show sdtm-mapper -p Environment` shows only `SDTM_PORT`/`HOME`, and `/etc/clincoder/sdtm_mapper.env` — its `EnvironmentFile` — does not set it), so the gate is dormant; and the quota file (`api_quota.json` under `/root/sdtm_mapper/`) does not exist yet — consistent, since the quota file is only written on a gated request and enforcement has never been armed.

THE SAS BRIDGE. Closest analogue: the SAS metadata server's user table with a per-user job quota — except deliberately without the server. For a SAS programmer the shock is that this is allowed to be a flat file: at six customers and single-digit requests per second, a JSON file read per request is simpler, auditable with `cat`, and versionable. The analogy misleads on concurrency: a metadata server

serializes writes; here two simultaneous requests can both read `used=49`, both write `used=50`, and a free-tier customer gets 51 calls. At this scale that is a rounding error, not a security hole — but it is a real race.

THE PACKAGES. `flask` (already the app framework — `request`, `jsonify`), plus stdlib `json`, `time`, `hashlib`, `pathlib`. No auth library, no ORM. The swap cost runs one direction: moving to hashed-at-rest keys or SQLite is an afternoon; the flat file cannot survive real traffic or a multi-worker deployment with per-request writes.

TRY IT YOURSELF. Issue, gate, quota — the whole lifecycle against temp files:

```
import hashlib, json, tempfile, time
from pathlib import Path

d = Path(tempfile.mkdtemp())
KEYS, QUOTA = d / "api_keys.json", d / "api_quota.json"
TIER_LIMIT = {"free": 50, "pro": 1000}

def issue_key(email, tier="free"):
    keys = json.loads(KEYS.read_text()) if KEYS.exists() else {}
    raw = "sdtmm_" + hashlib.sha256((email + str(time.time())).encode()).hexdigest()[:40]
    keys[raw] = {"email": email, "tier": tier,
                 "daily_limit": TIER_LIMIT.get(tier, 50), "revoked": False}
    KEYS.write_text(json.dumps(keys, indent=2))
    return raw

def gate(key):
    meta = (json.loads(KEYS.read_text()) if KEYS.exists() else {}).get(key)
    if not meta or meta.get("revoked"):
        return 401
    day = time.strftime("%Y-%m-%d")
    q = json.loads(QUOTA.read_text()) if QUOTA.exists() else {}
    used = q.get(key, {}).get(day, 0)
    if used >= meta["daily_limit"]:
        return 429
    q.setdefault(key, {})[day] = used + 1
    QUOTA.write_text(json.dumps(q))
    return None    # allow

k = issue_key("cro@example.com", "free")
assert gate(k) is None and gate("sdtmm_bogus") == 401
print("issued", k[:10] + "..." + k[-4:], "| first call allowed, bad key 401")
```

LIMITS — THE HONEST SECURITY POSTURE. (1) Keys are stored in plaintext; anyone who can read `/root/sdtm_mapper/api_keys.json` holds every customer credential. Standard practice is to store a hash and compare on lookup. (2) The quota write is `read → modify → write_text` with no lock — the race above; and a failed quota write is swallowed (`except Exception: pass`), which fails open: the call proceeds uncounted. Fail-open on metering is defensible for availability; know that it is a choice. (3) Matching is exact string equality on the header, not `hmac.compare_digest` — a timing-attack purist would object; against 40 hex chars over HTTPS this is theoretical, but it is a one-line fix. (4) The gate protects by exact path only; a new expensive endpoint is unprotected until someone remembers `PROTECTED`. (5) Enforcement status is invisible from outside — nothing logs "gate armed/dormant" at startup; you check the env, or you probe.

HOW TO IMPROVE IT. Arm-and-log: on import, log whether enforcement is active. Store `sha256(key)` as the dict key and hash the incoming header before lookup — this costs nothing and de-fangs the file-read threat. Use `fcntl.flock` around the quota update (the same upgrade path M3.2's ponytail comment names for its ledger).

LIFT AND REUSE. The liftable pattern is the deploy-dark gate: ship the enforcement code with enforcement off, verify nothing breaks, then arm with one env var — the same OFF-by-default discipline as `AI_SEND_SAMPLE_VALUES` (M3.3) and `SDTM_MAPPER_LLM` (M3.4). The interface: a `gate() -> None | (response, code)` hook, a `PROTECTED` set, and key/quota storage behind two functions you can later re-point at SQLite without touching the gate logic.

What this module still owes you

BACKPORTED FIXES IT HASN'T MADE. The mapper fork still carries the bare-`min()` crash in `extract_json`, the stale `call_llm` docstring, and no usage accounting. Every claim in M3.2 about the delta is a to-do list for `/root/sdtm_mapper/services/llm.py`.

NUMBERS IT REFUSES TO INVENT. The accuracy cost of metadata-only privacy mode (the bench script exists; no measured result was found written down). The kept-vs-dropped rate of `llm_mapper.propose` in practice (`proposals.jsonl` holds the evidence; nobody has aggregated it). The real chars-per-token ratio behind every `estimated: true` ledger line. Each is measurable with what is already on disk.

A DETERMINISM STORY IT CAN ONLY CONTAIN, NOT SOLVE. Temperature 0 is sent on every call and does not deliver reproducibility on the backend in use — measured (three runs, three different programs; 7/9 then 2/9 on identical prompts). The mitigations you have seen — execute-and-grade gates, drop-don't-trust validation, evidence logs of every attempt — manage the draw; nothing in this codebase can make a model call into a macro expansion. Any rebuild of this layer must start from that fact, not from the header comment that wishes it away.

THE THINKING-MODE SCAR'S FORENSICS. The incident (empty content, 5–13× slower) predates the current trees; no thinking-mode configuration survives anywhere in either repo to show you. What survives is the guard discipline it taught: every backend wrapper treats empty output and refusal prose as raised errors. If you rebuild with a thinking-capable model, budget for hidden-reasoning tokens explicitly and assert non-empty visible content — that is the failure class, and it will come back with different config knobs.

SECURITY DEBT, NAMED. Plaintext API keys, a hardcoded z.ai fallback key in source, a cross-app key-scrape fallback that required its own kill switch, and a fail-open quota counter. All are documented above with fixes; none is fixed.

Module 4 — Code generation and execution

This module covers the moment the pipeline stops deciding and starts doing: turning a mapping spec into a runnable SAS, R, and Python program, and then actually running the Python and R arms against real raw data, inside a sandbox, with a self-repair loop. It documents two files that share a name but not a body — `sdtm_mapper/services/codegen.py` (199 lines) and `sdtm_review/services/codegen.py` (501 lines) — plus the executor, the sandbox, and the two study-level orchestration modules of `sdtm_review`. Three distinctions anchor everything here: `golden/` holds human-blessed reference programs while `output/<hash>/` holds machine leftovers from past runs; `*_map.py` is the program the tool writes for you while `*_exec.py` is the program it ran; and the two `codegen.py` files are different code that must never be conflated.

For the SAS/R reader, the single most important frame: a generated mapping program is **NOT** a macro expansion. `%macro ae_map;` expands the same way every time; an LLM at temperature 0.0 does not (measured on this stack — see M4.7's `_archive` story). Every generation is a fresh draw that can be wrong in a novel way, which is why half of this module is not generation at all but verification and containment of what was generated.

M4.1 `/root/sdtm_mapper/services/codegen.py` — prompts an LLM into three programs

A mapping spec for AE exists — 24 rows, each saying things like `AESEV | IT.AESEV | Standardized | Char | Map Mild/Moderate/Severe Adverse Event to MILD/MODERATE/SEVERE | AESEV`. Nobody has written a program yet. This file's job is to get one program in each of three languages out of a language model, from that spec alone, and hand back text that will actually run when saved to a `.sas`, `.R`, or `.py` file.

WHAT IT DOES. It builds one large prompt per language — spec table plus a block of SDTM conformance rules — sends it to the LLM cascade (`services.llm.call_llm`), and strips the response down to bare code. `generate_all_code` runs all three languages, in parallel threads by default, and returns a dict of `{"sas": ..., "r": ..., "python": ..., "models": {...}}`.

WHY IT EXISTS. Writing a mapping program by hand is the job this whole product automates. But a naive "write me a SAS program for this spec" prompt produces programs that pass a glance and fail a run: blanks become `NA`, `Not Submitted` columns get named in a `transmute()` without being created, `STUDYID` gets assigned a literal. The bulk of this file — `_conformance_rules` and `_READ_RULE` — is the accumulated scar tissue that prevents those specific failures. Without it, a documented batch probe scored 0/2 domains (see the `sdtm_mapper` CLAUDE.md's 2026-07-20 postmortem).

INPUTS AND OUTPUTS. Input: `spec` (a list of dicts with keys `target`, `source`, `method`, `type`, `logic`, `ct`), `domain` (e.g. `"AE"`), `raw_name` (e.g. `"ae_raw.csv"`), optional `model` override. Output per generator: a `(code, model_used)` tuple — the code is a string of source in the target language, and `model_used` is whichever backend actually served the call, recorded for provenance because `"model": null` in a batch manifest "made it impossible to tell which backend produced a given artifact" (comment at line 109). For AE, the Python output is a program shaped like the golden `AE_map.py` shown in M4.5: read `ae_raw.csv` all-string, build 24 columns, sort, assign `AESEQ`, write `ae.csv` then `ae.xpt`.

HOW IT WORKS. The pipeline per language is: `_spec_table(spec)` flattens the spec into a pipe-delimited table for the prompt; `_conformance_rules(domain, lang)` builds the rules block; `generate_sas`/`generate_r`/`generate_python` interpolate both into a language-specific prompt and call `call_llm(prompt, max_tokens=6000, temperature=0.0, tier="code", override_model=model)`;

`_strip(out)` extracts code from the response. The rules block is domain-aware: it loads `knowledge/sdtm_ig34.json` at import time into `_KP`, pulls the domain's Required/Expected variables and natural key, and `_has_seq(domain)` decides whether to order or forbid a `--SEQ` derivation. Rule 0 is language-specific by construction:

```
def _conformance_rules(domain, lang):
    info = _KP["domains"].get(domain, {})
    keys = info.get("keys", [])
    req = [v["name"] for v in info.get("variables", []) if v.get("core") == "Req"]
```

The `lang` parameter ("py" | "r" | "sas") "is required rather than defaulted so a new generator cannot silently inherit another language's idiom, which is the bug this signature was added to fix" — before 2026-07-20 the pandas phrasing of rule 0 was injected into the R prompt too, and R fell back to `readr` defaults that turn `""` into `NA`. `_strip` deserves attention: it takes the longest fenced code block anywhere in the response, and if there are no fences, scans forward past prose lines until one matches a regex of plausible code starts (`libname`, `proc`, `library(`, `import`, `<-` …). The old version only stripped a fence at position zero, so "Let me write the R program:" survived into the customer-facing `.R` file and `Rscript` died with `unexpected symbol in "Let me"`. Finally `generate_all_code` fans out over a `ThreadPoolExecutor` — `max_workers` is 1 when `SDTM_USE_COMMANDCODE=1` (the gateway rate-limits concurrency) and 3 otherwise, overridable via `SDTM_CODEGEN_WORKERS`.

THE PACKAGES. Only stdlib (`os`, `json`, `re`, `concurrent.futures`) plus the project's own `services.llm`. There is deliberately no templating engine: the "template" is an f-string prompt, and the variable part is written by the model. The alternative design — Jinja2 templates emitting deterministic code with the LLM filling only the derivation logic — would be more reproducible but was not chosen; swapping to it would mean rewriting this file entirely rather than adjusting it, because here the LLM writes the whole program, control flow included.

THE SAS BRIDGE. This is the point where the macro analogy breaks. In SAS, `%ae_map(domain=AE)` resolves text deterministically; the risk surface is your macro logic, audited once. Here, each of the three programs is written fresh by a model each time, and the prompt is the only control. The rules block is best read as a macro whose expansion is probabilistic: rule 7 ("create every `Not Submitted` column before any select/keep step") is not a nicety, it is the fix for a real `object 'RFXSTDTC' not found` crash. `tests/test_codegen_prompt.py` (in `sdtm_review`) treats the prompt itself as code under test — 21 offline assertions that no pandas idiom reaches the R prompt, that STUDYID is never called a constant, and so on.

TRY IT YOURSELF. A miniature of `_spec_table` + prompt assembly, no LLM needed:

```
spec = [
    {"target": "STUDYID", "source": "STUDY", "method": "Direct"},
    {"target": "AESEV", "source": "IT.AESEV", "method": "Standardized",
     "logic": "Mild Adverse Event -> MILD"},
    {"target": "AESPID", "source": "", "method": "Not Submitted"},
]

def spec_table(spec):
    rows = ["target|source|method|logic"]
    for r in spec:
        rows.append("|".join([r.get("target", ""), r.get("source", ""),
                              r.get("method", ""), r.get("logic", "")]))
    return "\n".join(rows)

prompt = (
    "Write a pandas program mapping ae_raw.csv to SDTM AE.\n"
    + spec_table(spec)
    + "\nRULE: create every 'Not Submitted' column as '' before selecting it."
)
assert "AESPID||Not Submitted" in prompt
print(prompt)
```

LIMITS. Nothing here verifies that the returned text is a program — that check exists only in the `sdtm_review` fork (M4.2). A refusal string saved via this file goes straight into a customer deliverable. `_strip`'s longest-fence heuristic can pick the wrong block if the model emits two fenced programs. The SAS output is never executed anywhere on this host (no SAS runtime exists), so SAS defects are invisible until a customer runs the file. And temperature 0.0 does not buy determinism: identical calls return different programs (documented in memory `llm-nondeterminism-at-temperature-zero`, measured on glm-5.2).

HOW TO IMPROVE IT. Backport `looks_like_code` and `_call_checked` from the review fork — the mapper fork still delivers unverified text. Second, move the shared rule text into one file imported by both forks; today the `_READ_RULE` and `_conformance_rules` bodies are near-duplicated and have already drifted (the review fork's rule 1 has a completely rewritten STUDYID paragraph).

LIFT AND REUSE. The reusable shape is spec → prompt → strip, with the rules block as pluggable per-language text. If you lift it, keep the `lang`-required signature and carry the prompt tests with it — the prompt is the product here, and untested prompt reuse across languages is the documented 0/2 failure mode.

M4.2 `/root/sdtm_review/services/codegen.py` — the hardened fork that verifies output

On 2026-07-22, a real CDISCPILOT01 run delivered a `.sas` file whose entire content was "The request was rejected because it was considered high risk", and executed "Let me check if there's any raw data ... first." as Python. Nothing downstream reads SAS, so the first defect would have shipped to a customer silently. This file is what the mapper-fork `codegen.py` became after that class of incident: same three generators, wrapped in verification, retry, trap lists, and a deterministic post-processing epilogue.

WHAT IT DOES. Everything M4.1 does — spec table, conformance rules, three language generators, `_strip`, parallel `generate_all_code` — plus four additions: it checks that what came back is a program (`looks_like_code`), retries once with a stricter prompt when it is not (`_call_checked`), injects observed source values and legal CT terms into the prompt (`_value_domains`), and emits a harness-owned R epilogue that deterministically normalises serialization defects (`deterministic_epilogue`).

WHY IT EXISTS. The mapper fork trusts the model; this fork has receipts proving it should not. Each addition maps to a named incident: `_LANG_TRAPS` exists because "2 of 9 R programs died" in the first executed Cardiology run, both on defects now listed there; `_call_checked` exists because "3 of 5 datasets came back as conversation"; `_value_domains` exists because a spec told the model to consult "the sponsor lab-test dictionary" that does not exist, so the model invented one and "1,500 of 3,000 LBTESTCD cells came out blank" (documented in `ta_programs.build_programs`); the epilogue exists because `readr::write_csv`'s `na = "NA"` default "put 'NA' into roughly 79,000 cells across all nine domains".

INPUTS AND OUTPUTS. The generator signatures gain one parameter over the mapper fork: `generate_python(spec, domain, raw_name, model=None, raw_path=None)`. When `raw_path` points at the actual raw CSV (e.g. `/root/sdtm_review/corpus/tierB_robustness/<TA>/ae_raw.csv` — `ta_programs` resolves it under `ta_pack.CORPUS_ROOT`), `_value_domains` asks `services.value_domains.prompt_block` for the observed distinct values per column so the prompt can state them instead of leaving the model to guess. When `raw_path` is `None` the block is empty and — deliberately — logged: "codegen for %s got no raw_path: value domains NOT injected". Output shape is unchanged: `(code, model_used)` per generator; `generate_all_code` returns the same dict as M4.1. `looks_like_code(src, lang)` returns `(ok, reason)` tuples such as `(False, "does not parse as Python: invalid syntax (line 1)")`.

HOW IT WORKS. The delta over M4.1, in the order a call flows:

1. `_spec_table` now emits a `label` column. Rationale in its docstring: the IG's exact variable label ("Study Identifier") was already held by `spec_bridge` and withheld from the model, so the model regenerated ~37 labels per program from memory — "37 free opportunities per program to differ from the IG".
2. `_KP` comes from `services.knowledge.load_pack()` instead of reading `knowledge/sdtm_ig34.json` directly. The two forks' packs disagree — per the `sdtm_review` CLAUDE.md, CO has a `COSEQ` variable in this fork's pack and not in the mapper's, so `_has_seq("CO")` returns different answers in the two codebases. Assume nothing about domain metadata across forks.
3. `_conformance_rules` appends `_LANG_TRAPS[lang]` — concrete, named one-character slips ("write `TRUE ~ \"\"`, never `TRUE <- \"\"` inside `case_when()`"), plus R4: "NEVER INVENT REFERENCE DATA", the anti-fabrication rule, and R6: always `write_csv(..., na = "")`.
4. Every generator routes through `_call_checked(prompt, lang, model=None, max_tokens=6000)`:

```
def _call_checked(prompt, lang, model=None, max_tokens=6000):
    out, model_used = call_llm(prompt, max_tokens=max_tokens, temperature=0.0,
                               tier="code", override_model=model)
    code = _strip(out)
    ok, reason = looks_like_code(code, lang)
    if ok:
        return code, model_used
    retry = prompt + _STRICT_RETRY.format(reason=reason, lang=lang.upper())
```

The retry modifies the prompt — a bare re-ask at temperature 0.0 "returns an identical non-answer". If both attempts fail, it returns the second response anyway and lets the caller's own check reject it, because a silent best-effort "is exactly the behaviour that put a refusal string in a .sas file". 5. `looks_like_code` is deterministic and conservative: empty or <120 chars fails; text starting with any `_NOT_CODE_MARKERS` phrase ("request was rejected", "let me check", "as an ai", ...) fails; for Python the source must `compile(text, "<generated>", "exec")` — exact, and safe because `compile` does not execute; for R/SAS it only requires one token from `_CODE_TOKENS` (e.g. `library(`, `run;`). It answers "did the backend return prose or a refusal", never "is this program correct". 6.

`deterministic_epilogue(spec, domain)` builds an R `local({...})` block that the harness appends after the generated R program (callers do this; the function only returns the text). It re-reads the emitted `<domain>.csv`, maps `NA`/missing to `""`, fixes controlled-terminology case via a codelist inlined from `services.ct` (`"Asian"` → `"ASIAN"`, values not in the codelist left visibly wrong), nulls `N` in NY-codelist `--FL` flags, and writes `<domain>_postprocess.json` with per-rule change counts — "an invisible correction is indistinguishable from model luck". Documented caveat: it maps the literal `"NA"` to `""` everywhere, so a study that genuinely submits "NA" (Not Applicable is a real NY term) must not use it.

THE PACKAGES. Same as M4.1 plus `logging` and deferred imports of `services.value_domains` and `services.ct` (deferred so "a prompt extra must never break codegen"). The Python-validity check is the stdlib `compile` builtin — zero dependencies, and stronger than any regex. An AST-based check (`ast.parse`) would be equivalent; a linter would be overkill for the question being asked.

THE SAS BRIDGE. `looks_like_code` is the closest thing this pipeline has to a SAS shop's "did the log contain ERROR:" scan — except it runs before execution, on the program text, because for SAS there will never be a log: no SAS runtime exists on this host, so the SAS arm's only quality gates are this token check and the shared prompt. That asymmetry — Python checked by compilation, R checked by execution (M4.4), SAS checked by four substrings — is stated honestly in `ta_programs.NOT_EXECUTED_NOTE` (M4.7).

TRY IT YOURSELF. A miniature `looks_like_code`:

```
REFUSALS = ("let me check", "i cannot", "request was rejected", "as an ai")

def looks_like_code(text, lang="py"):
    text = (text or "").strip()
    if len(text) < 40:
        return False, "too short to be a program"
    low = text.lower()
    for marker in REFUSALS:
        if low.startswith(marker):
            return False, f"reads as a reply, not code ({marker!r})"
    if lang == "py":
        try:
            compile(text, "<generated>", "exec")   # parses, never executes
        except SyntaxError as exc:
            return False, f"not Python: {exc.msg} (line {exc.lineno})"
    return True, ""

assert looks_like_code("import pandas as pd\ndf = pd.DataFrame()\n" * 3)[0]
assert not looks_like_code("Let me check if the raw CSV exists first." * 3)[0]
print("both verdicts correct")
```

LIMITS. `looks_like_code` for R and SAS is four substrings — a refusal that happens to contain `library(` passes, and prose appended after valid code passes for all three languages. The retry costs one extra model call and still fails ~when the backend is having a policy day; there is no third attempt by design. `_value_domains` returning `""` on two very different conditions (no raw path vs. broken CT package) is only distinguishable via logs. `deterministic_epilogue` exists for R only; the Python arm has no equivalent epilogue, its serialization is trusted to the prompt plus `executor.run_python_program`'s XPT fallback. And nothing in this file executes anything — correctness is established one layer up.

HOW TO IMPROVE IT. Give R a real parse check: `Rscript -e 'parse(file=...)'` is cheap, available (R 4.5.2 is installed), and would upgrade the R arm from token-sniffing to the same standard Python enjoys.

Extend `deterministic_epilogue` with a Python twin so both executed arms get the same serialization floor.

LIFT AND REUSE. The lift-worthy asset is the check-and-retry envelope: `_call_checked` + `looks_like_code` + `_STRICT_RETRY` is a generic pattern for any "LLM must return an artifact of type X" call. Genericise by parameterising the validator and the refusal markers; keep the two-attempts-then-surface-failure contract, and keep the rule that a failed second attempt is returned visibly broken rather than silently absent.

M4.3 `/root/sdtm_review/services/sandbox.py` — a jail for code nobody reviewed

Until 2026-07-20, the programs this pipeline generated ran as root on a shared VPS, with a 150-second timeout as the only control — the module's own docstring records that "R's `system()` returned uid=0". The program being run was written by a language model from a user-supplied CSV. In a SAS shop's terms: imagine a portal where anyone can upload a dataset, a robot writes a SAS program about it that nobody reviews, and the server runs that program under the sysadmin account. That is what existed. This file is the fix.

WHAT IT DOES. Wraps any command in a `bubblewrap` (`bwrap`) container with no network, no shared process/IPC namespaces, the OS runtimes mounted read-only, and exactly one writable directory: the job's own workdir. A second, independent layer (`systemd-run` cgroup scopes) caps process count and memory. If containment is unavailable, it **refuses to run** rather than degrading — `SandboxUnavailable` is raised unless `SDTM_ALLOW_UNSANDBOXED=1` is deliberately set.

WHY IT EXISTS. The threat model in the docstring is concrete and local: a hostile generated program would read `/root/clinassist_v2/config.py` (which held the live OpenRouter key behind a real ~$4.4/day burn incident), or `.env` files and SSH keys; write outside its job dir for persistence or evidence-tampering; exfiltrate over the network; or fork-bomb a shared box. The attacker controls the contents of the program (via what the model does with the uploaded data), not the argv — "so argv hardening is irrelevant; the containment has to be at the OS boundary."

INPUTS AND OUTPUTS. `run(argv, workdir, timeout=150, limits=None, extra_ro=())` takes a command list like `["Rscript", "/path/AE_map.R"]` and a job directory. It returns a dict — `returncode`, `stdout`, `stderr`, `limits`, and two honesty flags: `sandboxed` (was bwrap containment active) and `resource_limited` (was the cgroup layer active). Both flags are in the payload deliberately: "an uncontained run must never be silently indistinguishable from a contained one." For the AE running example, the executor calls `sandbox.run(["python3", ".../AE_exec.py"], workdir=out_dir, timeout=150)` and gets back `{"returncode": 0, "stdout": "Wrote SDTM AE CSV: ae.csv (1191 records)...", "sandboxed": True, ...}` (row count illustrative — depends on the raw file).

HOW IT WORKS. Four functions in call order. `available()` probes rather than trusting `shutil.which("bwrap")` — it actually executes `/usr/bin/true` inside a fully built sandbox, because a bwrap binary on a kernel with user namespaces disabled "would otherwise look usable and fail at call time", and because an earlier probe that bound only `/usr` then invoked `/bin/true` reported the sandbox broken on a host where it worked ("Probe what you actually run"). `build_argv(argv, workdir, extra_ro=(), cfg=None)` assembles the jail:

```
cmd = [
    "bwrap",
    "--unshare-all",          # net, pid, ipc, uts, cgroup, user
    "--die-with-parent",      # no orphan survives the request
    "--new-session",          # detach from the controlling tty (TIOCSTI)
    "--proc", "/proc",
    "--dev", "/dev",
    "--size", tsz, "--tmpfs", "/tmp",
]   # excerpt -- --setenv lines, the ro-binds and the workdir bind follow
```

then `--ro-bind`s each existing path in `_RO_PATHS` (`/usr`, `/lib`, `/lib64`, `/bin`, `/sbin`, `/etc/R`, `/etc/ssl/certs`, R's site-library, ...), binds `workdir` read-write, and `--chdir`s into it. Note the in-sandbox tmpfs is size-capped (`tmpfs_size_bytes`, 256 MiB) because an unsized tmpfs "defaults to ~half of RAM" — a memory exhaustion route through `/tmp`. R is additionally muzzled via `R_LIBS_USER=/nonexistent` etc. so it can neither load a user library outside the jail nor emit confusing CRAN-mirror errors. `run()` then stacks the layers: if `_systemd_run_available()`, the whole bwrap command is prefixed with `systemd-run --scope -p TasksMax=64 -p MemoryMax=2G -p MemorySwapMax=0`; per-process rlimits (`RLIMIT_AS` 2 GiB, `RLIMIT_CPU` 120 s, `RLIMIT_FSIZE` 512 MiB, `RLIMIT_NOFILE` 256, core dumps off) are applied via `preexec_fn=_limits(cfg)` in the forked child; the wall clock is `subprocess.run(..., timeout=timeout)`.

The most instructive comment in the file is on `DEFAULTS["tasks_max"]`: `setrlimit(RLIMIT_NPROC)` **does not work here**, because the sandbox enters a new user namespace holding full capabilities and "the kernel skips the per-uid nproc check for a task with CAP_SYS_RESOURCE. An adversarial probe forked 250+ processes straight through a 64-proc rlimit. A pids-controller cgroup is the only thing that actually holds." That is a measured result, not a design guess — the fork-bomb ceiling lives in the cgroup, and the two layers fail differently: bwrap missing is fail-closed (raise), systemd-run missing is fail-open with `resource_limited: False` in the payload, "because a denial-of-service limit is lower severity than a containment breach and must never be silently claimed when absent."

THE PACKAGES. Stdlib only (`os`, `resource`, `shutil`, `subprocess`) plus two host binaries, both present today on this box (verified: `/usr/bin/bwrap`, `/usr/bin/systemd-run`, and `/usr/bin/Rscript` for the payloads). Alternatives: Docker/Podman give stronger isolation but require a daemon and image management, heavy for per-request 150-second jobs; `nsjail` is a close substitute for bwrap; seccomp filters would be a deeper layer bwrap does not add here. Swapping to containers means rewriting `build_argv` and re-answering the probe question, but the `run()` return contract could survive unchanged.

THE SAS BRIDGE. There is no equivalent of this in a SAS shop, and that absence is the point of the bridge: when a colleague hands you a program, institutional trust and code review stand in for containment. Here the "colleague" is a model that has, on record, invented subject IDs and hardcoded reference dates (M4.7's `_FABRICATION` list). Running its program is running an unreviewed stranger's program, every time — so the review step is replaced by an OS boundary.

TRY IT YOURSELF. The rlimit layer, without bwrap — cap a child's address space and watch a big allocation die while the parent survives:

```
import resource, subprocess, sys

def cap_memory():
    lim = 256 * 1024 * 1024                      # 256 MiB address space
    resource.setrlimit(resource.RLIMIT_AS, (lim, lim))
    resource.setrlimit(resource.RLIMIT_CPU, (5, 5))

hog = "x = bytearray(512 * 1024 * 1024); print('allocated')"
p = subprocess.run([sys.executable, "-c", hog], capture_output=True,
                   text=True, timeout=30, preexec_fn=cap_memory)
print("returncode:", p.returncode)               # non-zero: MemoryError
assert p.returncode != 0 and "allocated" not in p.stdout
small = subprocess.run([sys.executable, "-c", "print('fits')"],
                       capture_output=True, text=True, preexec_fn=cap_memory)
assert small.stdout.strip() == "fits"
print("cap held for the hog, small job unaffected")
```

LIMITS. The docstring's own honesty list, which should be quoted in any reuse: it does not defend against "a kernel escape from a user namespace. This is containment, not a VM"; the generated program "can still corrupt its own outputs" inside the writable workdir — grading the artifact is a different layer's job; and "the Flask service still running as root" is an acknowledged separate, larger fix — the sandbox shrinks the blast radius of generated code, it does not make the host safe. Two more gaps worth naming: the read-only binds still expose everything under `/usr` and `/etc/ssl/certs` to reading, so a generated program can fingerprint the host even if it cannot phone home; and `SDTM_ALLOW_UNSANDBOXED=1` is an environment variable — anything that can set the service's environment can disable the jail (it is loud in the payload, but loud is not prevented).

HOW TO IMPROVE IT. Drop root: run the sandboxed child under a dedicated uid even inside the userns, and migrate the Flask service off root so the "separate, larger fix" stops being deferred. Add a seccomp profile (bwrap supports `--seccomp`) to cut the syscall surface a kernel escape would need. Consider making `resource_limited: False` a hard failure in production, since systemd-run is expected present there.

LIFT AND REUSE. This file is already generic — nothing in it knows about SDTM. The right interface is exactly its current one: `run(argv, workdir, timeout, limits, extra_ro) -> dict` with the two honesty flags. If you lift one idea only, lift fail-closed with a loud override: a sandbox that silently degrades "is worse than no sandbox, because the caller believes it is protected."

M4.4 `/root/sdtm_review/services/executor.py` — runs, repairs, and compares generated programs

Codegen has handed back a Python string and an R string that look like AE mapping programs. Nobody knows yet whether either produces a dataset. This file finds out: it writes each program to disk, runs it in the sandbox, and — for the Python arm only — feeds any traceback back to the model for up to two repair attempts. Then it compares the two arms' outputs cell for cell.

WHAT IT DOES. Three public entry points. `run_python_program(py_code, raw_path, domain, out_dir)` is the primary arm: run, repair on failure, standardise outputs, guarantee an XPT. `run_r_program(r_code, raw_path, domain, out_dir, compare_to=None)` runs the R arm in its own `_r/` subdirectory and reports whether its CSV matches the Python arm's. `rerun_r_evidence(...)` re-executes a delivered R program in a fresh timestamped directory and returns the full console log, duration, and a reproducibility verdict — it exists "so the browser can SHOW a live re-run instead of asserting one".

WHY IT EXISTS. "Correctness = executed data, not SAS text" is this project's core gate (memory: `sdtm-data-equivalence-gate`). A generated program that was never run is a claim, not a result. And a generated program that fails is often one traceback away from working — the repair loop converts a class of one-line model slips into delivered datasets, while the two-arm comparison converts "we think the spec is unambiguous" into a measured cell count.

INPUTS AND OUTPUTS. Input: code strings from M4.2, the raw CSV path (e.g. `golden/AE/ae_raw.csv`), a domain, an output directory. `run_python_program` returns `{"ok": bool, "stderr": ..., "repairs": [...], "repaired": bool, "csv": ..., "xpt": ..., "n_rows": int, "columns": [...], "preview": [first 15 rows as dicts]}`. `run_r_program` adds `{"sandboxed": ..., "resource_limited": ..., "matches_python": bool|None, "compare_note": "..."}` — a `compare_note` reads like `"identical after sorting on USUBJID, AESTDTC, AESEQ"` or `"9,871 cells differ (EPOCH 3500; EGSEQ 3200; ...)"`. For the AE running example: the golden raw row `("CDISCPILOT01", "701-1015", "Application Site Erythema", "Mild Adverse Event", "Probably Related", IT.AESTDAT="01/03/2014")` goes in, and the executed program's `ae.csv` comes out with `USUBJID=CDISCPILOT01-701-1015, AESEQ=1, AESEV=MILD, AEREL=PROBABLY RELATED, AESTDTC=2014-01-03` — verified against the actual files in `golden/AE/`.

HOW IT WORKS — THE PYTHON ARM AND THE MAP/EXEC DISTINCTION. `_run_once` stages the job directory and this is where the module's central naming convention lives:

```
prog = os.path.join(out_dir, f"{domain}_exec.py")
open(prog, "w").write(py_code)
...
sb = sandbox.run(["python3", prog], workdir=out_dir, timeout=150)
```

`AE_exec.py` is **the program that ran** — the execution record. The deliverable the customer downloads is `AE_map.py`, written elsewhere (and, on a successful repair, overwritten inside the loop: `open(os.path.join(out_dir, f"{domain}_map.py"), "w").write(code)` — "persist repaired code" — so the delivered program is the one that actually worked, not the one that first crashed). In `golden/AE/` the two are byte-identical (verified with `cmp`); after a mid-run repair they can differ from what codegen originally emitted. Before running, `_run_once` greps the program for its `read_csv("...")` filename and copies the raw file under that name into the job dir — the program dictates its input name, the executor adapts. The loop in `run_python_program` runs up to `MAX_REPAIRS + 1 = 3` attempts; success is "a CSV other than the raw input exists". On failure, `_repair` sends the last 1500 characters of traceback plus the full program back to `call_llm` with an instruction block that ends "CRITICAL: PRESERVE the target-column mapping. Do NOT fix the error by deleting a column" — because deleting the failing derivation is the cheapest possible "fix" a model can produce. The repair response is gated by `looks_like_code`; if it is prose, `_repair` returns `None` and the loop stops rather than retrying, since "another round would ask the same question at temperature 0 and get the same non-answer, and overwriting a working-but-buggy program with commentary loses the program AND replaces a real runtime error with a syntax error in prose." Two real 2026-07-22 cases motivated that: a 9.8 KB DM program with one genuine typo was destroyed by a prose "repair", and an EC "repair" died as `SyntaxError`. Finally, the arm standardises: copy whatever CSV appeared to `<domain>.csv`, and if no `.xpt` exists, write one via `pyreadstat.write_xport(..., file_format_version=5)` in a try/except (`xpt_warn` on failure) — the CSV is the artifact that must always land.

HOW IT WORKS — THE R ARM AND THE COMPARISON. `run_r_program` writes `<domain>_map.R` into an `_r/` subdirectory ("R runs in its OWN subdirectory so it cannot overwrite the Python arm's `<domain>.csv` — the whole point is to compare two INDEPENDENT products of the same mapping spec") and deliberately has **no repair loop**: "a repaired R program is no longer the artifact the customer downloads." If the sandbox is unavailable it refuses, with `executed: False` in the payload. The

comparison, `_same_table(a_csv, b_csv, domain)`, is the module's most instructive function for a statistician: it compares stripped string values cell-for-cell, normalising **row order only** — "a wrong value must stay wrong." Rows are sorted on `_SORT_CANDIDATES = ("USUBJID", "{d}TESTCD", "{d}DTC", "{d}STDTC", "VISITNUM", "{d}SPID", "{d}SEQ")` — whichever exist in the data, `--SEQ` deliberately **last**, because SEQ is a derived ordinal the two arms derive differently (EG's R arm numbered 8, 15, 22 where Python numbered 2, 3, 4), and sorting on a column the arms disagree about actively misaligns the rows. The header comment carries the measurement: comparing EG positionally reported 33,747 differing cells; sorted on the natural key, 9,871 — "70% of the loudest number in the parity evidence was the comparison function, not the data." Mismatch reports name the worst columns (`"EPOCH 3500; EGSEQ 3200"`) because that "sends them to the cause", not just to a diff.

THE PACKAGES. `pandas` for reading and comparing the emitted CSVs (`dtype=str, keep_default_na=False` — the same all-string discipline the generated programs are told to use), `pyreadstat` for the XPT fallback (present today: v1.3.5 under `/usr/bin/python3`), stdlib `subprocess` / `glob` / `shutil` / `re`, and `services.sandbox` for every execution. `_strip` and `looks_like_code` are imported from `services.codegen` — a comment records that a local `_strip` copy once carried the same prose-leak bug, hence "single source of truth".

THE SAS BRIDGE. The repair loop is what a SAS programmer does by hand: run, read the log, fix the line, rerun — except the "programmer" fixing it is the same class of model that wrote the bug, so the loop is bounded (2 repairs), the fix is syntax-gated, and the instruction forbids fixing-by-deletion. The two-arm comparison is double programming, genuinely: independent Python and R implementations from the same spec, compared cell-by-cell. The honest caveat, stated in `ta_programs.NOT_EXECUTED_NOTE`: agreement between the arms "is evidence about the SPEC's clarity, not about correctness of the values" — both arms can share a wrong reading of an ambiguous spec, which is exactly true of human double programming too.

TRY IT YOURSELF. The whole chapter in 22 lines — generate a mapping program from a spec dict, exec it in a restricted namespace, get SDTM-ish rows:

```
raw_rows = [
    {"PATNUM": "701-1015", "IT.AETERM": "Application Site Erythema",
     "IT.AESEV": "Mild Adverse Event"},
    {"PATNUM": "701-1023", "IT.AETERM": "Erythema",
     "IT.AESEV": "Severe Adverse Event"},
]
spec = {"USUBJID": "'CDISCPILOT01-' + row['PATNUM']",
        "AETERM": "row['IT.AETERM']",
        "AESEV": "row['IT.AESEV'].split()[0].upper()"}

lines = ["def map_ae(raw):", "    out = []", "    for row in raw:"]
lines.append("        rec = {'DOMAIN': 'AE'}")
for target, expr in spec.items():
    lines.append(f"        rec[{target!r}] = {expr}")
lines += ["        out.append(rec)", "    return out"]
program = "\n".join(lines)

ns = {"__builtins__": {}}          # restricted namespace: no open, no import
exec(compile(program, "<generated>", "exec"), ns)
sdtm = ns["map_ae"](raw_rows)
assert sdtm[0]["AESEV"] == "MILD" and sdtm[1]["USUBJID"].endswith("1023")
print(sdtm)
```

(The empty `__builtins__` blocks `open` / `__import__` from the generated code — a toy of the real sandbox's point, and nothing like a security boundary against a determined payload.)

LIMITS. Success means "a CSV appeared", not "the program exited 0" — a program that writes `ae.csv` and then crashes counts as `ok: True` in the Python arm (the R arm does check `returncode == 0`). The `read_csv` grep in `_run_once` takes the first match, so a program that reads a lookup file before its raw file gets the raw data copied under the wrong name. `_same_table` requires identical column sets and row counts before comparing at all, so a single extra column reports "column sets differ" and no cell count. Repair costs up to two model calls per failed domain and can still deliver a wrong-but-running program — running proves execution, never values. And SAS, as always in this stack, is never executed here at all.

HOW TO IMPROVE IT. Record `returncode` in the Python arm's result and flag the wrote-then-crashed case. Sandbox-gate the comparison inputs' size (a generated program could emit a multi-GB CSV inside its 512 MiB file cap — several files evade it). Extend `rerun_r_evidence`'s fresh-directory, full-log pattern to the Python arm, which currently reruns in place.

LIFT AND REUSE. Two liftable assets. First, the bounded, syntax-gated repair loop: run → traceback → model → `looks_like_code` gate → stop on non-answer; parameterise the runner and the validator and it works for any generated-code pipeline. Second, `_same_table` as an order-insensitive dataset equality check with named worst offenders — for clinical data, keep the exact design decisions: natural-key sort, derived ordinals last, values never normalised.

M4.5 `/root/sdtm_mapper/golden/` and `/root/sdtm_mapper/output/` — blessed references versus past machine leftovers

Two directories sit side by side in `sdtm_mapper`, both full of files named `AE_map.py`, `ae.csv`, `AE_mapping_spec.xlsx`. One of them is source-of-truth; the other is a scrapyard of previous runs. Confusing them is the single easiest way to mis-document this codebase — a file in `output/` looks exactly like a module and is nothing of the kind.

WHAT IT DOES. `golden/` holds one directory per domain — `AE DM DS EX VS` — each containing the human-blessed reference set for that domain: the raw input (`ae_raw.csv`), the blessed mapping programs (`AE_map.py`, `AE_map.R`, `AE_map.sas`), the execution record (`AE_exec.py`, byte-identical to `AE_map.py` in AE — verified with `cmp`), the expected outputs (`ae.csv`, `ae.xpt`), the spec documents (`AE_mapping_spec.xlsx` / `.docx`), and a conformance report. `output/` holds 58 entries: 55 directories named by job hash (`005563d020de`, `9739c66d9ffa`, ...) plus `demo-ae`, `demo-dm`, `demo-vs` — each one the working directory of a **past pipeline run**, laid out by `executor._run_once` and friends.

WHY IT EXISTS. The golden set is the oracle. `scripts/codegen_parity.py` states the two uses explicitly in its header: `scripts/parity_gate.py` runs `golden/<DOM>/<DOM>_map.py` — "a STATIC file committed" — to prove the blessed program reproduces the blessed dataset, while `codegen_parity.py` runs the LIVE pipeline ("profile -> spec(LLM) -> code(LLM) -> execute -> compare to golden, cell by cell") and reports "CODEGEN PARITY: %d of %d domains reproduce golden". Without a golden set, "the generated program ran" would be the only obtainable evidence; with one, "the generated program reproduces the reference dataset cell-for-cell" becomes measurable. Module 8's parity story builds directly on this. `output/` exists because every execution needs a writable job directory (M4.3: exactly one), and old ones are simply never deleted.

INPUTS AND OUTPUTS. The AE golden raw file, `golden/AE/ae_raw.csv` (this is the on-disk name of Module 1's raw AE file), begins:

```
"STUDY","PATNUM","FOLDER","FOLDERL","IT.AETERM","AEOUTCOME","AELLT",...,"IT.AESEV","IT.AESER","IT.AE
REL",...,"IT.AESTDAT","IT.AEENDAT"
"CDISCPILOT01","701-1015","AE","Adverse Events","Application Site Erythema","Not Recovered/not
Resolved",...,"Mild Adverse Event","No","Probably Related",...,"01/16/2014","01/03/2014"
```

and the blessed output `golden/AE/ae.csv` begins:

```
STUDYID,DOMAIN,USUBJID,AESEQ,AESPID,AETERM,AELLT,AEDECOD,...,AESTDTC,AEENDTC,AESTDY,AEENDY
CDISCPILOT01,AE,CDISCPILOT01-701-1015,1,,Application Site Erythema,APPLICATION SITE
REDNESS,APPLICATION SITE ERYTHEMA,...,2014-01-03,,,
```

That pair is the contract: raw EDC-style columns with `IT.`-prefixed item names and MM/DD/YYYY dates in, SDTM AE with derived `USUBJID`, `AESEQ`, CT-standardised `AESEV=MILD` / `AESER=N`, ISO `AESTDTC=2014-01-03` out. (Note `AESTDY` / `AEENDY` are blank — the golden program leaves them empty without a `dm.csv`, honestly, rather than inventing a reference date.)

HOW IT WORKS — ANATOMY OF A GOLDEN PROGRAM. `golden/AE/AE_map.py` (269 lines, read in full for this module) is a straight-line pandas script, and its structure is the de-facto template every generated program is prompted toward. In order: a module docstring stating what it implements; path constants (`RAW_CSV = "ae_raw.csv"`, `OUT_CSV = "ae.csv"`, `OUT_XPT = "ae.xpt"`, `STUDY_DEFAULT = "CDISCPILOT01"`); three helpers — `get_col(df, name, default="")` (a missing column becomes a blank Series, never a KeyError), `to_iso_date(series)` (parses `%m/%d/%Y` first, generic parse as fallback, blanks stay blank), `study_day(dtc_iso, ref_iso)` (the SDTM +1 convention: `np.where(diff.notna() & (diff >= 0), diff + 1, diff)`); then one commented block per spec row:

```
sev_map = {
    "Mild Adverse Event": "MILD",
    "Moderate Adverse Event": "MODERATE",
    "Severe Adverse Event": "SEVERE",
}
sev_src = get_col(raw, "IT.AESEV")
ae["AESEV"] = sev_src.map(sev_map).fillna("")
```

`Not Submitted` variables are created empty (`ae["AESPID"] = ""` — rule 7 from M4.1, in the flesh); `dm.csv` is merged for `RFSTDTC` only if present, else an INFO line prints and study days stay blank; `AESEQ` comes from a `mergesort` sort on `["STUDYID","USUBJID","AEDECOD","AESTDTC","AETERM"]` then `groupby("USUBJID").cumcount() + 1`; a fixed 24-column order is enforced; CSV is written first, always, and the XPT write (`pyreadstat.write_xport(..., table_name="AE"[:8], file_format_version=5)`) is wrapped in try/except so a failed XPT never costs the CSV. A SAS programmer will recognise the whole file instantly: it is a data step with `PROC FORMAT`-style value maps, a `PROC SORT` + `BY`-group counter, and a `PROC COPY` to transport format — written longhand in pandas.

WHAT MAKES A PROGRAM GOLDEN. Not its origin — a golden program may well have started life generated — but its status: a human reviewed it, committed it as a static file, and the CI gates now treat its output as the oracle. The `golden/AE/AE_map.R.bak.p0patch` file sitting beside `AE_map.R` (dated 2026-07-21, later than the rest) is the visible trace of that curation: goldens get patched by hand, and the patch history is kept.

THE PACKAGES. The golden program itself uses `pandas`, `numpy`, `pyreadstat` — the same trio the generated programs are prompted to use, so golden and generated are comparable like-for-like. `output/` needs no packages; it is inert data.

TRY IT YOURSELF. A miniature parity gate — the golden-vs-generated comparison in 16 lines:

```
import io, csv

golden = "USUBJID,AESEV\nP-1015,MILD\nP-1023,SEVERE\n"
generated = "USUBJID,AESEV\nP-1023,SEVERE\nP-1015,MILD\n"     # same data, reordered

def rows(text):
    return list(csv.DictReader(io.StringIO(text)))

def parity(a_text, b_text, key="USUBJID"):
    a = sorted(rows(a_text), key=lambda r: r[key])   # order-insensitive,
    b = sorted(rows(b_text), key=lambda r: r[key])   # values untouched
    if len(a) != len(b):
        return False, f"row count differs ({len(a)} vs {len(b)})"
    diff = sum(1 for x, y in zip(a, b) for c in x if x[c].strip() != y[c].strip())
    return diff == 0, f"{diff} cells differ"

ok, note = parity(golden, generated)
assert ok, note
print("golden reproduced:", note)
```

LIMITS. Five domains have goldens; the pipeline claims many more, so most domains have no oracle at all (this is the Tier B situation M4.7 describes — "running it establishes that it produces a dataset, never that the values are correct"). The golden AE program hardcodes `STUDY_DEFAULT = "CDISCPILOT01"` and the `%m/%d/%Y` date layout — it is golden for this raw shape, not a universal AE mapper. And `output/` is an operational hazard exactly because nothing distinguishes its files from source by name: the 2026-07-26 incident in which a `--force` rebuild destroyed 27 generated programs happened because generated artifacts live in gitignored directories that a clean commit does not save. **Never document, edit, or rely on a file under `output/<hash>/` as if it were a module.**

HOW TO IMPROVE IT. Add a `MANIFEST.json` per golden domain recording who blessed it, when, from which run hash, and the digest of each blessed file — today that provenance lives in git history and filenames like `.bak.p0patch`. Give `output/` a retention policy (it accumulates forever) and stamp each job dir with the git HEAD and model that produced it, the way `ta_programs` payloads already do in the review fork.

LIFT AND REUSE. The golden-directory pattern itself: `raw input + blessed program + blessed output` per domain, with two separate CI gates (static program reproduces output; live pipeline reproduces output). It transfers to any codegen system. The interface is just a directory convention plus a cell-level comparator — resist the urge to make it a database.

M4.6 `/root/sdtm_review/services/ta_mapper.py` — deterministic column matching before any model

A therapeutic-area pack lands: nine `*_raw.csv` files, a few hundred columns. Until this module existed, every mapping decision — including `study_id` → `STUDYID` — went to a language model, which "bought variance at full price – 1.9 regenerations per domain, and a spec that told the generator to look up `lab_test_name` in a sponsor dictionary that does not exist." This file takes the decisions that are not decisions away from the model.

WHAT IT DOES. Eight deterministic matchers run in strict precedence order over every raw column; each returns a mapping and the rule id that produced it, so no line of the plan is unexplained. What no rule can decide is not guessed — it lands in a residual list, "which is the only thing a model or a human is ever asked about." It also routes each raw dataset to its domain (`route_dataset`), builds the

master mapping plan for a whole TA (`build_plan`), identifies wide Findings columns that are really tests (`pivot_columns`), and spots columns that belong in a different domain entirely (`better_home` — `height_cm` in `dm_raw.csv` is a VS finding, not a failure).

WHY IT EXISTS. Determinism, explainability, and reviewer economics. The module docstring's version of the last one is the memorable one: "A reviewer who must check 200 machine guesses checks none of them; a reviewer handed 20 open questions answers them." Sending only genuine judgment calls to a human (or a model) is the whole design; measured on the Cardiology pack, "the matchers decide the large majority of columns outright" (the docstring deliberately gives no bare percentage here — the per-run number lands in the plan's `totals`).

INPUTS AND OUTPUTS. `map_column(column, domain, values=None, ig_vars=None, rules_path=None)` takes one column name (plus optionally its observed values) and returns a dict: `{"column": "severity", "target": "AESEV", "rule": "R5:codelist", "why": "every value is a term of AESEV's codelist and of no other codelist in AE", "confidence": 0.85, "decided": True}` — or `decided: False` with `target: None`. `build_plan(ta, corpus_root=None)` walks every `*_raw.csv` under the TA's corpus directory and returns the master plan: per-domain variable tables (each row carrying `source_column`, `rule`, `why`, `confidence`, observed `max_length`, and a `status` of `mapped` / `derived` / `REQUIRED-UNSOURCED` / `not collected`), plus the three lists that matter — `residual_columns` (needs judgment), `machine_dispositions` (residuals the diagnosis already resolved), `required_unsourced` — and `totals` with `pct_deterministic`, `by_rule` counts, and a hard invariant: `decided + dispositioned + judgment must equal n_columns` or it raises.

HOW IT WORKS. The matcher table is the module in one screen:

```
MATCHERS = [
    ("R1:exact", _by_exact),
    ("R2:alias", _by_alias),
    ("R2.5:learned", _by_learned),
    ("R3:label", _by_label),
    ("R4:pattern", _by_pattern),
    ("R5:codelist", _by_codelist),
    ("R7:pivot", _by_pivot),
    ("R6:description", _by_description),
]
```

"Order IS precedence and must not be sorted" — a later rule may never overturn an earlier one, so a fuzzy description match can never displace an exact CDISC name. R1 tries plain equality against every IG variable before trying the underscore-stripped form against any, "so a real exact match never loses to a stripped coincidence." R2 reads `knowledge/raw_column_aliases.md` — markdown on purpose, because it is a human's file and "a JSON file loses the argument and keeps only the answer." R2.5 reads rules a reviewer's ledger decisions exported into `mapping_rules.json` (via `ta_knowledge`), ranked below the hand-written alias file on purpose. R3 compares denoised token sets of column name vs IG label (`_tokens` strips units and filler like `mmhg`, `bpm`). R4 is structural convention (`*_start_date` → `--STDTC`). R5 is the strongest data-driven signal: if every distinct value of `severity` is a term of exactly one codelist in the domain, "the column IS AESEV" — and it requires a unique winner, because two candidates "means the data cannot distinguish them and a human should." R7 recognises wide Findings columns as tests feeding `--ORRES`, and its docstring records why it sits before R6: R6 once mapped `qrs_duration_ms` to `EGTEST` "on the strength of the word 'name'... A weak textual matcher will always find something; the job of precedence is to make sure something better got there first." R6, description overlap, demands a ≥0.75 strict-majority overlap and a unique best. `CONFIDENCE` maps each rule id to a number that is explicitly "not a probability – a review priority." `route_dataset` combines

three reported-separately signals — filename stem, presence of the domain's Topic variable, and specific-column coverage (identifier hits like `STUDYID` are excluded because they "match every domain and so carry no discriminating information") — and is `confident` only when stem and column evidence agree. `build_plan` then assembles everything, computes `max_length` from the longest observed value ("a define.xml is a promise about the data, not about the standard"), reclassifies pivot-supplied `--TESTCD`/`--TEST` gaps and constant-derivables (`derived_rule`: DOMAIN, STUDYID, USUBJID, `--SEQ`) so the unsourced list is not "75% noise", and runs `mapping_diagnosis.diagnose_plan` + `_partition_dispositions` so the plan on disk already separates machine-resolved residuals (companion time columns → R8, SUPP qualifiers → R9, operational fields → R10, unambiguous re-homes → R11) from genuine judgment calls.

THE PACKAGES. Stdlib only — `re`, `os`, `collections.Counter`, `csv` — plus sibling services (`ig_reference`, `ta_ct`, `value_domains`, `ta_knowledge`, `mapping_diagnosis`) for the CDISC metadata. No pandas: `_read_csv` uses the stdlib `csv` module, entirely adequate for header-and-values scans. That absence is a portability feature — this module runs anywhere Python does.

THE SAS BRIDGE. This is the module a clinical programmer will recognise fastest, because it encodes the first pass of a mapping meeting: obvious renames, the sponsor's known aliases, label matches, naming conventions, "those values are clearly AESEV" — done before anyone argues. The precedence ladder is the meeting's seniority: an exact name beats an alias beats a label beats a hunch. And crucially, it does what a good lead programmer does with the leftovers: writes them on the board as open questions instead of guessing. Nothing here calls a model — "`residual()` is what a model is for" — which also means every decision is reproducible run-to-run, unlike everything in M4.1/M4.2.

TRY IT YOURSELF. A three-rule matcher with real precedence:

```
import re

IG_AE = {"AESEV": "Severity/Intensity", "AESTDTC": "Start Date/Time of Adverse Event",
         "AETERM": "Reported Term for the Adverse Event"}
CODELISTS = {"AESEV": {"MILD", "MODERATE", "SEVERE"}}

def by_exact(col, values):
    return (col.upper(), "R1:exact") if col.upper() in IG_AE else None

def by_pattern(col, values):
    if re.match(r"^(.*_)?start_?date$", col, re.IGNORECASE):
        return ("AESTDTC", "R4:pattern")
    return None

def by_codelist(col, values):
    vals = {v.strip().upper() for v in values if v.strip()}
    hits = [var for var, terms in CODELISTS.items() if vals and vals <= terms]
    return (hits[0], "R5:codelist") if len(hits) == 1 else None

def map_column(col, values=()):
    for rule in (by_exact, by_pattern, by_codelist):   # order IS precedence
        hit = rule(col, values)
        if hit:
            return {"column": col, "target": hit[0], "rule": hit[1]}
    return {"column": col, "target": None, "rule": None}   # residual: ask a human

assert map_column("aeterm")["rule"] == "R1:exact"
assert map_column("ae_start_date")["rule"] == "R4:pattern"
assert map_column("severity", ["Mild", "SEVERE"])["rule"] == "R5:codelist"
assert map_column("investigator_comment")["target"] is None
print("precedence held; one residual for a human")
```

LIMITS. The matchers are only as good as their reference metadata — R5 needs the CT package loaded, R2.5 needs the exported rules file, and the two forks' knowledge packs disagree (M4.2). A codelist match can be a coincidence on small value sets (guarded by `len(vals) > 40` reject and the unique-winner rule, but a two-value Y/N column fitting several NY-governed variables is exactly the ambiguity that lands in residual). `route_dataset`'s stem signal trusts filenames (`ae_raw.csv`), which is "exactly the kind of thing that is wrong once in a while, so it never decides alone." The docstring's "large majority" is honest phrasing for a number that varies by pack — check the plan's `totals.pct_deterministic` for the run you care about rather than quoting a fixed figure.

HOW TO IMPROVE IT. Persist per-rule precision from reviewer overturn events (`ta_knowledge` already has the ledger) so `CONFIDENCE` becomes measured rather than assigned. Add a matcher for `--ORRESU` unit columns riding alongside a pivoted test (currently units land in residual or companion diagnosis).

LIFT AND REUSE. The generic asset is precedence-ordered deterministic matchers with per-decision provenance — a pattern worth lifting into any schema-mapping problem (EHR ETL, ADaM specs, warehouse migrations). The right interface is exactly `map_column`'s: `(column, context) -> {target, rule, why, confidence, decided}`. Keep three properties in any port: order is precedence and is never sorted; every hit names its rule; and no-match is a first-class answer, not an exception.

M4.7 `/root/sdtm_review/services/ta_programs.py` — builds, caches, and audits domain programs

A reviewer opens the study browser and clicks "show me the R program for AE." Generating one takes minutes of model calls — unacceptable inside a page load — and generating one per therapeutic area would make twenty-one copies of the same file, because all 21 Tier B areas share byte-identical headers per domain. This module is the study-level answer: generate once per **header shape**, cache with provenance, and audit every cached program for the defects generation is known to produce.

WHAT IT DOES. `build_programs(domain, ...)` assembles the final spec (fleet-adjudicated spec → deterministic enforcement → human decisions → optional study visit map), calls `codegen.generate_all_code` for SAS, R, and Python, wraps each program with metadata (digest, line count, model, executability, fabrication/serialization findings), and writes the payload to `data/ta_packs/_programs/<DOMAIN>.json` — archiving whatever was there first. `load_programs(domain, ...)` reads that cache and returns `None` if nothing was ever generated: it "never generates. A page load must not be able to trigger minutes of model calls."

WHY IT EXISTS. Three reasons, numbered in its own docstring. Per-shape caching (one program serves all 21 TAs — until a study visit map is inlined, at which point `cache_name(domain, ta)` scopes the cache per TA, because "a program carrying one cannot be shared across therapeutic areas"). Honest executability (SAS "is never executed. There is no SAS runtime on either box… every consumer of this module is handed `executable: False` alongside it so a caller cannot present it as tested code by omission"). And content addressing: `digest` changes whenever the program text changes, so a reviewer's line-anchored comment "can be marked stale rather than silently re-pointed at whatever now occupies that line number."

INPUTS AND OUTPUTS. Input: a domain code, optional TA, optional injected `codegen`/`bridge` (so tests can exercise "a backend that returns prose instead of a program" offline). Output: a JSON payload per domain with `programs.{r,sas,python}` — each carrying `code`, `digest`, `lines`, `model`, `executable`, `note` (the blunt `NOT_EXECUTED_NOTE` texts), `is_code`, `not_code_reason`, `fabrication`, `output_defects` — plus `spec_rows`, `spec_origin` (fleet/human/placeholder counts), `git_head`, `git_dirty`, `study_context`,

`human_decisions_applied`, `spec_enforcement`, and an `applies_to` sentence stating the sharing claim explicitly. The cache lives at `/root/sdtm_review/data/ta_packs/_programs/` (present today: `AE.json`, `CM.json`, `DM.json`, …), filled by `scripts/build_ta_programs.py`, never by a request.

HOW IT WORKS. `build_programs` is a pipeline of guarded stages, each with a documented incident behind it:

1. `bridge.build_spec(shape_id_for(domain))` fetches the adjudicated spec (`shape_id_for` returns `"tierB_robustness:AE"`).
2. `spec_enforce.enforce_spec` runs before codegen. The comment carries the measurement for why: this enforcement module "existed since 2026-07-21 with nine rule families and a test suite, and the TA pipeline never called it" — left unenforced, the two arms guessed differently about underivable variables, and "Python invented MHSTDY = -5716, a study day fifteen years before the reference date. 9,371 of the differing cells between the arms were these columns."
3. `apply_decisions(spec, decisions, domain, header=None)` overlays accepted human decisions — precedence "human > fleet > placeholder" — carrying the reviewer's rule text verbatim so "the generated program is told why, not merely what." The `header` guard skips a RAW decision whose column is not in this file's header, because the decisions ledger is domain-keyed only and would otherwise leak across studies.
4. For visit-bearing domains (`VS`, `LB`, `EG`, `EX`, `PE`) with a TA given, `inject_visit_context` rewrites the VISITNUM/VISIT spec rows to inline the study's visit map — inline, "never fetched by the program", because "a generated program must reproduce its dataset without reading another domain's file." A blank VISITNUM had been collapsing "five or six distinct visits into one natural key."
5. `codegen.generate_all_code(..., raw_path=...)` — with the raw extract path when the TA has one, so value domains reach the prompt (the missing-lab-dictionary incident from M4.2).
6. Per language, the audit block: `looks_like_code`, then two regex detectors. `suspect_fabrication` matches signatures like `USUBJID\s*=\s*c\(\s*"` ("hardcoded subject identifiers") and `by\s*=\s*character\(\)` (cross-join) — every pattern "comes from a defect observed in a real generated program, not from theory." Its docstring holds the module's sharpest lesson: the LB program that fabricated a two-subject `dm_ctx` and a three-entry lab dictionary "passed conformance **41 checks to 2**... Only the values are fiction" — which is why detection matters more than the prompt. `suspect_output_defects` reads `write_csv`/`write.csv` call arguments for a missing `na = ""` — reading the arguments, because an earlier lookahead version "flagged the correct form... a detector that fires on the fix is worse than none, because it trains you to ignore it."
7. `_archive(target)` before writing: the previous payload is copied to `_previous/<name>.<digest>`. The docstring records the 2026-07-26 disaster verbatim — a `--force` regeneration replaced 27 programs (7 of 9 domains working) with a set where 2 of 9 worked, unrecoverably, because the directory is gitignored. And the line that defines this whole module family: "**Regeneration is not a repeat, it is a fresh draw** – the generator is not deterministic even at temperature 0 – so what gets overwritten may be strictly better than what replaces it."

`anchor_for("r", 42)` / `parse_anchor("program_r:42")` round out the file: ledger anchors tying a reviewer comment to a line of a specific program digest.

THE PACKAGES. Stdlib (`json`, `os`, `logging`, `datetime`) plus sibling services; the heavy lifting (LLM, spec) is behind deferred imports so importing this module is cheap. Content digests come from `services.provenance.digest_bytes`. A database would be the conventional alternative to JSON-files-in-a-directory; the files win here because the payloads are also the deliverable review artifacts and diff cleanly.

THE SAS BRIDGE. `EXECUTABLE = {"r": True, "python": True, "sas": False}` is the single most honest line in the codebase for a SAS reader: the SAS program — the deliverable your shop would actually run — is the one arm that has never been executed anywhere. Its quality evidence is indirect: the same spec produced R and Python arms that ran and (ideally) agreed. Treat the delivered `.sas` exactly as `NOT_EXECUTED_NOTE["sas"]` says: "a deliverable to be run in your own validated environment, not as tested code."

TRY IT YOURSELF. Content addressing plus a fabrication detector, both from stdlib:

```
import hashlib, re

program_v1 = 'library(dplyr)\nae <- read_csv("ae_raw.csv")\nwrite_csv(ae, "ae.csv", na = "")\n'
program_v2 = program_v1.replace('na = ""', "")          # someone dropped the na fix

def digest(code):
    return hashlib.sha256(code.encode()).hexdigest()[:12]

FABRICATION = [
    (r'USUBJID\s*=\s*c\(\s*"', "hardcoded subject identifiers"),
    (r"by\s*=\s*character\(\)", "cross-join multiplies every row"),
]

def audit(code):
    found = [why for pat, why in FABRICATION if re.search(pat, code)]
    for m in re.finditer(r"write_csv\s*\(([^)]*)\)", code):
        if "na =" not in m.group(1):
            found.append('write_csv without na = "" writes literal "NA" cells')
    return found

assert digest(program_v1) != digest(program_v2)          # comment anchors go stale, loudly
assert audit(program_v1) == []
assert audit('dm <- tibble(USUBJID = c("S-101"))') == ["hardcoded subject identifiers"]
print("v2 defects:", audit(program_v2))
```

LIMITS. The fabrication and serialization detectors are R-only (`if lang != "r": return []`) — a Python program that hardcodes subjects sails through, relying on the executed-parity net instead. They are regexes: a fabricated lookup built with `data.frame(id = ...)` instead of `USUBJID = c(` is invisible to them. The per-shape sharing claim is load-bearing and fragile — it "holds only while nothing study-specific is inlined", and the code enforces that only via `cache_name`'s TA scoping. `_archive` keeps every previous version forever (no pruning). And nothing in this module runs anything: `is_code: True` plus zero findings still means unproven values until M4.4's executor and an oracle say otherwise.

HOW TO IMPROVE IT. Port the two detectors to Python (the patterns translate directly: `pd.DataFrame({"USUBJID": [...]})`, `merge(..., how="cross")`). Wire a rejection loop: today a program with `fabrication` findings is cached and surfaced; `suspect_fabrication`'s docstring already says the point is that "a caller can reject the draw and generate again" — no caller does that automatically yet. Add cache pruning with a keep-last-N policy for `_previous/`.

LIFT AND REUSE. Three separable ideas. (1) Content-addressed artifact cache with archive-before-overwrite — lift `_archive` verbatim into any pipeline that regenerates nondeterministic artifacts; it is the coded form of the global backup rule this project learned the hard way. (2) The `executable`/`note` honesty envelope: every artifact carries its own evidentiary status, so no consumer can overstate it by omission. (3) Incident-derived static detectors: start the list empty, add a regex only when a real defect ships, and never let a detector fire on the fix.

What this module still owes you

THE OTHER EXECUTOR. `/root/sdtm_mapper/services/executor.py` (the mapper fork's) was outside this module's assigned files. It shares ancestry with M4.4 but, like the codegen pair, must be assumed divergent until read — do not transfer claims from this module to it.

HOW THE ARMS GET JUDGED. M4.4 measures whether two arms agree and M4.5 shows the oracle for five golden domains, but the scoring machinery — conformance checking (`core_validator`, the review harness), the accuracy gate's cheap→strong escalation, and the full parity evidence chain — belongs to later modules; Module 8's parity story builds directly on the golden-vs-generated distinction and `_same_table` semantics established here.

THE SPEC'S PROVENANCE. This module treats the mapping spec as given. How `spec_bridge` adjudicates fleet outputs against human decisions, what `spec_enforce`'s nine rule families actually enforce, and how reviewer ledger events become `mapping_rules.json` for R2.5 are upstream stories (Modules 3 and 5 territory).

THE LLM CASCADE ITSELF. `call_llm`'s backend ordering, spend guards, and kill switches were referenced but not documented here — they have their own module, and the CLAUDE.md files warn that comments about the cascade order have been stale before; trust the code.

NUMBERS WE DID NOT GIVE YOU. Wherever this module quotes a figure — 33,747 vs 9,871 cells, 41-to-2 conformance on a fabricated dataset, 250+ processes through a 64-proc rlimit, ~79,000 "NA" cells, 2-of-9 vs 7-of-9 working programs — it is a number recorded in the source being documented, from a specific past run, not a property you should expect to reproduce. The stack's own memory is blunt on this: any single generation run is a draw, not a result. What is verified today, on this host: every file, function, and path named above exists as described; `bwrap`, `systemd-run`, `Rscript` (R 4.5.2, dplyr 1.2.1, readr 2.2.0) and `/usr/bin/python3` 3.14.4 (pandas 3.0.3, pyreadstat 1.3.5) are installed; `golden/` holds AE, DM, DS, EX, VS; `output/` holds 58 entries; and `golden/AE/AE_map.py` is byte-identical to `golden/AE/AE_exec.py`. What is not verified here: any end-to-end generation run — this module read and traced the machinery; it did not spend model calls to spin it.

Module 5 — Validation and conformance

Module 4 ended with a synthetic AE dataset written to disk by generated code; this module is everything that refuses to take that file's word for it. There are five distinct layers here — two near-twin in-house rule engines, a set of millisecond post-run gates, a therapeutic-area rule registry, and the real CDISC CORE engine run as a subprocess — plus the two spec modules that define what "conforms to the mapping spec" even means. If you know Pinnacle 21 reports, you already know the shape of this module; the work is learning where a rules engine in code is stricter, weaker, or simply more honest than a P21 Excel tab.

M5.1 `/root/sdtm_mapper/services/validator.py` — ten fixed checks, Pinnacle-style severities

You upload a raw AE extract to the mapper, it generates a program, the program writes `AE.csv` — and thirty seconds later the UI shows a conformance report with a verdict, a percentage, and rows colour-coded Reject/Error/Warning/Notice. No Pinnacle 21 license was involved. This file is that report.

WHAT IT DOES. It reads a produced SDTM dataset (a CSV) plus the SDTMIG domain metadata for that domain, runs ten hard-coded structural checks over it, and returns a dict: per-check pass/fail rows, a severity summary, a weighted conformance percentage, and an overall verdict. It can also write the same report as a styled Excel workbook and, optionally, ask an LLM for a plain-English remediation paragraph.

WHY IT EXISTS. Pinnacle 21 is commercial and not installed on this box, but a mapping tool that emits datasets with no conformance feedback is unreviewable — the demo (built for an AstraZeneca interview) needed a P21-shaped answer to "is this output structurally sane?" without P21. The module docstring states the design constraint outright: "Pure-Python and deterministic — no LLM in the core checks (a regulatory requirement)." Without it, a generated dataset missing `AEDECOD` entirely would sail into the download button.

INPUTS AND OUTPUTS. Input: `validate(csv_path, domain)` — a path to e.g. `AE.csv` and the string `"AE"`. Domain metadata comes from `knowledge/sdtm_ig34.json` (loaded at import into `KP`; `KP["ig_version"]` is `"SDTMIG v3.4 (SDTM v2.0)"`). Output: a dict shaped like

```
#| skip-check
{"domain": "AE", "ig_version": "SDTMIG v3.4 (SDTM v2.0)", "n_rows": 2,
 "summary": {"Reject": 0, "Error": 3, "Warning": 1, "Notice": 0, "pass": 24, "total": 28},
 "conformance_pct": 88.2, "verdict": "Error",
 "checks": [{"id": "CG0030", "severity": "Error", "variable": "AEDECOD",
             "status": "fail", "message": "AEDECOD is blank on 1 record(s).", "count": 1}, ...]}
```

Every finding is one flat dict appended by the tiny helper `_chk(checks, cid, sev, variable, status, message, count=0)` — that six-field row is the finding shape for the whole in-house engine.

HOW IT WORKS. `validate` loads the CSV with `pd.read_csv(csv_path, dtype=str, keep_default_na=False)` — everything is a string, and blank stays `""` rather than becoming `NaN`, which matters because half the checks are "is this cell blank". Then ten numbered blocks run in order: CG0001 (DOMAIN present and equal to the domain code on every record — the only Reject alongside CG0010), CG0010 (every `core == "Req"` variable present as a column), CG0020 (Expected present — Warning only), CG0030 (Required populated on every record), CG0040/CG0041 (`--SEQ` numeric, unique within USUBJID), CG0050/CG0051 (record uniqueness — authoritative on `USUBJID + --SEQ` when the domain has a SEQ, natural key otherwise, with natural-key repeats downgraded to an informational Notice in event

domains), CG0060 (every `--DTC` variable matches the `ISO8601` regex `^\d{4}(-\d{2}(-\d{2}(T\d{2}(:\d{2}(:\d{2})?)?)?)?)?$` — partial dates like `2026-01` legal, `05JAN2026` not), CG0070 (controlled-terminology membership, discussed below), CG0080 (no columns outside the IG model), CG0090 (variable order, Notice). Scoring is severity-weighted:

```
SEV_WEIGHT = {"Reject": 10, "Error": 5, "Warning": 2, "Notice": 0}
```

`conformance_pct = 100 * (1 - penalty/max_pen)` where penalty sums the weights of failing checks — so one Reject costs as much as five Warnings, and Notices are free. The verdict is a simple ladder: any Reject → `"Reject"`, else any Error → `"Error"`, else any Warning → `"Warning"`, else `"Pass"`. The subtle part is `_ct_set(ct_text)`, which decides whether a variable's CT annotation is a checkable value list or merely a codelist name. `"NY"` resolves to `{"Y","N"}`; `"MILD/MODERATE/SEVERE"` resolves because it is in `SAFE_VALUE_LISTS`; but `"MEDDRA"`, `"TOXGR"` and thirty-odd other tokens in `CODELIST_NAMES` return `None` — deliberately unchecked, because value-checking against a codelist name would flag every legal value. Conservative by design: as the docstring says, the validator must never raise false positives on uncheckable CT.

THE P21 BRIDGE — WHERE IT HOLDS AND WHERE IT MISLEADS. The severity vocabulary (Reject > Error > Warning > Notice) is lifted straight from Pinnacle 21, and a SAS programmer will read the report instantly. Three places the analogy misleads. First, the `CG` prefixes look like published CDISC conformance rule ids but are house-assigned — CG0030 here is not CG0030 in any CDISC publication (the sister fork's `ta_compliance.py` fixes exactly this confusion by prefixing its rules `CLIN`). Second, P21 checks ~hundreds of rules with cross-domain joins; this is ten single-dataset structural checks — a clean report here is a far weaker claim than a clean P21 report. Third, the `conformance_pct` number has no P21 analogue at all: P21 counts issues, it does not average them into a percentage, and this percentage's denominator is "checks this engine happens to run", not "things that can be wrong".

THE PACKAGES. `pandas` for the CSV and the vectorised counting (`.str.strip()`, `.duplicated(subset=...)`); `re` for ISO 8601; `json` for the knowledge pack; `openpyxl` in `write_report_xlsx` for the styled workbook (fills, borders, `freeze_panes = "A5"`) because reviewers live in Excel. `remediation_summary(report)` optionally imports `services.llm.call_llm` (`tier="classify"`, `temperature=0.0`) inside a `try` — any exception falls back to a fixed sentence, so the deterministic report never depends on a model. Swap cost: the pandas usage is shallow (a `csv.DictReader` version is maybe 30 lines longer — `gates.py` in the sister fork proves it); openpyxl is the only real dependency and only for cosmetics.

TRY IT YOURSELF. A three-rule mini validator that catches a deliberately broken AE row — Module 4's synthetic subject 101002 with a blank `AEDECOD`, a garbage severity, and a SAS-format date:

```
import re
import pandas as pd

ae = pd.DataFrame([
    {"DOMAIN": "AE", "USUBJID": "SYNTH-001-101001", "AESEQ": "1",
     "AETERM": "HEADACHE", "AEDECOD": "Headache",
     "AESEV": "MODERATE", "AESTDTC": "2026-01-05"},
    {"DOMAIN": "AE", "USUBJID": "SYNTH-001-101002", "AESEQ": "1",
     "AETERM": "NAUSEA", "AEDECOD": "",            # required var blank
     "AESEV": "SEVERE!!",                          # outside CT
     "AESTDTC": "05JAN2026"},                      # SAS date, not ISO 8601
])
ISO = re.compile(r"^\d{4}(-\d{2}(-\d{2})?)?")
findings = []
for var in ("USUBJID", "AETERM", "AEDECOD"):        # rule 1: required populated
    n = (ae[var].str.strip() == "").sum()
    if n:
        findings.append(("CG0030", "Error", var, f"blank on {n} record(s)"))
bad = ~ae["AESEV"].isin({"MILD", "MODERATE", "SEVERE"})   # rule 2: CT membership
if bad.any():
    findings.append(("CG0070", "Error", "AESEV",
                     f"{bad.sum()} value(s) outside CT: {sorted(ae.loc[bad,'AESEV'])}"))
bad = ~ae["AESTDTC"].apply(lambda v: bool(ISO.fullmatch(v[:10])))  # rule 3: ISO 8601
if bad.any():
    findings.append(("CG0060", "Error", "AESTDTC",
                     f"{bad.sum()} non-ISO value(s): {sorted(ae.loc[bad,'AESTDTC'])}"))
for f in findings:
    print(f)
assert len(findings) == 3
```

LIMITS. Ten checks, one dataset at a time: no cross-domain checks at all (no "every AE USUBJID exists in DM"), no MedDRA/WHODrug dictionary validation, no value-level correctness — a dataset can pass everything here and still map the wrong source column into every cell. `conformance_pct` measures only what the engine checks, and the CT check covers only the value sets `_ct_set` can prove; "not flagged" and "valid" are different claims. The `--SEQ` numeric test accepts `"3.0"` (it strips one `.0`) but not `"3.00"`.

HOW TO IMPROVE IT. Make the rule list data instead of ten inline blocks (the sister fork's `ta_compliance.py` decorator registry is the finished version of that thought); add a `does_not_check` field per rule so the report carries its own scope statement; report `conformance_pct` with the check count beside it so 100% of 10 checks is never mistaken for 100% of P21.

LIFT AND REUSE. The genuinely portable pieces are `_ct_set` (the checkable-values-vs-codelist-name discrimination is a real, transferable insight — most homegrown CT checkers get this wrong and false-positive on `"MEDDRA"`), the `_chk` finding shape, and the severity-ladder verdict. Right interface: `validate(df, domain_meta) -> report` — take a DataFrame and a metadata dict rather than a path and a global pack, so the engine stops owning I/O and knowledge loading.

M5.2 `/root/sdtm_review/services/validator.py` — same engine, different knowledge, different verdicts

Run the same broken AE file through "the validator" in both repos and you can get different findings — not because the code diverged, but because three lines at the top decide which universe of SDTM metadata the ten checks live in.

WHAT IT DOES. This is the review fork's copy of the file you just read. The entire diff between the two files is the pack loading: the mapper does `KP = json.load(open(os.path.join(os.path.dirname(__file__), "..", "knowledge", "sdtm_ig34.json")))`, while this fork does `from services.knowledge import load_pack` then `KP = load_pack()`. Every check id, regex, weight, and function body is byte-identical (verified with `diff` on 2026-08-03: one 3-line hunk).

WHY IT EXISTS. The review fork rebuilt its domain knowledge as a versioned pack behind `services/knowledge.py` (its `load_pack()` reports `ig_version` `"SDTMIG v4.0 (SDTM v2.1)"`) instead of a raw JSON file (`"SDTMIG v3.4 (SDTM v2.0)"` in the mapper). Keeping the validator identical while swapping the knowledge source was the cheap fork; the price is that "the validator passed" is no longer one claim — it is a claim relative to a pack.

INPUTS AND OUTPUTS. Same signature, same report dict as M5.1. But concretely for our AE example: in the mapper pack, `AESEV` carries `ct: "MILD/MODERATE/SEVERE"` — a slash list that `_ct_set` resolves via `SAFE_VALUE_LISTS`, so CG0070 fires on `"SEVERE!!"`. In the review pack, the same variable carries `ct: "AESEV"` — a codelist name, which `_ct_set` returns `None` for, so this fork's CG0070 is silent on the identical bad value (verified by calling `_ct_set` from both trees). The review fork isn't blind to it — its `ta_ct/ct.py` layer resolves the real NCI codelist from `/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv` — but this file no longer catches it.

HOW IT WORKS. Identical control flow to M5.1; nothing to add except the consequence of the pack swap. The packs disagree on structure too, not just CT strings: CO defines `COSEQ` in the review pack and not in the mapper pack (verified against both packs; the review fork's CLAUDE.md warns about exactly this and makes its tests discover SEQ fixtures from the pack instead of hardcoding a domain). So CG0040/CG0041/CG0050 take different branches for the same CO dataset depending on which repo you are standing in. There is also a documented drift worth knowing: the review pack itself was later found to invent 42 variables the IG does not define and to miss 8 real ones — that audit lives in a comment block in `ta_compliance.assess` (M5.3), which is why that module overlays CDISC Library metadata while this validator still trusts the pack alone.

THE P21 BRIDGE. This is the chapter where the P21 analogy earns its keep: every P21 report is stamped with a config version and CT package version, and veterans know two P21 runs are only comparable if those match. Same law here, enforced by nothing: the report carries `KP["ig_version"]`, but nothing stops you comparing a v3.4-pack verdict against a v4.0-pack verdict as if they were the same instrument. In P21 you'd call that comparing across validator configs; here it's comparing across forks.

THE PACKAGES. Identical to M5.1 (`pandas`, `re`, `json`, `openpyxl`, optional `services.llm`). The one new dependency is conceptual: `services/knowledge.py` and whatever pack it loads. Swap cost of unifying the two forks onto one loader: trivial in code, non-trivial in verdict churn — checks that pass today would start failing purely from metadata movement.

TRY IT YOURSELF. The whole chapter in fifteen lines — one engine, two packs, two verdicts on the same row:

```
def validate(row, pack):
    findings = []
    for var, meta in pack.items():
        if meta["core"] == "Req" and not row.get(var, "").strip():
            findings.append(f"Required {var} is blank")
        allowed = meta.get("values")
        if allowed and row.get(var) and row[var] not in allowed:
            findings.append(f"{var}={row[var]!r} outside CT {sorted(allowed)}")
    return findings

row = {"USUBJID": "SYNTH-001-101002", "AESEV": "Severe"}
pack_a = {"USUBJID": {"core": "Req"},
          "AESEV": {"core": "Perm", "values": {"MILD", "MODERATE", "SEVERE"}}}
pack_b = {"USUBJID": {"core": "Req"},
          "AESEV": {"core": "Perm", "values": None}}   # codelist not resolvable
print("pack A:", validate(row, pack_a))   # fires: 'Severe' is not 'SEVERE'
print("pack B:", validate(row, pack_b))   # silent: nothing to check against
assert validate(row, pack_a) and not validate(row, pack_b)
```

LIMITS. Everything from M5.1, plus: the two copies are only identical today — nothing (no shared package, no test) pins them together, so the next edit to either silently forks the rule set itself, not just the metadata. And a "pass" from this file is quietly weaker on CT than the mapper's, because the v4 pack's ct strings are mostly codelist names that `_ct_set` refuses.

HOW TO IMPROVE IT. Delete one copy. Extract the engine into a shared module taking `(df, domain_meta)`, keep per-repo pack loaders, and add a test asserting both repos import the same engine version. Failing that, at minimum a CI check that `diff`s the two files and fails on divergence.

LIFT AND REUSE. The lesson more than the code: when you fork a validator, version the knowledge explicitly in every report (this code does, via `ig_version`) and treat cross-pack verdict comparisons as invalid by default. Any reusable rules engine should take the pack as a parameter, never import it at module load — module-load state is why these files had to be forked at all.

M5.3 `/root/sdtm_review/services/ta_compliance.py` — declarative CLIN rules that state their blind spots

A reviewer once looked at a compliance tab full of green and asked the only question that matters: green at what? An LB program that fabricated lab records from a two-subject context — tests that do not exist in any dictionary — had scored 41 checks passing and 2 failing. The module docstring records that incident, and the design answer to it is this file's signature feature: every rule carries a `does_not_check` field, and the report prints it.

WHAT IT DOES. A registry of 17 conformance rules, each a small function over a parsed CSV, registered by decorator, prefixed `CLIN` (CLIN0001–CLIN0006, CLIN0010–CLIN0013, CLIN0020–CLIN0022, CLIN0030–CLIN0031, CLIN0040, CLIN0050). `assess(domain, csv_path, dm_csv_path=None)` runs all of them over one produced dataset and returns per-rule results plus totals; `rule_catalogue()` returns the rule list itself so the UI can show a reviewer what is and is not covered.

WHY IT EXISTS. Two reasons, both stated in the docstring. Honesty about provenance: "Every rule id is prefixed `CLIN` for exactly one reason: a reader who sees `SD0064` or `CORE-000123` is entitled to assume the published rule of that number ran, and it did not." That is the anti-P21-cosplay rule — the mapper fork's `CG` prefixes are precisely the ambiguity this fixes. And speed: since 2026-07-26 real CDISC CORE runs in this repo (M5.4), but at ~66 s per study it cannot sit inside the accuracy loop that re-validates

on every iteration; these 17 rules cost milliseconds and act as the inline layer, with the `DISCLAIMER` string explicitly ranking itself below CORE.

INPUTS AND OUTPUTS. Input: a domain code and a produced CSV path, optionally a DM CSV path so the one cross-domain rule can run. Output shape, per rule:

```
#| skip-check
{"id": "CLIN0003", "severity": "Error", "title": "Required variables are populated",
 "checks": "No IG Required variable is blank on any record.",
 "does_not_check": "Whether the populated value is the correct one.",
 "status": "fail",
 "findings": [{"message": "AEDECOD blank on 1 record(s).", "count": 1, "examples": []}],
 "n_affected": 1}
```

plus `totals` (`rules_run`, `passed`, `failed`, `not_run`, `by_severity`) and the `disclaimer`. For our broken AE row: CLIN0003 fires on the blank `AEDECOD`, CLIN0020 fires on `05JAN2026`, and CLIN0030 fires on `SEVERE!!` — because this module's CT context comes from `ta_ct.assess_dataset(domain, csv_path)`, which resolves the real `AESEV` codelist that M5.2's `_ct_set` refused.

HOW IT WORKS. A rule is a frozen dataclass:

```
#| skip-check
@dataclass
class Rule:
    id: str
    severity: str
    title: str
    checks: str
    does_not_check: str
    fn: Callable
    cross_domain: bool = False
```

and the `rule(...)` decorator appends to the module-level `RULES` list, so adding a check is adding one decorated function — the docstring's point being that the rule set grows by what reviewers actually hit, not by padding a count. `assess` builds a `ctx` dict (`domain`, `header`, `rows` from a stdlib `csv.reader`, `ig_vars`, `ig_keys`, `ct`, `dm_subjects`) and calls each `r.fn(ctx)`; a non-empty findings list means `"fail"`, an exception inside a rule becomes a finding (`"rule raised {type}: ..."`) rather than sinking the report, and — the honest touch — a cross-domain rule with no DM supplied gets status `"not_run"`, never `"pass"`. The `ig_vars` construction carries a long comment worth reading verbatim in the source (lines 448–460): the local pack was audited against the CDISC Library and found to invent 42 variables the IG does not define and to miss 8 real ones (`AESCAN`, `AESOD`, `VSBLFL`, `LBBLFL`, `EGBLFL`, `PEBLFL`, `PEMODIFY`, `PEBODSYS`), so variable lists come from `ig_reference.variables(domain)` when its cache is readable, with the pack as fallback and natural keys always from the pack. Rule highlights beyond the structural ones you already know from M5.1: CLIN0012 doesn't just count natural-key duplicates, it diagnoses them (an always-blank key component is named as a mapping gap); CLIN0031 splits case-only CT mismatches from CLIN0030 because "this is a cheap mechanical fix, and burying it among real mapping errors hides it"; and CLIN0050 is the anti-fabrication rule born from the LB incident — for pairs like `("AEDECOD", "AETERM")` in `_CODED_PAIRS`, if every populated pair (minimum 5) is identical ignoring case, it declares no dictionary was applied, with the docstring caveat that partial identity proves nothing since "Headache" is a genuine MedDRA PT.

THE P21 BRIDGE. This file is the corrected mental model for a P21 user. A P21 report's power is breadth; this file's power is scoped honesty — 17 rules that each tell you what they didn't look at. Where the analogy misleads: P21's rules are externally versioned and citable in a submission; `CLIN` rules are

house rules, and the `DISCLAIMER` says so in the report body itself. One drift to know: that disclaimer and the module docstring both say "16 rules", but `RULES` contains 17 `@rule` registrations today — CLIN0050 was added without the prose catching up. Trust `len(RULES)`, or `rule_catalogue()`, never the string. A second, sharper drift: `services/study_spec_export.py` line 199 calls `ta_compliance.assess(ta)` with a single TA-name argument and then reads `comp.get('by_domain', ...)` — but `assess` requires `(domain, csv_path)` and never returns a `by_domain` key. That export path was written against an API that does not exist and will raise `TypeError` if exercised; the live callers (`routes_ta.py` lines 530 and 559, `scripts/conformance_watch.py` line 58) use the real signature.

THE PACKAGES. Stdlib almost throughout: `csv`, `re`, `collections.Counter`/`defaultdict`, `dataclasses`. No pandas — `_read` returns `(header, rows)` as plain lists, and `_col(header, rows, name)` is the entire column API. CT resolution delegates to `services.ta_ct` (which owns the NCI CT file) and IG metadata to `services.ig_reference`. The stdlib choice is deliberate: rules must be auditable line-by-line, and a `Counter` over tuples is easier to defend than a groupby.

TRY IT YOURSELF. The registry pattern with `does_not_check`, plus two AE rules firing — CLIN0050's all-identical heuristic and CLIN0021's complete-dates-only ordering check:

```
import csv, io
from dataclasses import dataclass
from typing import Callable

@dataclass
class Rule:
    id: str; severity: str; checks: str; does_not_check: str; fn: Callable

RULES = []
def rule(id, severity, checks, does_not_check):
    def deco(fn):
        RULES.append(Rule(id, severity, checks, does_not_check, fn))
        return fn
    return deco

@rule("MINI001", "Error", "AEDECOD differs from AETERM somewhere",
      "whether the MedDRA coding is CORRECT")
def coded_not_verbatim(rows):
    pairs = [(r["AEDECOD"], r["AETERM"]) for r in rows if r["AEDECOD"]]
    if pairs and all(c.upper() == v.upper() for c, v in pairs):
        return [f"AEDECOD == AETERM on all {len(pairs)} records: never coded"]
    return []

@rule("MINI002", "Error", "AESTDTC <= AEENDTC when both complete",
      "partial dates -- a year-only start cannot be ordered")
def start_before_end(rows):
    bad = [r for r in rows if len(r["AESTDTC"]) >= 10
           and len(r["AEENDTC"]) >= 10 and r["AESTDTC"] > r["AEENDTC"]]
    return [f"start after end on {len(bad)} record(s)"] if bad else []

CSV = """AETERM,AEDECOD,AESTDTC,AEENDTC
HEADACHE,HEADACHE,2026-01-05,2026-01-03
NAUSEA,NAUSEA,2026-02,2026-01
"""
rows = list(csv.DictReader(io.StringIO(CSV)))
for r in RULES:
    print(r.id, r.fn(rows), "| does NOT check:", r.does_not_check)
```

Note MINI002 stays silent on the second row — `2026-02 > 2026-01` looks orderable, but the rule (like CLIN0021) only compares strings of length ≥ 10, exactly because partial-date ordering is unreliable.

LIMITS. Single-dataset except CLIN0040 (subject-in-DM), and that one runs only when the caller supplies `dm_csv_path`. Everything the `does_not_check` fields say, taken together: nothing here knows whether the right value was chosen for a record — a structurally perfect dataset built from the wrong source column passes 17/17. CLIN0050's threshold (≥5 populated pairs, 100% identical) means a 4-record domain or a single hand-edited row evades it. The exception-to-finding conversion means a crashing rule looks like a data finding in the report; the message names the exception, but a skimming reader can misread it.

HOW TO IMPROVE IT. Fix the two prose/API drifts (the "16 rules" strings; the `study_spec_export.assess(ta)` call). Give `assess` a `not_run` path for CT too — today a missing CT file makes CT columns silently `not checkable` inside `ta_ct` rather than surfacing an explicit `not_run` status at the rule level the way CLIN0040 does. Then wire `rule_catalogue()` into the define.xml/reviewer's-guide pipeline so the coverage statement ships with the data.

LIFT AND REUSE. This is the most liftable file in the module. The `Rule` dataclass + decorator registry + `does_not_check` field is a complete, dependency-free pattern for any domain-rules engine (finance, lab pipelines, ETL contracts). The right interface is exactly what it has: `assess(unit, data, context=None) -> {rules: [...], totals: {...}}` plus `rule_catalogue()`. Genericise by making `RULES` a parameter of `assess` instead of module state, so two rule packs can coexist in one process.

M5.4 `/root/sdtm_review/services/core_engine.py` — running CDISC's real rules engine honestly

On 2026-07-26 someone measured what everyone assumes is a shortcut: pointing CDISC CORE at a single VS dataset with `-dp` gave 4 rules passing and 275 skipped; pointing it at the whole study folder gave 89 passing. Same data, same 430 rules. The single-file number "means nothing", the module docstring says, and that measurement — kept as a table in the docstring — is why this wrapper refuses to expose single-dataset mode at all.

WHAT IT DOES. Wraps the open-source CDISC CORE rules engine (installed at `/opt/cosa/cdisc-rules-engine`, run with `/root/tlf_studio/.venv/bin/python`) as a subprocess: stages a study's XPT files into a temp folder, invokes `core.py validate` against SDTMIG 3-4 with a pinned CT package, parses the JSON report into the app's row shape, and persists timestamped evidence under `evidence/conformance/<TA>/`. This is the only layer in either repo where a `CORE-000xxx` id genuinely means CDISC's published rule ran.

WHY IT EXISTS. The in-house layers (M5.1–M5.3, M5.5) are house rules; a sponsor-facing conformance claim needs the authoritative engine. And it needed wrapping because CORE has operational teeth: it takes ~66 s over nine domains (so it cannot run on page load), most SDTM rules join across domains (so invocation shape changes the answer — the table above), and it distinguishes "rule failed" from "rule could not evaluate" (which a naive pass-rate folds into false assurance).

INPUTS AND OUTPUTS. `run_study(domain_xpts, out_dir=None, ct_package=None, define_xml_path=None, timeout=TIMEOUT, standard=STANDARD, version=VERSION)` takes `{"AE": "/path/to/ae.xpt", ...}` and returns a dict that always carries `available` and, on any failure, `error` with **no conformance numbers at all** — "a partial number would be read as a result." On success:

```
#| skip-check
{"available": True, "rules_loaded": 430, "evaluated": 101, "passed": 89,
 "failed": 12, "execution_errors": 177, "skipped": 152, "pass_rate_pct": 88.1,
 "rules": [...], "findings": [...], "execution_error_rules": [...],
 "provenance": {"engine_version": ..., "rules_cache_sha256": ...,
                "ct_package": "sdtmct-2026-03-27", "started_utc": ..., "wall_seconds": ...}}
```

Real numbers from this box: the stored `latest.json` for Cardiology (run 2026-08-03T11:17 UTC) shows 89 passed, 12 failed, 177 execution errors, 62.7 s wall. Note `pass_rate_pct` is computed over `evaluated = pass + fail` only, and the code comments it is "never returned without execution_errors beside it."

HOW IT WORKS. Availability first: `availability()` checks four concrete paths (`core.py`, the interpreter, the cache dir, `rules*.pkl` files) and names the missing one — "a bare `False` sends the reader to the wrong file." Then `run_study`: `stage_study` copies each XPT to `<tmp>/data/<domain lower>.xpt` because CORE infers domain from filename stem; the command is built as

```
#| skip-check
cmd = [PY_BIN, "core.py", "validate",
       "-s", standard, "-v", version,
       "-ca", CACHE_DIR,
       "-d", os.path.join(tmp, "data"), "-ft", "xpt",
       "-ct", ct_package,
       "-o", out_base, "-of", "JSON",
       "-p", "disabled"]
```

with `-dxp define.xml` appended when supplied, and executed via `subprocess.run(cmd, cwd=ENGINE_DIR, capture_output=True, text=True, timeout=timeout)`. Timeout and missing-report cases each return an `error` dict, never raise into the Flask request. `_normalise` maps CORE's four statuses through `_STATUS` (`SUCCESS`→`pass`, `ISSUE REPORTED`→`fail`, `EXECUTION ERROR`→`execution_error`, `SKIPPED`→`skipped`) with the fail-safe that **anything unrecognised becomes `execution_error`**, "an unknown status is not good news" — and splits `Issue_Summary` rows containing "rule evaluation error" out of findings, because presenting "rule could not run" as a data finding is the opposite of what it says. Provenance is obsessive on purpose: `engine_version(report)` reads `CORE_Engine_Version` from the report the engine just wrote (reading the source tree would name what's checked out, not what ran); `_cache_digest()` sha256s the rules cache so the report names its rule set; `latest_ct_package()` pins the newest `sdtmct-*.pkl` (currently `sdtmct-2026-03-27` on this host) because "a conformance claim without one is unreproducible." The evidence layer: `run_and_save(ta)` takes a lock file (`os.open` with `O_CREAT|O_EXCL` — five reviewers share the app, two concurrent runs would race), calls `run_ta` (which asks `ta_registry.executed_domains(ta)` and `ta_output.load_output` for each domain's XPT via `find_xpt`, recording domains with no XPT in `executed_without_xpt` — "the difference between 'clean' and 'not looked at'"), then `save_result` writes `run_<stamp>.json` before repointing `latest.json` ("a crash between the two loses the pointer, not the evidence"). TA names pass `safe_ta` (`^[A-Za-z0-9][A-Za-z0-9_-]{0,63}$`) because `ta` arrives from a POST body and is used as a directory component — the comment spells out the `{"ta": "../../../root/.ssh"}` attack. Flask endpoints in `routes_ta.py` (`/api/ta/core/run` at line 605, status/history around 586–602) are the callers.

THE P21 BRIDGE. For a P21 user this chapter is the biggest recalibration. P21 reports have two data states per check (issue / no issue, plus "not applicable" folded away); CORE-through-this-wrapper has **four**, and keeps them apart all the way to the UI: `pass`, `fail`, `execution_error`, `skipped`. The 2026-07-26 measurement is the whole argument: run CORE the convenient way and 275 of 430 rules silently skip because their cross-domain joins have nothing to join to — the P21 habit of validating one dataset "quickly" produces a number that looks like a report and isn't one. Also note what the `DISCLAIMER`

constant says P21-style tools never print: “this checks conformance, not correctness … a clean report is not clinical review.”

THE PACKAGES. Stdlib only — `subprocess`, `tempfile`, `shutil`, `glob`, `hashlib`, `json`, `os`, `re`, `collections.Counter`, `datetime`. The heavy dependency is external: the CORE engine itself, its pickled rules cache, and a Python interpreter borrowed from another app’s venv (`/root/tlf_studio/.venv/bin/python`, overridable via `CORE_PY_BIN`). That borrowed-interpreter coupling is the fragile joint: rebuild tlf_studio’s venv and CORE dies here, though `availability()` will at least name the missing path. Alternatives: `cdisc-rules-engine` is pip-installable as a library, but the subprocess boundary is deliberate — it isolates a 66-second, memory-hungry engine from the Flask process and makes the timeout enforceable.

TRY IT YOURSELF. The wrapper pattern without CORE: invoke an external “engine” via subprocess, refuse to confuse its three statuses, and compute the pass rate over evaluated rules only:

```
import json, os, subprocess, sys, tempfile

ENGINE = r'''
import json, sys
json.dump({"Rules_Report": [
    {"core_id": "X001", "status": "SUCCESS"},
    {"core_id": "X002", "status": "ISSUE REPORTED"},
    {"core_id": "X003", "status": "EXECUTION ERROR"},
]}, open(sys.argv[1], "w"))
'''
STATUS = {"SUCCESS": "pass", "ISSUE REPORTED": "fail",
          "EXECUTION ERROR": "execution_error"}

with tempfile.TemporaryDirectory() as tmp:
    eng = os.path.join(tmp, "engine.py"); rep = os.path.join(tmp, "r.json")
    open(eng, "w").write(ENGINE)
    proc = subprocess.run([sys.executable, eng, rep],
                          capture_output=True, text=True, timeout=30)
    if not os.path.isfile(rep):                       # engine failure != clean
        raise SystemExit(f"no report (rc={proc.returncode}): {proc.stderr}")
    rules = json.load(open(rep))["Rules_Report"]

counts = {}
for r in rules:  # unknown status is NOT good news
    s = STATUS.get(r["status"], "execution_error")
    counts[s] = counts.get(s, 0) + 1
evaluated = counts.get("pass", 0) + counts.get("fail", 0)
print(counts, "| pass rate over EVALUATED only:",
      round(100 * counts.get("pass", 0) / evaluated, 1), "%")
assert counts == {"pass": 1, "fail": 1, "execution_error": 1}
```

LIMITS. The Cardiology numbers show the honest ceiling: 177 of 430 rules currently end in execution errors even in study-folder mode — mostly rules needing domains or metadata this study doesn’t stage — so “89 passed” is 89 of 101 evaluated, not of 430, and the report structure forces you to see that. `run_ta` validates only domains that emitted an XPT; a domain that executed but wrote no XPT is invisible to CORE (reported in `executed_without_xpt`, but still unvalidated). The lock file has no expiry logic beyond reporting its age; a crashed run can leave `.running` behind until manually cleared. And nothing here interprets CORE findings — it transports them.

HOW TO IMPROVE IT. Auto-stale the lock (delete `.running` older than `TIMEOUT`). Give `availability()` a version pin — today any checked-out engine passes; asserting a known-good engine version would

catch a silent upgrade changing verdicts. Classify the 177 execution errors by missing prerequisite (no DM? no TS? no define.xml?) so the number becomes a work list instead of a residue.

LIFT AND REUSE. The whole file is a template for "wrap an external validator as evidence": availability-with-named-paths, canonical staging, status normalisation with a fail-safe default, provenance from the artifact not the source tree, timestamped-file-then-pointer persistence, and a single-flight lock. The right interface is exactly `run_study(inputs) -> dict-with-available-and-either-numbers-or-error`. If you lift one idea only, take this: never return partial conformance numbers with an error; return the error alone.

M5.5 `/root/sdtm_review/services/spec_bridge.py` — human decision beats machine spec, always

The fleet mapper had adjudicated hundreds of column mappings and reviewers had signed off decisions in the review UI — and none of it could generate a single program, because the fleet spoke `{sdtm_var, source, raw_var, rule}` and the code generator spoke `{target, source, method, logic, ct}`. Two vocabularies, no translator. This file is the translator, and it decides who wins when they disagree.

WHAT IT DOES. `build_spec(shape_id, ...)` assembles a code-generator spec for one raw data shape: one row per SDTM variable in the domain, in IG order, each row sourced from — in strict precedence — a human review decision, else the adjudicated fleet mapping, else an honest `"Not Submitted"` placeholder. It returns `(spec_rows, report)` where the report names the origin of every single row.

WHY IT EXISTS. The docstring states the political core: "human decision > adjudicated spec > (nothing). A value a person signed off in the review UI overrides whatever the models agreed on. That is the whole point of the ledger — if the model's answer won, the review would be theatre." Without this module the review loop is decorative; with it, every reviewer click becomes a line in the generated program. It is validation-adjacent rather than a validator: it defines the spec that M5.6 corrects and M5.7's gates check outputs against.

INPUTS AND OUTPUTS. Input: a `shape_id` like `corpus:study:AE` (domain defaults to the segment after the last `:`), optionally the real file's `header` list. Output row (built by `_row`, with IG metadata filled from the pack):

```
#| skip-check
{"target": "AESTDTC", "label": "Start Date/Time of Adverse Event", "type": "Char",
 "core": "Exp", "source": "start_date", "method": "Direct",
 "logic": "ISO 8601 conversion of the collected onset date", "ct": "ISO8601"}
```

and a report like `{"n_rows": 40, "n_human": 3, "n_fleet": 28, "n_placeholder": 9, "has_fleet_spec": True, "provenance": [...]}` — so "why does the generated program map AESTDTC from start_date" is answerable without re-derivation.

HOW IT WORKS. The vocabulary translation is one load-bearing dict:

```
_METHOD = {
    "RAW": "Direct",
    "DERIVED": "Derived",
    "CONSTANT": "Assigned",
    "NOT_COLLECTED": "Not Submitted",
}
```

— load-bearing because, as the comment says, the generator's prompt special-cases the exact strings `'Assigned'` and `'Not Submitted'` (Module 4 told the story of the crash when `Not Submitted` columns weren't created; this is where that string is minted). `build_spec` loads IG variables via `mapping_review.load_ig()`, the adjudicated spec via `load_fleet_spec(shape_id)` (a JSON under `evidence/raw_inventory/specs/`, filename slugged by `_slug`), and human decisions via `mr.current_decisions(db_path)` filtered to this shape. Then per variable: take the human decision unless it is a `RAW` decision naming a column absent from `header` — the cross-study contamination guard ("shape_id is domain-keyed only … a human RAW decision filed under this shape_id may have been recorded against a different study's header"); else the fleet mapping; else the placeholder. `available_shapes()` reads `INDEX.json` rather than un-slugging filenames because reversing `':'→'__'` is ambiguous once a corpus name contains a dash. Callers: `ta_programs.py:215` (program build), `parity_harness.py:249`, `study_browser.py:244`, `routes_ta.py:639`.

THE P21 BRIDGE. No P21 analogue validates specs — P21 starts where data exists. The nearest clinical-programming analogue is the mapping specification document that a human reviews before programming begins; this module is that document made executable, with tracked changes. Where the analogy misleads: in a SAS shop the spec is frozen and the program follows it; here the spec is reassembled on every build from a live decision database, so "the spec" is a query result with provenance, not a versioned document — auditability comes from the per-row `origin` field, not from a signature page.

THE PACKAGES. Stdlib `json` and `os` only, plus two in-house imports (`services.mapping_review`, the IG pack behind it). The docstring makes the absence a feature: "no LLM call, no network … Converting a decision into a spec row is arithmetic, not judgement."

TRY IT YOURSELF. The precedence overlay with the header guard, in miniature:

```
METHOD = {"RAW": "Direct", "DERIVED": "Derived",
          "CONSTANT": "Assigned", "NOT_COLLECTED": "Not Submitted"}

ig_vars = ["STUDYID", "USUBJID", "AETERM", "AEDECOD", "AESTDTC"]
fleet = {"AETERM":  {"source": "RAW", "raw_var": "event_term", "rule": "copy"},
         "AESTDTC": {"source": "RAW", "raw_var": "start_date", "rule": "ISO 8601"}}
human = {"AESTDTC": {"source": "RAW", "raw_var": "onset_dt",
                     "rule": "onset_dt is the collected onset", "reviewer": "bd"}}
header = ["event_term", "onset_dt", "subject"]   # the real file's columns

spec, provenance = [], []
for name in ig_vars:
    h, f = human.get(name), fleet.get(name)
    if h and h["source"] == "RAW" and h["raw_var"] not in header:
        h = None                        # decision from another study's header
    pick, origin = (h, f"human:{h['reviewer']}") if h else \
                   (f, "fleet:adjudicated") if f else (None, "placeholder")
    spec.append({"target": name,
                 "source": pick["raw_var"] if pick and pick["source"] == "RAW" else "",
                 "method": METHOD[pick["source"]] if pick else "Not Submitted"})
    provenance.append((name, origin))
for p in provenance: print(p)
assert dict(provenance)["AESTDTC"] == "human:bd"      # human beat the fleet
assert dict(provenance)["AEDECOD"] == "placeholder"   # honest gap, not a guess
```

LIMITS. The header guard's ceiling is documented in the docstring itself: when no `header` is passed it falls back to the shape's canonical header via `mr._header_for` — "same ceiling as everywhere else this guard lives", meaning a decision recorded against a same-shaped-but-different study can still overlay if the column names coincide. A shape with no fleet spec and no decisions yields an all-placeholder

spec that will generate a program full of empty columns — valid, honest, useless. Nothing validates that a human decision's `rule` text is implementable; garbage prose flows straight into the generator's prompt.

HOW TO IMPROVE IT. Version the assembled spec: hash `(fleet file, decision set)` into the report so two builds can be proven identical. Validate human `rule` text minimally at decision time (non-empty, names a real column for RAW). Surface `n_placeholder` in the UI as a to-do count — it already exists in the report.

LIFT AND REUSE. The pattern is "three-tier override with provenance," and it applies anywhere machine output meets human review (labeling pipelines, config generation, translation memories). Genericise as `overlay(layers: list[dict], guard: Callable) -> (merged, provenance)` where layers are ordered by authority. Keep the two rules that make it trustworthy: the highest tier wins unless a guard disqualifies it, and every merged value records which tier produced it.

M5.6 `/root/sdtm_review/services/spec_enforce.py` — deterministic corrections applied after the model speaks

An attribution exercise on 2026-07-20/21 decomposed DM's oracle failure cell by cell and found something uncomfortable: every differing cell traced to a decision no model was needed for — derivable variables marked `Not Submitted` (240 cells), the study-id prefix (120), a Y/N death flag where SDTM wants Y-or-null (59), CT casing (32). Prompt fixes had chased these stochastically, "trading one failure for another." This module stops asking.

WHAT IT DOES. `enforce_spec(spec, domain, study_id=None, profile=None, pack_domain=None)` takes the LLM-produced mapping spec (a list of row dicts) and rewrites specific rows in place according to fixed rules grounded in the SDTMIG reading, returning `(spec, actions)` — where `actions` is a ledger of every change, because "an invisible correction is indistinguishable from model luck."

WHY IT EXISTS. It is the "conforms to spec" half of the loop seen from the other side: instead of validating output against a spec, it makes the spec itself deterministic wherever determinism is possible, so both language arms (R and Python) are told the same thing before any code runs. Its sibling `parity_harness.py` normalises after execution (serialization, ordinals); this module resolves real semantic decisions before it — the harness docstring explicitly hands EPOCH, `--DY` and the DM date columns to `spec_enforce` because normalising those post-hoc "would hide a defect rather than fix one."

INPUTS AND OUTPUTS. Input: spec rows shaped as in M5.5, plus optionally the engine's column `profile` (raw sample values) and the pack's `pack_domain` (per-variable CT names). Output: the mutated spec plus actions like

```
#| skip-check
{"target": "AESER", "rule": "flag_y_or_null",
 "from": {"logic": "Y if serious else 'N'"},
 "to": {"logic": "Assign 'Y' when the condition holds; otherwise leave AESER blank..."}}
```

For the AE running example, two rules bite: any `--FL` variable whose pack CT is `NY` gets the Y-or-null rewrite (`flag_y_or_null` fires when the logic contains `'N'` or "otherwise" or is empty), and `AESTDY` — study day, which needs `DM.RFSTDTC` that this extract does not carry — is forced to `method="Not Submitted"` with logic that forbids improvisation: "Do NOT read any other dataset file and do NOT invent a join or a lookup."

HOW IT WORKS. Three tiny helpers (`_row` finds a target row, `_has_source` says whether a row genuinely maps from somewhere, `_set` applies fields and appends the before/after action), then a sequence of named rules, each a conditional block: `usubjid_pinned_to_subjid_source` (the model "was observed concatenating PROJECT-SITE-SUBJECT, which no downstream instruction can reliably undo"); `age_derivable_from_brthdtc_rfstdtc` (completed years, deliberately not the naive year difference — the comment notes the SYNTH-001 oracle uses year difference and that discrepancy "is surfaced, not coded to"); the `rfxstdtc/rfxendtc_assumed_reference_*` pair, whose logic text embeds the word `ASSUMPTION` so the generated program documents its own epistemic status; `seq_canonical_derivation` ("THE canonical deterministic derivation … There is nothing for a model to decide here"); `flag_y_or_null` ; `standardized_values_already_ct_terms` (when every raw sample is a case-variant of a codelist term — checked via `services.ct.terms_for` — the answer is passthrough, because a model's enumerated `case_when` "BLANKS values its enumeration misses (60+60 cells/run)" and "a value outside the codelist must stay visibly wrong, never be blanked"); `actual_arm_equals_planned_absent_exposure` ; `context_unavailable_leave_blank` for `--DY` / `--ENRF` / `--STRF` / `VISITDY` / `EPOCH` / `--LOBXFL` / `--BLFL` when the reference date is not in this spec — each variable class annotated with the incident that put it there (MH.MHENRF came back `'BEFORE'` on 120 records; EG's Python arm set `EGLOBXFL='Y'` on 700 records with no exposure reference); and `country_already_alpha3` . Rules needing an absent input simply do not fire. Callers: `services/engine.py:331–332` (right after spec generation), `ta_programs.py:227` , `parity_harness.py:251` .

THE P21 BRIDGE. Think of this as the reviewer's red pen on a junior programmer's draft spec, applied mechanically. The P21-world habit it corrects is deeper than any validator: in a SAS shop, a wrong spec gets caught when validation output diverges and a human argues; here the divergence was nondeterministic (a model redraws the spec each run — see the memory note that temperature=0 is not reproducible), so post-hoc arguing never converges. The counter-intuitive move for a P21 mind: some corrections make the dataset worse against the oracle on purpose — `context_unavailable_leave_blank` produces blank study-day cells that count as errors in the cell-by-cell grade, because a fabricated value that grades well is the thing this codebase refuses hardest. Those blanks are the quantified "StudyContext backlog" (`docs/STUDY_CONTEXT_LIMITATIONS.md`), to be bought back "by architecture, not by letting the model guess."

THE PACKAGES. Pure Python, zero imports at module level except one lazy `from services import ct as _ctmod` inside the standardized-values rule. Every rule is string inspection over dicts. That is the point: this file's authority comes from being readable in one sitting by the person who has to defend the assumption text.

TRY IT YOURSELF. A one-rule enforcer with the action ledger — the Y-or-null flag correction on our AE spec:

```
spec = [
    {"target": "AESER", "source": "serious", "method": "Derived",
     "logic": "Y when serious=1 otherwise 'N'"},          # wrong: never 'N'
    {"target": "AESTDTC", "source": "start_date", "method": "Derived",
     "logic": "convert to ISO 8601"},                      # fine, untouched
]
ct_by_var = {"AESER": "NY", "AESTDTC": "ISO8601"}

def enforce(spec):
    actions = []
    for row in spec:
        tgt = row["target"]
        flaggish = tgt.endswith("FL") or ct_by_var.get(tgt) == "NY"
        if flaggish and ("'N'" in row["logic"] or "otherwise" in row["logic"].lower()):
            before = row["logic"]
            row["logic"] = (f"Assign 'Y' when the condition holds; otherwise "
                            f"leave {tgt} blank. SDTM flags are Y or null, never 'N'.")
            actions.append({"target": tgt, "rule": "flag_y_or_null",
                            "from": before, "to": row["logic"]})
    return spec, actions

spec, actions = enforce(spec)
for a in actions: print(a["target"], "<-", a["rule"])
assert len(actions) == 1 and actions[0]["target"] == "AESER"
assert "never 'N'" in spec[0]["logic"] and spec[1]["logic"] == "convert to ISO 8601"
```

(SDTM aside for the beginner: `AESER` is Y/N per the IG — the strict Y-or-null convention applies to `--FL` flags like `AESDTH`. The real module scopes the rule with `tgt.endswith("FL") and ct_by_var.get(tgt) == "NY"`; the demo widens it to any NY-codelist variable to show the mechanism.)

LIMITS. The rules edit logic text, then trust the generator to obey prose — enforcement of the spec is not enforcement of the program (that is what M5.7's gates and the oracle are for). Detection is string matching: `flag_y_or_null` triggers on the word "otherwise", so a logic sentence using "otherwise" legitimately still gets rewritten; `country_already_alpha3` needs sample values in `profile` and stays silent without them. Assumptions like `RFXSTDTC = RFSTDTC` are documented, not verified — with real exposure data they are simply wrong, and only the `ASSUMPTION` text in define.xml-bound logic warns anyone. The rule set is DM-heavy because DM is where the attribution was done; other domains' deterministic residue is not yet mined.

HOW TO IMPROVE IT. Drive rules from the same attribution machinery that motivated them: any failure class that recurs across N runs with an identical deterministic fix gets promoted to a rule (the annotation-engine work in this repo already closes a similar loop for mapping dispositions). Emit the `actions` ledger into define.xml comments so the assumption text reaches the reviewer, not just the manifest.

LIFT AND REUSE. The reusable object is the pattern: post-LLM deterministic patcher with an action ledger. Interface: `enforce(doc, rules, context) -> (doc, actions)` where every action carries `rule`, `from`, `to`. Use it anywhere an LLM emits structured plans that contain decisions with exactly one defensible answer — the module's epigraph generalises: if a correction can be computed, computing it beats prompting for it, every time.

M5.7 `/root/sdtm_review/services/gates.py` — millisecond checks that must be able to fail

This project has a scar to justify this file's existence, and the fix documents it in its own header. `scripts/ci_gate.py` — an earlier "gate" — validated the golden output files for conformance. As

`scripts/parity_gate.py`'s docstring puts it: "Static conformant files are always conformant — it passes with the engine deleted. It tests the answer key, not the student." A gate had been green without the pipeline it guarded ever running. The replacement executes generated code and grades produced cells; Module 8 tells that story in depth. This file is the same philosophy applied per-run: four checks that cost milliseconds, need no model, and — its docstring — "FAIL BY NAME: the point is that a whole class of defect stays fixed regardless of what the stochastic generator emits on a given run. A gate that cannot fail is not evidence — every gate here has a negative control in tests/test_gates.py."

WHAT IT DOES. `run_gates(produced_path, spec, domain, study_id, pack_domain)` reads a produced dataset (stdlib `csv.DictReader`, BOM-tolerant `utf-8-sig`) and runs four named gates over it against the mapping spec and the pack metadata, returning `{"status": "pass"|"fail", "gates": {name: {...}}}`.

WHY IT EXISTS. The generator is stochastic (measured in this project: temperature 0 does not give reproducibility), so a defect fixed by prompt engineering can resurface on any run. A deterministic post-run check pins each class of defect closed permanently: once `not_submitted_created` exists, the Module-4 crash class (`transmute()` naming a column no step created) can never silently ship again, whatever the model samples.

INPUTS AND OUTPUTS. Inputs: the produced CSV path; the spec rows (only `target` is read); the domain code; the study id; the pack's domain dict (for `variables` core flags, `keys`, and CT names). Output for our broken AE file with `AEDECOD` blank and `AESEV = "SEVERE!!"`:

```
#| skip-check
{"status": "fail",
 "gates": {"not_submitted_created": {"status": "pass", "targets": 40},
           "required_nonblank": {"status": "fail", "detail": {"AEDECOD": "1 blank row(s)"}},
           "ct_conformance": {"status": "fail", "detail": {"AESEV": ["SEVERE!!"]},
                              "checked_vars": ["AESEV", ...]},
           "key_integrity": {"status": "pass", "keys": [...], "rows": 2}}}
```

HOW IT WORKS. Four blocks in one function, no classes. `not_submitted_created`: every spec `target` must exist as a column — the direct codification of the `Not Submitted` crash lesson. `required_nonblank`: every pack `core == "Req"` variable present and populated on every row (an absent column reports as `"column absent"`, not a crash). `ct_conformance`: the three-status gate — if `ct_mod.load_codelists()` cannot read the CT package (`services/ct.py`'s `CT_PATH` is `/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv`), the gate is `"unchecked"`, because the docstring's law is "could-not-measure must never read as measured-clean"; otherwise every column with resolvable terms (`ct_mod.terms_for(name, ct)`) is checked and offenders listed (capped at 8 per variable). `key_integrity`: pack `keys` present, non-blank, unique as a tuple, `STUDYID` equal to the study id on every row, and `USUBJID` starting with `study_id + "-"`. Verdict:

```
#| skip-check
overall = "fail" if any(g.get("status") == "fail" for g in gates.values()) else "pass"
```

Note what that line does not do: `"unchecked"` does not fail the overall status — an honest limitation (the UI renders it distinctly, but a caller reading only `status` sees `"pass"` with an unmeasured CT gate). The one caller is `scripts/batch_run.py:835`, step 3c of the batch pipeline, right before the oracle comparison — gates catch structural violations cheaply so the expensive cell-by-cell grade explains only genuine mapping disagreements.

THE P21 BRIDGE. A P21 user should map "gate" to "the checks you run before you bother diffing against QC output" — but the framing difference is the important part. A P21 report is a findings list: hundreds of rows, severity-sorted, someone triages. A gate is a decision: one bit, wired into an

automated pipeline, nobody triages. The design consequences follow: gates are few (four, not four hundred), each named for the incident class it closes, and each is required to have a negative control test proving it can go red. P21 has no equivalent of a negative control — nobody feeds P21 a deliberately broken dataset to check P21 still works — and that absence is exactly how ci_gate-style false greens survive in report-centric thinking. The project's wider history makes the stakes concrete: the "proven" VPS1 engine numbers turned out to be the best 2 of 17 domains measured on SAS text similarity, not data (`docs/BASELINE_2026-07-21_DATA_GATE.md`), and the first honestly-gated baseline scored 0/11. Green that cannot fail is not green. Module 8 covers the gate ecosystem — `parity_gate.py`, `reference_gate.py`, the promote.sh deploy gate — in full.

THE PACKAGES. Stdlib `csv` plus `services.ct` for codelists. No pandas, deliberately: five hundred rows of dicts and a `set` beat a DataFrame import when the whole job is membership and blank checks, and the code stays greppable by a reviewer who does not read pandas.

TRY IT YOURSELF. A gate with three statuses and its own negative control:

```
def ct_gate(rows, codelists):
    terms = codelists.get("AESEV")
    if terms is None:
        return {"status": "unchecked", "detail": "CT package not readable"}
    bad = sorted({r["AESEV"] for r in rows if r["AESEV"] and r["AESEV"] not in terms})
    return ({"status": "fail", "detail": {"AESEV": bad}} if bad
            else {"status": "pass"})

rows_good = [{"AESEV": "MILD"}, {"AESEV": "SEVERE"}]
rows_bad  = [{"AESEV": "MILD"}, {"AESEV": "Grade 3"}]
CT = {"AESEV": {"MILD", "MODERATE", "SEVERE"}}

assert ct_gate(rows_good, CT)["status"] == "pass"
# Negative control: the gate MUST fail on planted bad data,
# else it is not evidence of anything.
assert ct_gate(rows_bad, CT)["status"] == "fail"
# Missing codelist is 'unchecked', never silently 'pass'.
assert ct_gate(rows_bad, {})["status"] == "unchecked"
print("gate verdicts:", ct_gate(rows_good, CT)["status"],
      ct_gate(rows_bad, CT)["status"], ct_gate(rows_bad, {})["status"])
```

LIMITS. Four gates cover four defect classes; everything else passes through — no date logic, no cross-domain checks, no value correctness (the oracle's job). The `"unchecked"` status does not propagate into the overall verdict, so an environment with a missing CT file yields overall `"pass"` datasets whose CT was never measured — visible in the per-gate detail, invisible in the one-bit summary. `ct_conformance` checks only columns whose codelist `terms_for` can resolve; "not checked" and "clean" remain different claims here just as in M5.1. And gates read the pack — the same pack M5.3's comment documents as imperfect — so a pack error is a gate error.

HOW TO IMPROVE IT. Make `"unchecked"` at least a distinct overall status (`"pass_with_unchecked"`), so pipelines can refuse to promote on unmeasured CT. Add the two cheapest missing gates from incident history: ISO 8601 on `--DTC` columns and the literal-`NA`-string check (both already exist as CLIN rules; gates and rules could share predicates). Long-term, derive gates from `ta_compliance` rules tagged `gate=True` instead of maintaining parallel logic.

LIFT AND REUSE. The three-status gate result (`pass`/`fail`/`unchecked`) plus the negative-control discipline is the whole export. Interface: `run_gates(artifact, contract, context) -> {"status", "gates"}` — and a repo convention that no gate merges without a test feeding it planted-bad input. If you carry one sentence out of this module, carry the docstring's: a gate that cannot fail is not evidence.

What this module still owes you

THE GATE ECOSYSTEM IN DEPTH (MODULE 8'S TERRITORY). This module told the ci_gate false-green scar in one paragraph and pointed at `scripts/parity_gate.py`; Module 8 owes you the full mechanics — how the parity gate isolates execution in temp dirs, the `SAS_POLICY` decision to remove rather than soften the false three-language-parity claim (SAS is delivered, never verified, `sas_deliverable` is a delivery decision not a verification result), `scripts/reference_gate.py`, and how `promote.sh` chains parity → security scan → boot check before prod.

WHAT NO LAYER HERE MEASURES. Every chapter repeated it because every file repeats it: five validation layers, zero of them check whether the right value was chosen for a record. Correctness in this project is the cell-by-cell oracle comparison inside `scripts/batch_run.py` — Module 4/8 material — and the honest baseline under that gate was 0/11 domains. A reader who leaves this module believing "passes CORE" means "correct" has learned the exact mistake the `DISCLAIMER` strings exist to prevent.

UNVERIFIED EDGES, STATED PLAINLY. The CORE numbers quoted (89/12/177, 62.7 s; the 4-vs-89 invocation table) are stored measurements from this host, not re-run for this book. The `study_spec_export.py assess(ta)` mismatch was found by reading, not by triggering it. The two validator forks were byte-diffed today; nothing pins them tomorrow. And the CT file at `/root/.hermes/kb/cdisc/sdtm_ct_2025-03-28.csv` is a March 2025 package while CORE pins `sdtmct-2026-03-27` — two layers, two CT vintages, one more way "conformant" is always a claim relative to an instrument.

Module 6 — Review, scoring and diagnosis

This module covers the half of the SDTM Review tool that judges output instead of producing it: the human scoring rubric, the cheap-then-strong accuracy gate, the append-only decision ledgers, the triage engine that explains why a column could not be mapped, the dependency graph, and the study-level cross-dataset checks. For a SAS/R clinical programmer, this is the QC side of the house — but it is QC by structured adjudication and deterministic checks, **not** independent reprogramming, and each chapter marks where that difference bites. Everything here was read from the live code at `/root/sdtm_review/services/` on 2026-08-03; the state store is the SQLite file `/root/sdtm_review/data/review.db` (note: `/root/sdtm_review/review.db` at the repo root is a stray zero-byte file — the code's `DB_PATH` constants all point at `data/review.db`).

M6.1 `services/review.py` — the five-score rubric behind every verdict

A reviewer has just watched the mapper generate an AE program for a dataset with no oracle — no `expected/AE.csv` to diff against. There is no machine truth available, so their judgement is the quality signal. If that judgement lands as a Slack message ("looks fine, CT is a bit off"), it evaporates. This module is where it lands instead, as five integers and a verdict, attached to a specific recorded run.

WHAT IT DOES. Persists structured reviewer scores for mapper runs into two SQLite tables, `runs` and `reviews`, inside `data/review.db`. Every review is five 1–5 scores (spec, code, controlled terminology, derivations, define.xml), one verdict (`accept` / `accept_with_changes` / `reject`), and an optional comment that becomes mandatory the moment any score drops to 2 or below or the verdict is `reject`.

WHY IT EXISTS. The module docstring says it plainly: these scores are "the only quality signal available for datasets that have no machine oracle (Tier B's 189 multi-TA files, the 22 cdisc_benchmark TAs)". Without this module, quality on oracle-less data is anecdote. With it, every judgement is attributable, timestamped, aggregatable, and — crucially — explained when it is negative, because `save_review()` raises `ValueError` on an unexplained low score: "an unexplained low score cannot drive improvement".

INPUTS AND OUTPUTS. Input to `record_run()`: a dataset id like `"tierA:CDISCPILOT01:AE"`, TA, domain, model name, coverage percentage, input SHA-256, and an oracle flag; it returns a `run_id` (either the pipeline's job id, passed in so the UI prefill matches, or a fresh `uuid4().hex`). Input to `save_review()`: that `run_id` plus the five scores, verdict and comment; output is a `review_id`. A real row from the live database:

```
run_id=fc2616d36fb5  dataset_id=tierA:CDISCPILOT01:DM  model=commandcode:mimo-v2.5-pro
reviewer=smoke  scores=4,4,3,4,3  verdict=accept_with_changes
```

`aggregate(by="domain")` returns mean scores per group, and every mean carries its `n` — the docstring's rule is "A 5.0 from one reviewer must never be presentable as equivalent to a 5.0 from twelve." `export_csv()` writes the flattened run+review join, including a computed `score_overall` (the plain mean of the five).

HOW IT WORKS. The schema is created idempotently by `init_db()` (called at Flask startup in `app.py`). `record_run()` validates `dataset_id` and `status` (must be `ok`/`error` — a failed generation is still recorded, because "it failed" is itself a finding), then inserts with `INSERT OR IGNORE` so a re-run of

`/api/profile` for the same job cannot 500 or clobber the original timestamp. `save_review()` funnels each score through `_validate_score`, which contains a detail worth stealing:

```
def _validate_score(name, value):
    # bool is a subclass of int; True would silently become 1.
    if isinstance(value, bool) or not isinstance(value, int):
        raise ValueError(f"{name} must be an integer 1-5, got {value!r}")
    if value < 1 or value > 5:
        raise ValueError(f"{name} must be between 1 and 5, got {value}")
    return value
```

It then checks the run exists (`refusing orphan review`), and only then inserts — all writes serialized under a module-level `threading.Lock` (`_WRITE_LOCK`) on top of SQLite's own locking, with WAL journal mode enabled in `_connect()`. The HTTP layer is `routes_review.py`: `POST /api/review` calls `save_review` and maps `ValueError` to HTTP 400 (user error, with the rubric reason), everything else to 500. `GET /api/review/aggregate?by=domain|ta|reviewer` calls `aggregate`, whose `by` argument is resolved through a fixed whitelist dict (`AGGREGATE_KEYS`) before being interpolated into SQL — column names never come from caller text.

There is one honesty wrinkle in the live data: several `reviews` rows carry `reviewer='gate'` with straight 5s. Those are written by `scripts/review_gate.py` (gate G5), which deliberately stores one valid review and then confirms an unexplained low score is refused — a negative control proving the validation exists, not a human opinion. That same script hardcodes `"DM"` as the domain when recording its probe run, which is why the live `runs` table shows rows with `dataset_id=tierA:CDISCPILOT01:AE` but `domain=DM`. The rows are probe artifacts; treat the `domain` column of gate-authored runs as meaningless.

THE PACKAGES. Stdlib only: `sqlite3`, `csv`, `uuid`, `threading`, `datetime`. No ORM, no pandas. The docstring's closing line explains the design: "This module knows nothing about SDTM. It only knows how to record judgement." Swapping SQLite for Postgres would cost a connection-string change and losing the zero-ops single-file deployment; nothing in the query surface needs it at this scale.

THE BRIDGE, FOR SAS/R PROGRAMMERS. This rubric looks like a QC checklist, and the temptation is to read `score_code=5` the way you would read "QC PASS" on a double-programmed table. Resist it. In double programming, PASS means an independent program produced byte-identical output. Here, `score_code=5` means one person looked at LLM-generated code and liked it — no independent derivation happened. The project learned this distinction the hard way (see memory `sdtm-data-equivalence-gate`: correctness is executed data, not SAS text that reads well). The rubric is a structured opinion store; the executed-data comparison lives elsewhere (the oracle diff and the accuracy gate, next chapter).

TRY IT YOURSELF. The whole validation core in 25 lines, stdlib only (verified running on this host):

```
import sqlite3
SCORES = ("spec", "code", "ct", "deriv", "define")
def validate(scores, verdict, comment):
    for name, v in scores.items():
        if isinstance(v, bool) or not isinstance(v, int) or not 1 <= v <= 5:
            raise ValueError(f"{name} must be an integer 1-5")
    if verdict not in ("accept", "accept_with_changes", "reject"):
        raise ValueError("unknown verdict")
    low = [n for n, v in scores.items() if v <= 2]
    if (low or verdict == "reject") and not comment.strip():
        raise ValueError("a low score without a comment cannot drive improvement")
    return True

db = sqlite3.connect(":memory:")
db.execute("CREATE TABLE reviews (run_id, reviewer, "
           + ", ".join(f"score_{s}" for s in SCORES) + ", verdict, comment)")
ok = {"spec": 4, "code": 5, "ct": 3, "deriv": 4, "define": 4}
validate(ok, "accept_with_changes", "")
db.execute("INSERT INTO reviews VALUES (?,?,?,?,?,?,?,?,?)",
           ("ae-demo-001", "reviewer1", *ok.values(), "accept_with_changes", ""))
try:
    validate({"spec": 1, "code": 5, "ct": 5, "deriv": 5, "define": 5}, "accept", "")
except ValueError as e:
    print("refused:", e)
print("stored:", db.execute("SELECT count(*) FROM reviews").fetchone()[0])
```

LIMITS. Scores are ordinal opinions, and this project has direct evidence of how noisy scored judgement is: an LLM judge in the project's sibling harness scored identical runs 21, then 19, then 17 out of 35, and the pass/fail verdict flipped between runs (memory `llm-judge-scores-are-noisy`). The rubric here is filled by humans, not an LLM — but the lesson transfers: a single 4 vs a single 3 is inside the noise band of one person's afternoon. The `n`-with-every-mean rule is the module's defense; a defense it does not have is inter-rater calibration — nothing measures whether two reviewers use the scale the same way. Also: no update or delete path exists for a mistaken review, which is good for audit and bad for typos.

HOW TO IMPROVE IT. (1) Add a `rubric_version` column now, before the five dimensions ever change — an aggregate over two silently different rubrics is worse than none. (2) Record `has_oracle` runs' machine result alongside, so human scores can be calibrated against runs where truth was available. (3) Two-reviewer agreement stats (even a plain percent-agreement per dimension) would convert "scores exist" into "scores mean something".

LIFT AND REUSE. The reusable core is exactly what the docstring claims: a domain-blind "record judgement" service — five named integer dimensions, verdict whitelist, comment-on-low-score, orphan refusal, means-carry-n. Genericize by making `SCORE_FIELDS` and `VERDICTS` constructor arguments and you have a rubric store for any review workflow. Keep the `ValueError`-means-400 contract as the interface: validation lives in the service, HTTP translation in the route.

M6.2 `services/accuracy_gate.py` — cheap model first, escalate only on failure

A synthetic AE file arrives at `/api/auto`. Running every generation on a premium model would be accurate and ruinous; running everything on a cheap model would be affordable and occasionally wrong. The 41 lines of this file are the deal between the two: let the cheap model try, let a deterministic checker grade the result, and pay for the strong model only when the grade says so.

WHAT IT DOES. `run_with_gate()` generates a mapping spec and code with a cheap model, executes the generated Python against the raw file, validates the produced CSV with the rule-based CDISC checker in `services/validator.py`, and — only if execution failed or conformance fell below threshold — repeats the whole attempt with a stronger model. The result carries a `gate` dict recording which model ran, what the cheap attempt scored, and whether escalation happened.

WHY IT EXISTS. The file's own docstring: "Delivers ~Opus accuracy at ~cheap-model cost (most runs pass on the cheap model)." Without the gate you choose one model for all traffic and eat either the cost or the error rate. Note the "most runs pass" claim is the design intent, not a number this file measures — the measured picture from the project's batch evidence is far more sober (the first honestly-gated baseline scored 0 of 11 domains data-equivalent; see Limits).

INPUTS AND OUTPUTS. Input: a dataset `profile` dict (from the upload/classify step), a `domain` string like `"AE"`, the raw filename, source path, output directory, and optional model/threshold overrides. Output: the full attempt dict —

```
{"spec": {...}, "code": {...}, "exec": {"ok": True, "n_rows": 78},
 "report": {"conformance_pct": 100.0, "verdict": "Pass", ...},
 "model": "deepseek/deepseek-v4-pro",
 "gate": {"cheap_model": "deepseek/deepseek-v4-pro",
          "cheap_conformance": 100.0, "escalated": False}}
```

For the running AE example: the cheap attempt writes `ae.csv` into the job directory; `validator.validate("<out_dir>/ae.csv", "AE")` walks the IG rules and returns `conformance_pct` and a verdict.

HOW IT WORKS. Two functions, no classes. `_attempt()` is the whole pipeline in five calls:

```
def _attempt(profile, domain, raw_name, src, out_dir, model):
    spec_res = generate_mapping_spec(profile, domain, model=model)
    code = generate_all_code(spec_res["spec"], domain, raw_name, model=model,
                             raw_path=src)
    ex = run_python_program(code["python"], src, domain, out_dir)
    report = None
    csv = os.path.join(out_dir, f"{domain.lower()}.csv")
    if ex.get("ok") and os.path.exists(csv):
        report = validator.validate(csv, domain)
    return {"spec": spec_res, "code": code, "exec": ex, "report": report, "model": model}
```

`run_with_gate()` calls `_attempt` with `CHEAP` (`"deepseek/deepseek-v4-pro"`), reads `conformance_pct` and `verdict` off the report, and passes if `exec.ok` and (`verdict == "Pass"` or `conformance >= threshold`, default 100.0). Otherwise one retry with `STRONG` (`os.environ.get("SDTM_STRONG_MODEL", "minimax/minimax-m2.5")`) — and the strong result is returned whether or not it is better; there is no best-of-two comparison. The verdict itself comes from `validator.validate`, which computes a severity-weighted penalty over rule checks (`conformance = round(100 * (1 - penalty / max_pen), 1)`) and maps any Reject/Error/Warning finding to the corresponding verdict — pure arithmetic over deterministic rules, reproducible run to run.

The one caller is `app.py`'s `/api/auto` route (`from services.accuracy_gate import run_with_gate` at line 9). That route refuses caller-supplied model ids unless they are in a four-model low-cost allowlist — a public endpoint that forwarded arbitrary model names to a per-token billing API is exactly how the project's cost incident happened.

CODE-VS-COMMENT DRIFT, TWICE. The `/api/auto` docstring in `app.py` still says "auto-escalate to Opus"; the code escalates to minimax, and `accuracy_gate.py`'s own comment explains why: STRONG "was Opus

until 2026-07-19", and the escalation path goes through OpenRouter, where Anthropic models are hard-denied by the spend guard. Trust the constant, not the route docstring. Same lesson as the `llm.py` cascade comment documented in this repo's CLAUDE.md.

THE HONESTY THREAD: WHY THE GATE'S JUDGE IS DETERMINISTIC. This is the right place to bake in the project's most important scoring lesson. Elsewhere in this project's history, an LLM was used as a judge with a 35-point rubric; identical runs scored 21, then 19, then 17, and the pass/fail verdict flipped across runs with no change in the thing being judged. The recorded conclusion (memory `llm-judge-scores-are-noisy`): a judge is useful for finding defects; a judge score is not a number you can gate on or tune against. `accuracy_gate.py` is the structural answer to that failure class — the pass/fail decision here is made by `validator.validate`, a rule-based checker whose 100.0 means the same thing every run. The LLMs in this file only generate; they never grade. Whenever you see a pipeline where an LLM's opinion of an LLM's output decides escalation, expect verdict flips; put a deterministic function at the decision point instead, even a crude one.

THE PACKAGES. Stdlib `os` plus four sibling services (`engine`, `codegen`, `executor`, `validator`). No retry library, no config framework — the gate is small enough that adding either would double its size. Swapping the escalation model is an environment variable (`SDTM_STRONG_MODEL`).

THE BRIDGE, FOR SAS/R PROGRAMMERS. The gate resembles your production/QC loop compressed into one process: produce, check, rework on failure. The mislead: your QC check is an independent reproduction; this check is conformance, not correctness. `validator.validate` confirms the AE dataset is structurally legal SDTM — required variables present, ISO dates, CT values valid. It cannot know that AESEV was populated from the wrong raw column, as long as the wrong column's values are legal severity terms. A dataset can gate through at 100.0 and still disagree with the oracle cell-by-cell. Conformance-pass is necessary, nowhere near sufficient.

TRY IT YOURSELF. The gate's control flow with stub attempts (verified running):

```
def attempt(model):
    # stand-in for generate -> execute -> validate; swap in real calls
    quality = {"cheap": 88.5, "strong": 100.0}
    conf = quality[model]
    return {"model": model, "conformance_pct": conf,
            "verdict": "Pass" if conf >= 100 else "Warning"}

def run_with_gate(threshold=100.0):
    first = attempt("cheap")
    if first["verdict"] == "Pass" or first["conformance_pct"] >= threshold:
        return {**first, "escalated": False}
    second = attempt("strong")
    return {**second, "escalated": True,
            "cheap_conformance": first["conformance_pct"]}

print(run_with_gate())              # escalates: cheap scored 88.5 < 100
print(run_with_gate(threshold=85))  # cheap is good enough at a laxer bar
```

LIMITS. (1) One escalation, then done — if the strong model also fails, the failure is returned, not looped. (2) The strong attempt regenerates from scratch rather than repairing the cheap attempt, so a near-miss costs a full second generation. (3) Nothing here measures data equivalence against an oracle; the project's measured numbers (the VPS1 "proven" figures were merely the best 2 of 17 on SAS text similarity, and the honest data-equivalence baseline was 0/11 — memory `sdtm-data-equivalence-gate`) came from a separate harness, and no per-gate escalation-rate statistic is recorded anywhere in this file, so "most runs pass on the cheap model" remains an assumption at this layer. (4) Generation is

nondeterministic even at temperature 0 (measured on glm in this project), so a gate pass is a property of this run's artifact, not of the model-prompt pair.

HOW TO IMPROVE IT. Log every gate decision (cheap score, escalated, strong score) to a table — three columns would turn the docstring's "~cheap-model cost" claim into a measured escalation rate. Second: when the produced domain has an oracle on disk, run the data diff inside the gate and make equivalence, not conformance, the pass condition.

LIFT AND REUSE. The pattern is generator/critic separation with a deterministic critic, and it ports to any LLM pipeline: `attempt(model) -> artifact`, `grade(artifact) -> number` (pure function), `gate(threshold)`. Keep the grade function importable and side-effect-free; the moment grading calls a model, you have reintroduced the 21→19→17 problem.

M6.3 `services/mapping_review.py` — human decisions become durable, auditable rules

Two models in the mapping fleet disagreed about where `AE_SEVERITY` should go. That disagreement is now a queue item, and somebody has to answer it. The answer could die in a spreadsheet — or it could be validated against ground truth, appended to a ledger no one can rewrite, and exported as a rule so the next study never asks the question again. This 612-line module is the second path.

WHAT IT DOES. Three things, per its own docstring: a **decision ledger** (the `mapping_decisions` table in `data/review.db` — append-only, a correction is a new row that supersedes, never an UPDATE), a **compliance check** run at save time against ground truth (the SDTM IG v4 knowledge pack for "does this variable exist", the raw file's real header for "does this column exist"), and a **rule exporter** that turns accumulated decisions into `knowledge/mapping_rules.json` keyed by (domain, normalized raw column), which seeds the fleet's prompt on the next study.

WHY IT EXISTS. "Those flags are a review QUEUE, not an answer." Without this module, human adjudication evaporates per-study and the same sponsor-specific question gets re-asked and re-paid-for every time. Without the append-only design, you cannot answer the auditor's question — "what did we think on date X, and who changed it?" — which a mutable table structurally cannot answer.

INPUTS AND OUTPUTS. The unit of work is one decision: `save_decision(shape_id, domain, sdtm_var, source, raw_var, rule, note, reviewer, ...)` where `source` is one of `("RAW", "DERIVED", "CONSTANT", "NOT_COLLECTED")` and `shape_id` looks like `"tierB_robustness:AE"` (corpus:domain). Output: the saved row plus any advisory warnings. Here is the real AE decision sitting in `data/review.db` today — the running example's scored-and-decided mapping:

```
decision_id=3c7deeffe77a4700a3278d06de6aed2e  ts=2026-07-27T20:00:50+00:00
reviewer=apikey:bhanuji.d@gmail.com  shape_id=tierB_robustness:AE
sdtm_var=AETERM  source=RAW  raw_var=AETERM_VERBATIM
note="SYNTH01 E2E: confirming LLM proposal (0.95) -- verbatim AE te..."
```

`build_queue()` returns the full review queue: one item per IG variable per shape, each carrying every model's proposal as dropdown `options`, the file's real `header`, and a `status`.

HOW IT WORKS. `connect()` opens `data/review.db`, applies the `CREATE TABLE IF NOT EXISTS` schema, and back-fills the `action` column via a tolerated `ALTER TABLE` (the poor-man's migration — `sqlite3.OperationalError` means "already there"). The save path is `save_decision` → `check()` → insert. `check()` is the compliance gate, and its error/warning split is the design decision worth studying: an SDTM variable that does not exist, or a raw column not in the file, makes the spec un-runnable — that is an **error** and blocks the save with `ValueError`. Conventions are **warnings** that save with the row: `--`

`SEQ` from RAW ("normally DERIVED"), `--DTC` from RAW ("must be ISO 8601"), `--DY` / `EPOCH` from RAW ("needs injected study context — DM.RFSTDTC, SE element windows"), Required variable marked NOT_COLLECTED, CT-controlled variables ("the rule must recode to those values"). A reviewer can depart from convention, with the departure recorded in the `warnings` column; they cannot save something unexecutable.

Reading is reconstruction, never stored state. `current_decisions()` replays all `action='map'` rows in `(ts, rowid)` order into a dict, so the latest write wins by arrival order:

```
def current_decisions(db_path=None):
    """(shape_id, sdtm_var) -> the LATEST decision. Supersession by timestamp."""
    conn = connect(db_path)
    rows = conn.execute(
        "SELECT * FROM mapping_decisions WHERE action = 'map' "
        "ORDER BY ts ASC, rowid ASC").fetchall()
    conn.close()
    out = {}
    for r in rows:
        out[(r["shape_id"], r["sdtm_var"])] = dict(r)
    return out
```

`lock_state()` does the same replay over `lock` / `unlock` events — locks are derived, never stored, and both `save_decision` and `add_event("unlock", ...)` enforce "the lock holder or nobody" with `PermissionError` (a `ponytail:` comment in the source flags the missing superuser override as a known ceiling). `swap_decisions()` interchanges two variables' RAW sources in one transaction — both rows or neither — for the classic "AESTDTC and AEENDTC are crossed" fix. `apply_decisions(plan)` overlays the latest human decisions onto a built plan dict at read time, so the plan file stays a pure build artifact; it also gates RAW overlays on the consuming plan actually having the column, because `shape_id` is domain-keyed and TA-blind (two studies sharing AE share a shape_id — a documented deferred fix, header-hash shape ids). `build_queue()` merges four sources (fleet proposals, adjudicated spec, real headers, human decisions) and sorts by triage priority: `disputed` (models disagreed) → `unanswered` (Required, no proposal) → `single` → `agreed` → `empty` → `decided`. Finally the learning loop: `export_rules()` delegates to `ta_knowledge.build_rules`, and `suggest_from_rules(domain, raw_var)` answers "what did we decide last time" for prefill.

Both `_normalise` and `export_rules` are thin delegations that memorialize bugs: this module once normalized `start_date` to `startdate` while `ta_knowledge` kept `start_date` — two writers keying the same rule file two ways, so lookups silently missed — and `export_rules` once wrote its own incompatible document to the same path, last-writer-wins. The fix in both cases was "exactly one definition", and the docstrings keep the scar tissue visible.

THE PACKAGES. Stdlib only by declaration: "Deliberately stdlib + the existing knowledge pack only. No LLM call happens in this module: a human's decision is the input, not something to be second-guessed by a model." `headers_by_shape()` even imports the inventory scanner by file path via `importlib.util` rather than duplicating the file-discovery logic — one definition of "what raw files exist".

THE BRIDGE, FOR SAS/R PROGRAMMERS. The ledger is your data management query log and your spec sign-off sheet fused into one table — and the append-only discipline is exactly what 21 CFR Part 11 audit trails demand, implemented in 40 lines of SQL instead of a validated document management system. The mislead: in your world, a signed-off spec row was written by someone who derived the mapping. Here, most `options` in the queue were proposed by models, and the human act is choosing among proposals validated only for structural legality. `check()` will stop `AETERM ← NOT_A_COLUMN`; it will happily record `AETERM ← AE_PT` (the preferred term column), because both columns exist and AETERM

has no codelist. Adjudication catches different errors than reprogramming does — it is strong on "is this legal" and weak on "is this the column the protocol meant".

TRY IT YOURSELF. Append-only supersession in miniature (verified running):

```
import sqlite3, uuid
from datetime import datetime, timezone

db = sqlite3.connect(":memory:")
db.execute("""CREATE TABLE decisions (
    decision_id TEXT PRIMARY KEY, ts TEXT, reviewer TEXT,
    shape_id TEXT, sdtm_var TEXT, source TEXT, raw_var TEXT)""")

def save(shape, var, source, raw, who):
    db.execute("INSERT INTO decisions VALUES (?,?,?,?,?,?,?)",
               (uuid.uuid4().hex, datetime.now(timezone.utc).isoformat(),
                who, shape, var, source, raw))

# First call, then a correction. Both rows survive; the newest one wins.
save("SYNTH-001:AE", "AESEV", "RAW", "AE_SERIOUS", "modelA")   # wrong column
save("SYNTH-001:AE", "AESEV", "RAW", "AE_SEVERITY", "bhanu")   # human fix

def current():
    out = {}
    for r in db.execute("SELECT * FROM decisions ORDER BY ts, rowid"):
        out[(r[3], r[4])] = {"source": r[5], "raw_var": r[6], "by": r[2]}
    return out

print(current()[("SYNTH-001:AE", "AESEV")])   # the fix
print("ledger rows:", db.execute("SELECT count(*) FROM decisions").fetchone()[0])
```

LIMITS. Supersession orders by `ts` at one-second precision with `rowid` as tiebreak — two decisions in the same second from different connections resolve by insert order, which is fine single-node and wrong the day this is distributed. The TA-blind `shape_id` is the biggest known ceiling (documented in the source): decisions leak across studies that share a domain, mitigated but not fixed by the column-existence gate in `apply_decisions`. And the learning loop's value is unmeasured here: `mapping_rules.json` seeds the next fleet run, but no number in this module says how many re-adjudications it actually saved.

HOW TO IMPROVE IT. Header-hash shape ids (the source's own named fix). A `supersedes` column pointing at the prior `decision_id` would make history reconstruction explicit instead of order-inferred. And count rule hits per fleet run so the learning loop earns its keep with a number.

LIFT AND REUSE. The generic asset is the validated append-only decision ledger: `check() -> (errors, warnings)`, errors block, warnings travel with the row, reads replay the log. That pattern fits any human-in-the-loop pipeline (label adjudication, config approval). Lift `save_decision`/`current_decisions`/`lock_state` and parameterize `check`; the SDTM-specific conventions are just one plug-in validator.

M6.4 `services/mapping_diagnosis.py` — explaining why a column stayed unmapped

The deterministic mapper (`ta_mapper`) refused to map `ae_start_time` and put it in the residual list with the message "no deterministic rule matched this column" — which tells the reviewer nothing they didn't know from the column being in the list. This module replaces that shrug with a diagnosis: "`ae_start_time` is the clock time belonging to `ae_start_date`, which is already mapped to AESTDTC. ISO 8601 carries date and time in ONE value — emit AESTDTC=YYYY-MM-DDThh:mm." That sentence

is the deliberately-wrong-AE-mapping half of our running example: dropping the time column silently would have lost collected precision, and this is the machinery that catches it.

WHAT IT DOES. For every column the strict matchers refused, `diagnose()` runs a **lenient** near-miss scorer (the same token comparison the matchers use, minus the thresholds) plus content-shape analysis of the actual values, and classifies the refusal into one of nine statuses. Its cardinal rule, verbatim from the docstring: "**Nothing here maps anything.** Every function returns a description of the problem for a human to resolve. A diagnosis that quietly promoted itself to a mapping would reintroduce exactly the guessing the strict matchers exist to prevent."

WHY IT EXISTS. An undifferentiated residual count is a number reviewers ignore; a triaged list ("12 near-miss, 3 ambiguous") is a queue they work. This is the module that turned the annotation engine's residuals into the 93.3%-machine / 10-judgment split recorded in the project state (memory `sdtm-annotation-engine-state`) — the statuses are what make a residual machine-dispositionable versus genuinely a human call.

INPUTS AND OUTPUTS. Input: a column name, its domain, sample values, and `claimed` — a dict mapping each SDTM target to the raw column that already won it, so the diagnosis can say who took the spot. Output: a dict with `status`, `suggestion`, a full-sentence `reason`, ranked `candidates`, and `content_shape`. Verified live on this host:

```
#| skip-check  (imports ClinCoder services; shown as the real call surface)
from services import mapping_diagnosis as md
d = md.diagnose("ae_start_time", "AE", ["08:30", "14:05"],
                claimed={"AESTDTC": {"column": "ae_start_date", "rule": "R2:alias"}})
# d["status"] == "companion";  d["suggestion"] == "combine into AESTDTC"
md.diagnose("entered_by", "AE", ["jsmith", "mjones"])["status"]  # "operational"
```

HOW IT WORKS. The status taxonomy, in the order `diagnose()` checks it — the ordering is the logic, most-machine-decidable first:

1. `operational` — matches the `_OPERATIONAL` regex (`entered`, `verified`, `created_at`, record ids…): audit-trail fields, suggestion `exclude`. Machine-decidable.
2. `pivot_candidate` — the column is a published `--TESTCD` term of its **own** Findings domain (via `test_code_home` against `ct_index.codelist(f"{domain}TESTCD")` submission values and CDISC synonyms — what turns `heart_rate_bpm` into `HR`): one record of a wide layout the pivot matcher missed, not a variable.
3. `wrong_domain` — a published TESTCD term of a different domain: `height_cm` in a DM file is one record of VS, and "Putting it in SUPP{domain} would file a vital sign under demographics." When the host domain has its own test codelist, a `caveat` is attached (CT may simply lack a synonym for this study's wording) — and that caveat is exactly what downgrades the verdict from machine-decidable to judgment.
4. `measurement_ambiguous` — numeric, ≥ `MEASUREMENT_MIN_DISTINCT` (20) distinct values, in a Findings domain: a measurement, not a qualifier; `measurement_candidates()` names the plausible TESTCD terms and the reason says choosing among them "is a PROTOCOL question — the raw column does not say". Judgment, permanently.
5. `supplemental` — nothing in the IG resembles it: belongs in `SUPP--`, not dropped.
6. `companion` — the `<stem>_time` whose `<stem>_date` already won a `--DTC` (our AE example). Machine-decidable.

7. `target_taken` — the closest target is already claimed by another column: "Two raw columns competing for one SDTM variable is a question about the STUDY, not about string similarity, so the tool will not pick." Judgment.
8. `ambiguous` — two candidates within `TIE_BAND` (0.10) of each other: "ranking them would invent a confidence the evidence does not support." Judgment.
9. `near_miss` — one clear leader below the strict threshold: "Likely correct but unproven — confirm it and add it to knowledge/raw_column_aliases.md so the next run decides it deterministically."

The scorer, `_similarity()`, is weighted Jaccard token overlap against a variable's name (1.0), label (0.9) and description (0.45) — the description down-weighted because "a description mentions many concepts and will overlap with almost anything, which is precisely how `R6:description` once mapped `qrs_duration_ms` to `EGTEST` on the strength of the word 'name'." `content_shape()` classifies the values themselves (clock time / ISO date / yes-no flag / numeric / categorical / free text) so the reviewer sees what the data is, independent of the name. `diagnose_plan()` applies all this to every entry in `plan["residual_columns"]` in place and returns a status tally; it is called from `ta_mapper.build_plan` itself (line 568) and from the `routes_ta.py` re-diagnosis endpoint. One subtle correctness note preserved in a comment: `n_distinct` must come from the plan's whole-column count, because the `examples` list holds five values, and recomputing distinctness from it "would report every column as low-cardinality and silently disable this rule".

THE PACKAGES. Stdlib `re` plus two sibling read-only knowledge modules (`ig_reference`, `ct_index`) and `ta_mapper._tokens` for the one token definition. No fuzzy-matching library — `rapidfuzz` would give better string scores, but the constants (`FLOOR = 0.20`, `TIE_BAND = 0.10`) are calibrated to this coarse Jaccard measure; swapping the scorer means re-deriving both.

THE BRIDGE, FOR SAS/R PROGRAMMERS. This is the annotated-CRF conversation you have with data management, automated: "that's an operational field", "that's really a VS record", "those two columns both claim AESTDTC — which did the protocol mean?" The taxonomy's real contribution is the machine-vs-judgment boundary: statuses 1, 2 and 6 can be auto-dispositioned; `target_taken`, `ambiguous`, and `measurement_ambiguous` are declared permanently human because they are study questions, not string questions. The mislead to avoid: `near_miss` scores look like confidence values. They are Jaccard overlaps of tokenized names — a 0.55 is not "55% probability correct", and treating the ranking as calibrated probability is exactly the invented confidence the `ambiguous` status exists to refuse.

TRY IT YOURSELF. The scorer and tie-band in 20 lines (verified running — note both AE date targets tie, so the verdict is `ambiguous`, which is correct: the column name alone cannot say start or end):

```
import re
TIE_BAND = 0.10

def tokens(text):
    return {t for t in re.split(r"[^a-z0-9]+", (text or "").lower()) if t}

def similarity(column, var_name, var_label):
    col = tokens(column)
    best = 0.0
    for text, weight in ((var_name, 1.0), (var_label, 0.9)):
        toks = tokens(text)
        if col and toks:
            best = max(best, len(col & toks) / len(col | toks) * weight)
    return round(best, 3)

ig = [("AESTDTC", "Start Date/Time of Adverse Event"),
      ("AEENDTC", "End Date/Time of Adverse Event")]
scores = sorted(((similarity("ae_event_date", n, l), n) for n, l in ig),
                reverse=True)
print(scores)                       # [(0.257, 'AESTDTC'), (0.257, 'AEENDTC')]
gap = scores[0][0] - scores[1][0]
print("ambiguous" if gap < TIE_BAND else f"near_miss -> {scores[0][1]}")
```

LIMITS. `FLOOR`, `TIE_BAND` and `MEASUREMENT_MIN_DISTINCT` are chosen constants with stated rationales, not fitted numbers — nothing measures their false-triage rate. `_TESTCD_DOMAINS` is restricted to six domains this corpus can produce, so a real study with, say, MB or PC findings gets no `wrong_domain` detection there. `content_shape` samples the first 200 values, so a column that switches format mid-file is classified by its head. And the `_OPERATIONAL` regex is anglophone: `saisie_par` sails through.

HOW TO IMPROVE IT. Feed the human's eventual disposition of each diagnosis back into a counter per status — that turns "near_miss is usually right" from folklore into a rate, and tells you which statuses are safe to auto-accept. Second, let `diagnose` read `knowledge/raw_column_aliases.md` so a confirmed near-miss visibly disappears from the next run's residuals.

LIFT AND REUSE. The liftable design is refusal with a reason: a strict system plus a lenient explainer that shares the strict system's scoring primitives but may not decide. Interface: `diagnose(item, context) -> {status, reason, suggestion, candidates}` with an ordered, closed status set split into machine-decidable and judgment. Any classifier with a "reject" bucket (document routing, entity linking) benefits from exactly this second pass.

M6.5 `services/ta_review.py` — four reviewers argue; the ledger keeps everything

A reviewer in the Cardiology workspace thinks the adjudicated `AESEV` mapping is wrong and wants it to come from `severity` instead. In most tools they would edit the spec, and the disagreement — the most informative artifact in the whole review — would vanish into the diff. Here they file a proposal, it sits beside the adjudicated row, someone else accepts or declines it later, and every step of the argument is a permanent, ordered row.

WHAT IT DOES. An append-only event ledger for therapeutic-area review: `comment`, `proposal` and `resolution` events INSERTed into the `ta_events` table of `data/review.db`, never UPDATEd or DELETEd, plus one deliberate exception — `ta_presence`, an upserted, expiring "who is looking at which domain right now" table.

WHY IT EXISTS. The docstring's argument: append-only means "four people can work the same domain at the same time with no locking, no merge, and no possibility of one overwriting another. The

disagreement between them is the valuable part; a store that resolved conflicts by picking a winner would throw away exactly what we are collecting." Replaying a variable's events in `seq` order "replays the whole argument, in order, forever."

INPUTS AND OUTPUTS. `add_comment(ta, domain, anchor_type, anchor, body, reviewer, ...)` attaches a remark to one of four anchor types — `("spec_var", "raw_column", "domain", "program_line")` — where `program_line` uses anchors like `program_r:42` and a `payload` carrying the digest of the program text it was written against, "so a regenerated program can mark it stale instead of silently re-pointing it at different code." `add_proposal(...)` requires that at least one of `source` / `raw_var` / `rule` changes ("a proposal must change something"). `resolve_proposal(proposal_id, decision, reviewer)` appends a `resolution` with `payload={"decision": "accepted"|"declined"}` and `parent_id` pointing at the proposal. Real AE rows from the live ledger:

```
Cardiology|AE|comment|AESEQ  |"The specification currently indicates that the var..."
Vaccine   |AE|comment|program_r:42|"Phase 2 check -- does this guard blank dat..."
```

Reads: `feed(ta, since)` returns everything after a cursor plus active reviewers — with the carefully-reasoned rule that `cursor` is "the highest seq actually returned, never the table's max — an empty poll must leave the client's cursor exactly where it was, or a write landing between the query and the response would be skipped forever." `open_proposals()` is a LEFT JOIN of proposals against resolutions where the resolution is NULL; `counts_by_domain()` feeds the UI's domain rail.

HOW IT WORKS. All writes funnel through one private `_append()` that stamps `uuid4` event id, UTC timestamp, and JSON payload (`sort_keys=True` for stable diffs). Two typed exceptions guard the resolution path: `UnknownProposal` (the id doesn't exist) and `AlreadyResolved` — a second decision is refused rather than appended, because it "would make 'is this open?' depend on which resolution you read, and the domain rail's open count would stop meaning anything." Attribution is enforced in the service, not just the HTTP route:

```
def _require(value: str, what: str) -> str:
    out = (value or "").strip()
    if not out:
        raise ValueError(f"{what} is required and was empty")
    return out
```

— "a caller may prove who they are, never assert it." Presence is the mirror image: `ping()` upserts one row per reviewer (`ON CONFLICT(reviewer) DO UPDATE`), and `active_reviewers()` filters to a 30-second window (`PRESENCE_WINDOW_S`). Ephemeral state gets ephemeral storage; nothing about it is worth keeping. One operational subtlety: `TA_REVIEW_DB` (env var) points a staging instance at its own ledger — because the store has no delete path by design, a smoke test against production would leave its scratch remarks in front of reviewers permanently. The live database shows this failure mode anyway: two Vaccine-TA comments read "Deployment check 2026-07-26 – ignore," appended before that guard existed and now immortal.

RELATIONSHIP TO THE OTHER TWO LEDGERS. `review.db` now holds three review stores with three granularities: `reviews` (M6.1) scores a whole run; `mapping_decisions` (M6.3) records the current answer per variable with supersession; `ta_events` records the conversation — including positions that lost. The learning loop reads two of them: `ta_knowledge.build_rules` merges `mapping_decisions` with the accepted proposals in `ta_events` into one `mapping_rules.json` under one normalizer (the single-writer fix noted in M6.3).

THE PACKAGES. Stdlib only: `sqlite3`, `json`, `uuid`, `time`, `datetime`. The event-sourcing pattern here is what people install Kafka for; at four reviewers and one process, a SQLite table with an AUTOINCREMENT `seq` is the honest version.

THE BRIDGE, FOR SAS/R PROGRAMMERS. `ta_events` is your data-review meeting minutes plus your spec change log, made queryable. The `proposal → resolution` pair maps onto "QC finding → PD response" in a double-programming log — with one inversion worth noticing: in your world the finding usually comes from an independent program's mismatch; here it comes from a reviewer's reading of an LLM-adjudicated spec. The evidence base for a proposal is opinion-about-generated-text, not a discrepancy between two derivations, so the rate of proposals tells you about reviewer attention, not about defect density — do not read a quiet ledger as a clean spec.

TRY IT YOURSELF. Event-sourced proposals in 20 lines, no database needed (verified running):

```
events = []  # the ledger: INSERT only, never UPDATE

def append(kind, reviewer, anchor, payload, parent=""):
    ev = {"seq": len(events) + 1, "kind": kind, "reviewer": reviewer,
          "anchor": anchor, "payload": payload, "parent": parent}
    events.append(ev)
    return ev

p = append("proposal", "reviewer2", "AESEV",
           {"raw_var": "severity_grade", "rule": "recode 1-4 -> MILD..SEVERE"})
append("comment", "reviewer3", "AESEV", {"body": "grade 4 has no SEV term"})
append("resolution", "lead", "AESEV", {"decision": "declined"}, parent=p["seq"])

def open_proposals():
    resolved = {e["parent"] for e in events if e["kind"] == "resolution"}
    return [e for e in events if e["kind"] == "proposal"
            and e["seq"] not in resolved]

print("open:", open_proposals())          # [] -- the lead declined it
print("full argument:", [(e["seq"], e["kind"], e["reviewer"]) for e in events])
```

LIMITS. No pagination on `events_for` — a long-lived TA replays its entire history per call (fine at 30 rows, the live count today; a ceiling later). `feed` polls; there is no push, so "real-time" is really the client's poll interval. `counts_by_domain` loads all events then all open proposals — two full scans per rail refresh. Anchors are free strings by design (that's how `program_line` arrived with no schema change), which also means a typo'd anchor orphans a comment forever; only `anchor_type` is validated. And presence is trust-based: `ping` believes whatever reviewer name the API key middleware resolved.

HOW TO IMPROVE IT. Add `LIMIT`/offset to `events_for` before it hurts. A `stale` computed flag on program-line comments (compare stored digest to current program digest) is designed for in the payload but the comparison lives client-side; hoisting it into `feed` would make staleness authoritative. If reviewer count grows, per-TA SQLite files beat one shared file — the schema already keys everything by `ta`.

LIFT AND REUSE. This is the cleanest liftable module in the set: an attributable append-only discussion ledger with typed anchors and proposal/resolution semantics, zero SDTM knowledge anywhere in it. The interface to keep: `append-only writes raising typed errors (UnknownProposal, AlreadyResolved)`, `feed(since) -> {events, cursor, presence}`, and the rule that the cursor is the highest seq returned. Any collaborative review surface — code review, label QA, document sign-off — can mount it unchanged.

M6.6 `services/mapping_graph.py` — two graphs, one alive and one dead

Open `/api/ta/study-graph` and you get a ring of eleven domain nodes — DM dark, the rest lighter — with colored arcs saying things like "DM.RFSTDTC anchors –DY" into AE. That picture answers the review question no per-variable table can: which domains can even be built standalone, and which need study context injected first? The same file also contains an older per-variable graph renderer that would crash on its first line of real work. Both halves are documented here, because knowing which half is load-bearing is the chapter.

WHAT IT DOES. The live half, `synth_study_graph_data()`, builds a domain-dependency graph for the synthetic corpus `SYNTH-001`: nodes are the eleven domains from `data/synthetic_v4/MANIFEST.json` (DM, AE, CO, DV, FA, IE, PR, QS, SE, SU, SV — 60 subjects), edges are cross-domain dependencies read off the expected CSV headers on disk. `render_study_graph_html()` turns that into a standalone SVG page (ring layout, edges bowed apart per kind so parallel DM→AE arcs stay readable, plus a per-domain raw→expected table). "Everything below is read off MANIFEST.json and the CSV headers on disk — no LLM, no guesses."

WHY IT EXISTS. The four edge kinds are exactly the cross-domain dependency classes that Module P1's study-context work identified — the things a domain program cannot compute from its own raw file:

```
subjects   DM -> every subject domain    USUBJID must exist in DM       (XD001)
study_day  DM -> any domain with --DY    DM.RFSTDTC anchors the offset
epoch      SE -> any domain with EPOCH   the SE element windows decide it
visit      SV -> any domain with VISIT*  the visit map supplies VISITNUM/VISITDY
```

An AE program that derives `AESTDY` needs `DM.RFSTDTC`; one that fills `EPOCH` needs SE. The graph makes the build order and the injection requirements visible in one glance — that is the question it answers.

INPUTS AND OUTPUTS. Input: a corpus directory (default `data/synthetic_v4/`) containing `MANIFEST.json` and the raw/expected CSVs. Output: `{study_id, subjects, nodes, edges, edge_kinds, timestamp}` where an AE node looks like `{"id": "AE", "role": "anchor", "raw_rows": 78, "expected_rows": 78, "raw_cols": 12, "expected_cols": 17}` and AE collects all four edge kinds, because `expected/AE.csv`'s header really carries `USUBJID`, `AESTDY` / `AEENDY`, and `EPOCH` (verified against the file: `STUDYID,DOMAIN,USUBJID,AESEQ,AESPID,AETERM,AEDECOD,AEBODSYS,AESEV,AESER,AEACN,AEOUT,EPOCH,AESTDTC,AEENDTC,AE`
Note AE has no VISIT column, so no SV edge — "A dependency is asserted only when the target's expected header actually carries the dependent column — read from disk, never assumed." One header subtlety is handled in code: `VISITDY` ends in `DY` but is a planned day from SV, not an RFSTDTC offset, so the study-day check excludes `VISIT*` columns.

HOW IT WORKS. `_csv_header()` reads each file's first row (returning `[]` on `OSError` rather than raising). The edge derivation is a dozen lines of set membership over each domain's expected header. The renderer places nodes on a circle with basic trigonometry, fans parallel edges apart by bowing each kind a different perpendicular distance, and inlines all CSS/SVG — no JS library, the tooltips are SVG `<title>` elements. Serving happens in `routes_ta.py` (lines 1548–1564): one JSON endpoint, one HTML endpoint.

THE DEAD HALF. The top of the file — `plan_to_graph_data()`, `render_graph_html()`, `_layout_nodes()`, and the `__main__` block — is stale against the current codebase in three independent ways, verified today:

1. It calls `mapping_review.list_decisions(ta)`; **no such function exists** in `mapping_review.py` (the API is `current_decisions()` / `all_events()`). Instant `AttributeError`.
2. It calls `ig_reference.load()`; `ig_reference.py` exports `domains()` and `variables()`, **not** `load()`.

3. It iterates `plan.get('domains', {}).items()` expecting a dict of `{domain: {"items": [...]}}`, but `ta_mapper.build_plan()` returns `plan["domains"]` as a **list** of dicts with a `variables` key — `.items()` on a list is a `TypeError` even before the missing functions bite.

Nothing routes to these functions (the only importers of `mapping_graph` in `app.py` / `routes_ta.py` call `synth_study_graph_data` and `render_study_graph_html`), so nothing crashes in production — but `services/study_spec_export.py` also calls the nonexistent `mapping_review.list_decisions(ta)`, so the same fossil API has a second, less obviously dead caller. If you rebuild this module elsewhere, take the bottom half and delete the top; its confidence-threshold buckets (`>= 0.95` mapped, `>= 0.5` ambiguous, else pending) describe a plan schema this codebase no longer produces.

THE PACKAGES. Stdlib: `csv`, `json`, `math`, `os`, `collections`, `datetime`. No graphviz, no networkx, no d3 — for eleven nodes and ~25 edges, hand-rolled SVG is smaller than any dependency's import line, and the output is a single self-contained HTML file you can email.

THE BRIDGE, FOR SAS/R PROGRAMMERS. You already hold this graph in your head as "run DM first, then SE and SV, then everything else" — the standard SDTM build order. What the explicit graph adds is checkability: the edges are asserted from the expected headers, so if a domain's expected file gains an `EPOCH` column, the graph grows the SE edge without anyone updating a diagram. The mislead: this graph shows schema-level dependency (the column exists), not data-level correctness (the values are right). A DM→AE study_day edge does not tell you `AESTDY` was computed correctly — that is Module 6.7's job and the oracle diff's job.

TRY IT YOURSELF. The whole edge-derivation idea in 20 lines (verified running; prints the six edges for a four-domain mini-study):

```
expected_headers = {
    "DM": ["STUDYID", "USUBJID", "RFSTDTC"],
    "AE": ["STUDYID", "USUBJID", "AESTDY", "EPOCH", "VISITNUM"],
    "SE": ["STUDYID", "USUBJID", "EPOCH"],
    "SV": ["STUDYID", "USUBJID", "VISITNUM"],
}
edges = []
for domain, cols in expected_headers.items():
    colset = set(cols)
    if domain != "DM" and "USUBJID" in colset:
        edges.append(("DM", domain, "subjects"))
    if domain != "DM" and any(c.endswith("DY") and not c.startswith("VISIT")
                              for c in colset):
        edges.append(("DM", domain, "study_day"))
    if domain != "SE" and "EPOCH" in colset:
        edges.append(("SE", domain, "epoch"))
    if domain != "SV" and "VISITNUM" in colset:
        edges.append(("SV", domain, "visit"))
for e in edges:
    print(e)
```

LIMITS. The graph is hardwired to the synthetic corpus layout (`MANIFEST.json` + `raw/` + `expected/`); pointing it at a real study means producing that manifest first. The four edge kinds are the only ones detected — RELREC-style links, `--LNKID` chains, and SUPP– relationships are invisible. And the dead top half is a standing trap for future maintainers: it reads as the module's main feature (it owns the docstring) while being triply broken.

HOW TO IMPROVE IT. Delete `plan_to_graph_data` / `render_graph_html` / `_layout_nodes` and the `__main__` block, or rewrite them against `current_decisions()` and the list-shaped plan — half measures leave the

trap armed. Then generalize `synth_study_graph_data(base_dir)` to derive the manifest from a directory listing so real studies get the same picture.

LIFT AND REUSE. The reusable idea is dependency edges asserted from artifact headers, never from documentation. Interface: `graph(dir) -> {nodes, edges}` where every edge rule is "target's header contains X". That works for any pipeline with file contracts — ADaM (ADSL → every BDS dataset is exactly the DM→subject-domain edge), or ETL DAGs generally.

M6.7 `services/cross_dataset.py` — defects that only exist between datasets

An AE record for subject `SYNTH-001-099` is flawless SDTM: valid dates, valid CT, unique AESEQ. It is also garbage, because no such subject exists in DM — the enrollment anchor. No single-dataset validator can see that, because the defect is not in either file; it is in the relationship between them. This module is the only validator in the codebase that reads two files at once.

WHAT IT DOES. `validate_study(study_dir)` sweeps a folder of SDTM domain CSVs and runs six study-level checks — referential integrity, STUDYID consistency, DOMAIN-column-vs-filename agreement, within-subject `--SEQ` uniqueness, one-record-per-subject DM, and DM's very existence. Returns `{status, findings[], datasets_checked, checks_run, study_id}` where `status` is `"pass"` only when no Reject/Error-severity finding exists.

WHY IT EXISTS. The docstring locates the gap precisely: "Every other validator in this codebase is scoped to ONE dataset: services/validator.py takes validate(csv_path, domain), and core_validator's run_core() passes CORE a single -dp . That leaves a whole class of real conformance defects unreachable — the ones that only exist in the relationship BETWEEN datasets." It is also explicit about what it is not: "these are structural/referential checks, NOT a reimplementation of CDISC conformance rules. The authoritative per-dataset engine remains CDISC CORE... Findings here are additive to CORE, not a substitute."

INPUTS AND OUTPUTS. Input: a directory of CSVs whose filename stems are domain codes (that convention is load-bearing: it is "what lets XD003 catch a file whose DOMAIN column disagrees with its own name"). Output findings are flat dicts. For the running example — an AE file carrying the orphan subject above — the XD001 finding is:

```
{"check_id": "XD001", "severity": "Error", "domain": "AE",
 "message": "AE references subjects not present in DM.",
 "detail": ["SYNTH-001-099"], "count": 1}
```

with `count` deliberately being distinct subjects, not rows (the inline comment says so), and `detail` capped at 10 examples so a thousand-orphan disaster stays readable.

HOW IT WORKS. `_discover()` maps `DOMAIN -> path` for every `.csv`; `_read()` loads each via `csv.DictReader`, returning `[]` on anything unreadable "rather than raising — a malformed file is reported as a finding by the caller, not by an exception that aborts the whole study." The study's identity is decided by vote: the modal `STUDYID` across every file becomes truth for XD002 — no configuration, no argument. Then the checks, in order: **XD006** — no DM present means "referential integrity cannot be assessed" (an Error naming the datasets that are present); **XD005** — DM must be one record per USUBJID; then per domain: **XD001** orphan subjects (skipped for `_NO_SUBJECT_DOMAINS = {"TA", "TE", "TI", "TS", "TV", "RELREC", "SUPPQUAL"}` — trial-design datasets legitimately have no USUBJID, and checking them "would be a false positive"); **XD002** stray STUDYIDs; **XD003** DOMAIN column vs filename; **XD004** `{dom}SEQ` unique within USUBJID, using a `defaultdict(Counter)` and only

running when the column exists in the header. `summarize()` compresses a report to one log line: `FAIL - 2 finding(s) across 11 datasets: XD001x1, XD004x1`.

Two design rules deserve the reader's attention. First, independence: the module "never imports services.engine / codegen / executor, following the rule that the thing which grades must not be the thing being graded" — the same generator/critic separation as M6.2, enforced at the import level. Second, callers: `scripts/batch_run.py` (the batch evidence pipeline) and `services/define_recon.py` import it; it takes plain CSVs and stdlib only, so it runs identically inside and outside the app.

THE PACKAGES. Stdlib `csv`, `os`, `re`, `collections` — deliberately dependency-free. pandas would shorten `_col` and the groupbys, at the price of coupling the grader to the same library stack the generated code uses; the import-independence rule is worth more than the saved lines.

THE BRIDGE, FOR SAS/R PROGRAMMERS. XD001 is the `PROC SQL` you have written a hundred times — `select USUBJID from AE where USUBJID not in (select USUBJID from DM)` — and Pinnacle 21's SD-family rules are the commercial superset of this file. The honest framing (which the module itself makes) is that this is a safety net under the generated data, not your independent QC: when the mapper's generated AE program invents or mangles a USUBJID, XD001 catches it mechanically. What it will never catch is the deeper class you check by hand: AE start dates before informed consent, deaths in AE with no DS record, `AESTDTC > AEENDTC`. The chapter-6 theme applies one last time — every automated check here is necessary, none is sufficient, and the file says so in its own docstring rather than leaving you to discover it.

TRY IT YOURSELF. XD001 and XD004 on an in-memory study (verified running):

```
from collections import Counter

dm = [{"USUBJID": "SYNTH-001-001"}, {"USUBJID": "SYNTH-001-002"}]
ae = [{"USUBJID": "SYNTH-001-001", "AESEQ": "1"},
      {"USUBJID": "SYNTH-001-001", "AESEQ": "1"},   # duplicate AESEQ
      {"USUBJID": "SYNTH-001-099", "AESEQ": "1"}]   # subject not in DM

dm_subjects = {r["USUBJID"] for r in dm}
orphans = {r["USUBJID"] for r in ae} - dm_subjects
print("XD001 orphans:", sorted(orphans))            # ['SYNTH-001-099']

seen = Counter((r["USUBJID"], r["AESEQ"]) for r in ae)
dupes = [k for k, n in seen.items() if n > 1]
print("XD004 duplicate AESEQ:", dupes)              # [('SYNTH-001-001', '1')]
```

LIMITS. Six checks is a small net: no date-window checks (AE within DM's RFSTDTC/RFENDTC), no RELREC validation, no visit-consistency against SV, no EPOCH-vs-SE-window verification — note the graph in M6.6 knows about those last two dependencies but nothing here validates their values. The modal-STUDYID vote fails on a 50/50 split (Counter's `most_common` picks arbitrarily) and calls the majority wrong side stray even if the minority is right. Everything loads into memory as lists of dicts — fine for 78-row synthetic AE files, a ceiling for a 500k-row LB. And `_read()` swallowing every exception means a half-corrupt file silently shrinks to zero rows, which then passes every check.

HOW TO IMPROVE IT. The highest-value next checks are exactly the graph's other two edges: study-day recomputation (parse `--DTC`, recompute `--DY` from `DM.RFSTDTC`, compare) and EPOCH-vs-SE windows. Second: distinguish "file unreadable" from "file empty" so a corrupt CSV becomes a finding, not a silent pass.

LIFT AND REUSE. The finding record — `{check_id, severity, domain, message, detail, count}` with capped detail and severity-driven overall status — is a clean generic shape for any multi-file

consistency checker; keep check ids stable so downstream evidence stays comparable across runs. Genericize by passing a list of check functions `(name, fn(data) -> findings)` instead of the inline sequence, and the module becomes a study-level lint framework with SDTM as one rule pack.

What this module still owes you

NUMBERS IT NEVER MEASURES. The gate's "most runs pass on the cheap model" has no recorded escalation rate; the diagnosis engine's triage statuses have no measured false-triage rate; the learning loop (`mapping_rules.json`) counts no rule hits. The measured truths that do exist live outside these files and are sobering: the VPS1 "proven" numbers were merely the best 2 of 17 on SAS text similarity, and the first data-equivalence baseline was 0/11 (memory `sdtm-data-equivalence-gate`). Whenever this module's prose says "~Opus accuracy" or "usually right", read it as design intent, not evidence.

THE JUDGE LESSON, RESTATED ONCE. The one scoring incident this project has actually measured — an LLM judge scoring identical runs 21, 19, 17 out of 35 while the verdict flipped — is why every decision point in this module is deterministic (validator arithmetic, IG membership, header membership, set differences) and every opinion (rubric scores, proposals, diagnoses) is stored as attributable data rather than used as a gate. If you rebuild any of this, preserve that split before preserving anything else.

INDEPENDENCE IT DOES NOT HAVE. Nothing in this module is independent reprogramming. The rubric scores generated artifacts; the ledgers record choices among model proposals; the diagnosis explains the mapper's own refusals; the gate's checker verifies conformance, not correctness. The only truly independent check in the set is `cross_dataset.py` — stdlib-only, import-isolated from the generators — and it covers six structural rules. A regulatory submission still needs the double-programmed comparison this tool explicitly is not.

KNOWN BROKEN CODE LEFT IN PLACE. `mapping_graph.py`'s per-variable half (three independent breakages) and `study_spec_export.py`'s call to the nonexistent `mapping_review.list_decisions` are live fossils as of 2026-08-03. The `app.py` `/api/auto` docstring still says "Opus". Immortal "ignore me" comments sit in the Vaccine TA ledger. Document-vs-code drift is a recurring pattern in this repo — its own CLAUDE.md says "trust the code, not either comment", and this module is where you practice that.

WHERE THE STATE LIVES, IN ONE PLACE. Everything reviewable persists in `/root/sdtm_review/data/review.db`: `runs` + `reviews` (run-level rubric, M6.1), `mapping_decisions` (per-variable ledger with map/comment/lock/unlock actions, M6.3), `ta_events` + `ta_presence` (the argument and who's present, M6.5). All three stores are append-only where it matters, replayed rather than mutated, and none of them can be deleted from inside the app. Back the file up with `sqlite3.Connection.backup()`, never `cp` — a rule this project wrote in blood elsewhere.

Module 7 — Export and deliverables

Everything before this module decided things: which raw file is which domain, which column feeds which SDTM variable, which rule made each call. This module is where those decisions leave the system as artifacts a human or a regulator can hold — an Excel mapping spec, a Word document, DDS JSON, define.xml, a therapeutic-area pack, a zipped review bundle, and a run report. The running example throughout is the synthetic Cardiology AE domain: the raw file `ae_raw.csv`, whose `event_term` column was mapped to `AETERM` by rule `R2:alias`, followed from `plan.json` all the way into `define.xml` and the reviewer's zip.

All paths are under `/root/sdtm_review` unless stated otherwise. Every symbol, file, and behaviour claimed here was read from the code and, where possible, checked against artifacts actually on disk on 2026-08-03 (measured: openpyxl 3.1.5, python-docx 1.2.0, Python 3.14.4 at `/usr/bin/python3`, which is the interpreter `sdtm-review.service` runs).

M7.1 `services/mapping_plan_export.py` — the mapping plan for human readers

A study lead asks: "which raw column feeds AETERM, who decided that, and on what grounds?" The answer exists — it is one entry deep inside `plan.json`, a quarter-megabyte JSON file — but nobody reviews JSON in a meeting. This module renders the same plan three ways: a filterable Excel workbook, a diffable Markdown file, and an SVG graph, plus a fleet-wide MASTER roll-up across every therapeutic area.

WHAT IT DOES. Takes the mapping plan dict built by `services/ta_mapper.build_plan()` and writes it as `plan.json`, `plan.md`, `plan.xlsx`, and `graph.html`, all in `data/mapping_plans/<TA>/`. A separate function flattens every TA's plan into `MASTER.xlsx` and `MASTER.json` at the root. Nothing here decides anything — three renderings of one source of truth, each carrying the rule id that made every mapping.

WHY IT EXISTS. Three different readers need the same plan in three shapes. A clinical programmer writing the AE program wants a per-domain sheet; a lead reviewing the study wants one sortable surface (the module's own comment: "stitching nine sheets together by hand is how a reviewer stops reviewing"); a pull-request reviewer wants a text diff, which a workbook cannot give. Without this module, `plan.json` would be the only artifact, and a plan you cannot interrogate is a plan you have to trust.

INPUTS AND OUTPUTS. Input: the plan dict — keys `ta`, `reference`, `totals`, `scope`, `datasets`, `domains` (a **list** of dicts, each with `domain`, `label`, `source_file`, `sdtm_class`, `variables`), `residual_columns`, `required_unsourced`. Output for Cardiology, verified on disk: `plan.md` line 81 reads

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

and `plan.xlsx` has exactly these sheets: `Summary`, `Routing`, `Specification (all)`, `REVIEW pending`, `REVIEW required unsourced`, `SUPP qualifiers`, `Change log`, then one sheet per domain (`AE`, `CM`, `DM`, `EG`, `EX`, `LB`, `MH`, `PE`, `VS`).

HOW IT WORKS. `scripts/build_mapping_plan.py` calls, in order: `write_json`, `write_markdown`, `write_graph`, then `write_workbook` inside a try/except (a missing openpyxl degrades, never crashes). Text renderings go through `_atomic_write`, which writes `path + ".tmp"` then `os.replace` — the comments flag these files as "read-hot" because Flask routes (`/api/ta/plan/download`, `/api/ta/plan/graph` in `routes_ta.py`) serve them straight off disk, so a reader must never observe a half-written file.

`to_markdown` builds the header stats, the rule table, the routing table, the two warning sections (residual columns, `REQUIRED-UNSOURCED` variables), then a per-domain table. `write_workbook` builds sheets through a shared `_add_sheet` helper; the interesting scar is in its comment:

```
#| skip-check  (verbatim excerpt from services/mapping_plan_export.py, _add_sheet)
    # get_column_letter, not chr(64+i): past column Z the old expression
    # fell back to "A" and re-set the first column's width once per extra
    # column, so a wide sheet came out with one enormous column and the
    # rest at default.
```

`to_graph_html` hand-rolls a three-column SVG (raw file → domain → variable) with colour keyed to the rule, not the domain — grey and dashed means "nothing decided it", deliberately the style that jumps out. `build_master(root, ...)` walks `<root>/<TA>/plan.json`, tolerates one bad plan without sinking the rest (`failures` list), and writes `MASTER.json` plus a three-sheet `MASTER.xlsx` (`Summary`, `All mappings`, `REVIEW pending` — verified on disk).

THE PACKAGES. openpyxl for the workbooks — it is already the fleet's Excel library and supports `auto_filter`, `freeze_panes`, `PatternFill` per cell after the fact, which xlsxwriter also does but xlsxwriter cannot read files back (irrelevant here) and would be a second dependency for zero gain. The graph is hand-written SVG in an f-string rather than a chart library, because the page is served behind a strict CSP and must fetch nothing. Markdown and JSON are stdlib string work. Swapping openpyxl for xlsxwriter would cost a mechanical rewrite of every `ws.cell(...)`/`ws.append(...)` call and the styling idiom; nothing conceptual.

TRY IT YOURSELF. A two-sheet spec workbook with the same styling idiom (no ClinCoder imports):

```
from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill
from openpyxl.utils import get_column_letter

plan = [("AETERM", "Req", "event_term", "R2:alias"),
        ("AESEQ",  "Req", "",           "R0:derived")]
wb = Workbook()
ws = wb.active; ws.title = "AE"
ws.append(["variable", "core", "source column", "rule"])
for c in range(1, 5):
    ws.cell(row=1, column=c).font = Font(bold=True, color="FFFFFF")
    ws.cell(row=1, column=c).fill = PatternFill("solid", fgColor="2F3A4A")
for row in plan:
    ws.append(row)
ws.freeze_panes = "A2"
ws.auto_filter.ref = ws.dimensions
for i, w in enumerate([14, 8, 24, 16], start=1):
    ws.column_dimensions[get_column_letter(i)].width = w
rv = wb.create_sheet("REVIEW pending")
rv.append(["column", "why the tool is not sure"])
rv.append(["investigator_initials", "no rule matched; 41 distinct values"])
wb.save("mini_plan.xlsx")
print("wrote mini_plan.xlsx")
```

LIMITS. The workbook and markdown can drift from `plan.json` only in the window between writes — `build_mapping_plan` writes all four in one pass, but there is no lock preventing a download during a rebuild beyond the atomic rename per file. The SVG graph does not scale: layout is fixed-width 1180px and grows vertically per variable, so a 60-variable domain makes a long page. And this module has an abandoned sibling: `services/study_spec_export.py` promises a study-level workbook (inventory, spec, change log, conformance) but is imported by nothing and is broken against today's codebase — it calls `ig_reference.load()`, `ta_ct.VAR_TO_CODELIST`, `value_domains.ct_index()`, and

`mapping_review.list_decisions()`, none of which exist (verified by grep; `mapping_review` has only `current_decisions`), calls `ta_compliance.assess(ta)` against a signature that is actually `assess(domain, csv_path, dm_csv_path=None)`, and iterates `plan.get('domains', {}).keys()` although `build_plan` returns `domains` as a list. It also has a genuine arithmetic bug: `blank_pct = (100 * inv.get('blanks', 0) / max(1, sum(1 for _ in [1])))` — the denominator is always 1, so "Blanks %" is blank count × 100. Treat the file as a fossil of an earlier plan shape, not a working exporter.

HOW TO IMPROVE IT. Delete or rewrite `study_spec_export.py` against the current plan shape. Give `build_master` an mtime skip so unchanged TAs are not re-read on every rebuild. Emit the graph's coordinates from a small layout function instead of inline arithmetic so a second rendering (e.g. per-domain zoom) becomes possible.

LIFT AND REUSE. The reusable core is the pattern, not the code: one canonical dict, N renderings, every rendering carrying decision provenance, all writes atomic because a web route serves the same file. `_atomic_write` and `_add_sheet` are the two functions worth lifting verbatim; the right generic interface is `render(plan: dict) -> str | Workbook` per format, with the plan schema pinned by a test.

M7.2 `services/docgen.py` — the spec as Word and Excel

An interviewer — or a sponsor — asks to see the mapping specification, and "here is a JSON blob" is not an answer. This 105-line module turns one domain's mapping result into the two documents every clinical data manager already knows how to read: a formatted Excel spec and a Word document with a title, a rationale section, and tables.

WHAT IT DOES. `write_excel(result, profile, classification, out_path, study_id="STUDY01")` builds a three-sheet workbook — "Mapping Spec", "Source Profile", "Accuracy & Coverage" — with a navy title bar, colour-coded Method cells, and frozen header rows. `write_word(...)` builds a .docx: heading, classification rationale, a six-column variable mapping table, a source-profile table, and an italic footer stating the document is an AI-assisted draft requiring qualified review. `app.py` line 10 imports both (`from services.docgen import write_excel, write_word`), so this is live code on the serving path.

WHY IT EXISTS. This repo began as the SDTM Mapper demo app (see the repo's own `CLAUDE.md`: built for an interview), and a demo whose output is a styled, regulatory-looking spec document lands very differently from one whose output is JSON. The document is the deliverable in that workflow. Without it the app has no downloadable spec at all.

INPUTS AND OUTPUTS. Inputs are three dicts from the mapper engine: `result` (keys used: `domain`, `ig_version`, `spec` — a list of rows with `target`, `label`, `type`, `core`, `source`, `method`, `logic`, `ct` — and `coverage`), `profile` (`n_rows`, `n_cols`, `columns` with `name`/`pct_filled`/`n_unique`/`samples`), and `classification` (`label`, `class`, `structure`, `confidence`, `rationale`). For an AE run, one spec row would be `{"target": "AETERM", "label": "Reported Term for the Adverse Event", "core": "Req", "source": "event_term", "method": "Rename", "logic": "..."}` — and `write_excel` paints its Method cell `DDEBF7` (the "Rename" blue from `method_colors`). Output: one `.xlsx` and one `.docx` at the paths you give it.

HOW IT WORKS. `write_excel` is straight-line openpyxl: a merged title row (`ws.merge_cells(start_row=1, start_column=1, end_row=1, end_column=9)`), a header row at row 4 with `Font(bold=True, color="FFFFFF")` on `PatternFill("solid", fgColor=NAVY)`, then one row per spec entry starting at row 5, with `ws.freeze_panes = "A5"`. The border is built with an idiom worth knowing: `Border(*[Side(style="thin", color="BFBFBF")]*4)` — the same `Side` splatted into left/right/top/bottom. `write_word` uses python-docx: `doc.styles["Normal"].font.name = "Calibri"`, `doc.add_heading(...,`

`level=0)`, `doc.add_table(rows=1, cols=6)` with `table.style = "Light Grid Accent 1"`, and — because header-cell bolding in python-docx has no shortcut — a double loop over `hdr[i].paragraphs` and their `runs` setting `run.bold = True`. Data rows come from `table.add_row().cells` with plain `.text` assignment, labels truncated to 40 characters and logic to 120 so the table stays printable.

THE PACKAGES. python-docx is the only serious pure-Python .docx writer; the alternatives are templating a .docx by hand (fragile ZIP+XML surgery) or generating via LibreOffice headless (a process dependency measured in hundreds of MB). Its cost is verbosity — no "make this row bold" one-liner — and no page-layout control worth the name. openpyxl again for the Excel side, same reasoning as M7.1. Verified on this host: python-docx 1.2.0 and every API named above (`Document`, `Pt`, `add_heading`, `add_table`, `table.style`, `add_row().cells`) executed cleanly under `/usr/bin/python3`. One caveat for the SAS programmer used to PROC REPORT: nothing here computes; every number in the document was computed upstream and is merely typeset.

TRY IT YOURSELF. A one-table spec document:

```
from docx import Document
from docx.shared import Pt

spec = [("AETERM", "Reported Term for the Adverse Event", "Req", "event_term"),
        ("AESEQ",  "Sequence Number",                     "Req", "(derived)")]
doc = Document()
doc.styles["Normal"].font.name = "Calibri"
doc.styles["Normal"].font.size = Pt(10)
doc.add_heading("SDTM Mapping Specification — AE", level=0)
doc.add_paragraph("Synthetic study, draft spec. Requires qualified review.")
table = doc.add_table(rows=1, cols=4)
table.style = "Light Grid Accent 1"
for i, h in enumerate(["SDTM Var", "Label", "Core", "Source"]):
    cell = table.rows[0].cells[i]
    cell.text = h
    for para in cell.paragraphs:
        for run in para.runs:
            run.bold = True
for row in spec:
    cells = table.add_row().cells
    for i, val in enumerate(row):
        cells[i].text = val
doc.save("mini_spec.docx")
print("wrote mini_spec.docx")
```

LIMITS. There is a latent bug preserved in amber on line 31: `Alignment(wrap_text=True, vertical="top", size=9) if False else Alignment(wrap_text=True, vertical="top")`. The dead branch passes `size=` to `Alignment`, which raises `TypeError: unexpected keyword argument 'size'` on openpyxl 3.1.5 (measured) — it never runs only because of the literal `if False`. The Word table truncates `logic` at 120 characters with no ellipsis marker, so a reviewer cannot tell a short derivation from a truncated one. `study_id` defaults to `"STUDY01"`, which will silently label a real study's spec with a placeholder if a caller forgets the argument. And unlike M7.1's exports there is no atomic write — `wb.save(out_path)` and `doc.save(out_path)` write in place.

HOW TO IMPROVE IT. Delete the `if False` expression outright. Append "…" on truncation. Route saves through the same tmp-then-`os.replace` idiom as `mapping_plan_export._atomic_write`. If the document must ever carry a signature/approval block, add it as data (name, role, date placeholders) rather than styling.

LIFT AND REUSE. The double-loop-to-bold-a-header-row and the `Border(*[Side(...)]*4)` idioms are the transferable bits. The right generic interface for reuse is `write_spec(rows: list[dict], meta: dict, out_path)` where `rows` is format-agnostic — this module is 90% styling and 10% shape, so genericising the shape costs little.

M7.3 `services/dds_export.py` — mapping plan becomes CDISC DDS JSON

CDISC published a reference implementation for producing Define-XML — the "Data Definition Engine", a three-stage LOAD → REFINE → GENERATE pipeline — and shipped it with the LOAD stage empty: `src/loaders/` upstream contains nothing but `.gitkeep`, because loading assumes a CDISC Library membership key. ClinCoder's mapper already knows everything a loader would need to know. This module makes the mapper be the loader: it emits the mapping plan as DDS JSON, the input format CDISC's own generator consumes.

WHAT IT DOES. `plan_to_dds(plan, ...)` converts the plan dict into one DDS JSON document — study header, `itemGroups` (one per domain), `items` (one per emitted variable, each with an `origin`), `methods` (real derivation text for `R0:derived` variables), `codeLists` (only the terms the study's produced data actually uses), and value-level metadata (`conditions`, `whereClauses`, per-ItemGroup `slices`) for Findings domains. `write(plan, path)` dumps it to `dds.json`. For a beginner: a **Data Definition Spec** here is "define.xml, but as JSON, before it becomes XML" — the machine-readable description of every dataset and variable you intend to submit.

WHY IT EXISTS. Without it, ClinCoder would need its own Define-XML serializer (M7.4's sidebar shows how that earlier attempt went). With it, the chain is: raw CSV → deterministic mapper → DDS JSON → **CDISC's own generator** → define.xml, no Library key anywhere. It also fills the placeholders CDISC's own pilot fixture leaves as `"__PLACEHOLDER__"` — `origin.type`, `keySequence`, method text — which are exactly the three things the mapper computes.

INPUTS AND OUTPUTS. Input: the plan dict plus `ta_for_values`, the TA whose produced datasets are read to measure lengths and observed codelist values. Output, verified in `data/mapping_plans/Cardiology/dds.json`: top level carries `"odmVersion": "2.0"`, `"defineVersion": "2.1.0"`, `"fileType": "Snapshot"`; the AE ItemGroup is `{"OID": "IG.AE", "keySequence": ["STUDYID", "USUBJID", "AEDECOD", "AESTDTC"], "structure": "One record per adverse event per subject", ...}` with 20 items, and the running example's item is exactly:

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

`length: 12` is not a default — it is the longest `event_term` value actually present in the produced AE dataset (`max_length` measured by the mapper; `_length()` falls back to `_DEFAULT_LENGTH` only when nothing was observed). AE's derived variables get real methods: `MT.AE.DOMAIN` ("The literal domain code – a constant, never collected") and `MT.AE.AESEQ`.

HOW IT WORKS. `plan_to_dds` first calls `build_codelists(plan, ta)` — which reads each domain's produced CSV via `services.ta_output.load_output` and `_observed_values`, resolves each variable's governing codelist through `services.ct_index`, and emits **only the terms the study uses**, never the full published list (the comment's argument: the full LBTESTCD list is 2,438 terms and would bury the six the study measures). Values in the data that are not published terms are reported in an `unpublished_values` provenance block instead of being laundered into a sponsor extension — the

module refuses to "assert the wrong value was intended". Then `build_methods(plan)` turns every `R0:derived` row into a `MethodDef` whose text comes from `ta_mapper.derived_rule()`. The per-item loop skips any variable with neither a source column nor a derived rule ("a variable nothing maps and nothing derives was not collected in this study"), assigns origins via `ORIGIN_BY_RULE` (`R0:derived` → Derived; `R1` / `R2` / `R3` / `R4` / `R2.5:learned` → Collected), with one hand-carved exception documented in `_origin()`:

```
#| skip-check  (verbatim excerpt from services/dds_export.py, _origin)
    # USUBJID derived once, in DM, concatenation -- carried into
    # other domain there. Marking Predecessor everywhere gave DM
    # Predecessor pointing itself; marking Derived everywhere claimed
    # eight domains each build it independently. Neither SDTM says.
    if row.get("variable") == "USUBJID" and (row.get("domain") or "") != "DM":
        return {"type": "Predecessor", "predecessor": "DM.USUBJID"}
```

Finally `build_value_level` walks Findings domains only, builds one `WhereClause` per `--TESTCD` value and per-test ItemDefs for `ORRES/ORRESU/STRESC/STRESN/STRESU` (`VLM_VARIABLES`), with dataType/length/unit measured per test from the produced rows. Codelist failures and VLM failures are each swallowed (`except Exception`) — a define without codelists still ships. Everything is stamped with `CT_VERSION = "2025-03-28"` and a long `x_clincoder` provenance block naming every policy and every omission (e.g. `"not_populated": ["annotatedCRF -- no CRF available this corpus"]`).

THE PACKAGES. Stdlib only — `json`, `csv`, `os`, `datetime`. That is the point: the heavy dependencies (odmlib, xmlschema) stay on CDISC's side of the process boundary (M7.4). Origin `Source` is deliberately omitted for Collected variables: CDISC's `items.py` attaches an annotated-CRF page reference to every Collected+Investigator origin, and no annotated CRF exists in this corpus — naming Investigator would put a reference to a nonexistent CRF page into a regulatory document.

BRIDGE FOR THE SAS/R PROGRAMMER. DDS JSON plays the role your define spec workbook played when you fed Pinnacle 21 Define Designer or a `%make_define` macro. The analogy misleads in one important way: your spec workbook was authored; this file is computed, and its lengths and codelist subsets are measurements of produced data, not declarations of intent. Also note DDS is a pre-release CDISC model — `services/cdisc_repos.py` pins the checkout (tag `clincoder-pinned-20260726`, `data-definition-engine` at commit `096b2bfe3`, MIT-licensed) precisely because CDISC has promised breaking changes.

TRY IT YOURSELF. Build a miniature DDS-shaped document from a two-row plan (stdlib only):

```
import json, datetime

plan = {"ta": "Demo", "domains": [{"domain": "AE", "sdtm_class": "Events",
        "source_file": "ae_raw.csv", "n_variables": 2, "variables": [
            {"variable": "AETERM", "label": "Reported Term for the Adverse Event",
             "core": "Req", "role": "Topic", "type": "Char",
             "source_column": "event_term", "rule": "R2:alias", "max_length": 12},
            {"variable": "AESEQ", "label": "Sequence Number", "core": "Req",
             "role": "Identifier", "type": "Num", "source_column": None,
             "rule": "R0:derived"}]}]}
origin = {"R0:derived": {"type": "Derived"}, "R2:alias": {"type": "Collected"}}
groups = []
for dom in plan["domains"]:
    items = [{"OID": f"IT.{dom['domain']}.{v['variable']}", "name": v["variable"],
              "mandatory": v["core"] == "Req",
              "dataType": "float" if v["type"] == "Num" else "text",
              "length": v.get("max_length") or 8,
              "origin": origin[v["rule"]]} for v in dom["variables"]]
    groups.append({"OID": f"IG.{dom['domain']}", "name": dom["domain"],
                   "items": items})
dds = {"OID": "DEF.DEMO", "defineVersion": "2.1.0", "odmVersion": "2.0",
       "creationDateTime": datetime.datetime.now(datetime.timezone.utc)
           .strftime("%Y-%m-%dT%H:%M:%SZ"), "itemGroups": groups}
print(json.dumps(dds, indent=1)[:400])
```

LIMITS. The module states its own honestly, in code comments and in the emitted provenance block, and the file's docstring is your model for how: `R2.5:learned` origins are all marked Collected although some learned rules encode genuine recodes (the `ponytail:` comment names the fix — a per-entry origin flag in `mapping_rules.json`); `Origin/Source` is unknown for LB vs AE and therefore absent; codelists exist only for TAs with executed output. One thing to know that the file does not say: the `"odmVersion": "2.0"` field describes the DDS model, not the XML that comes out — the generator stamps `ODMVersion="1.3.2"` on the actual define.xml (see M7.4). A `#| skip-check` note is not needed anywhere here; everything above was read from the emitted Cardiology artifacts.

HOW TO IMPROVE IT. The per-entry origin flag for `R2.5:learned`. A `Source` field the day dataset provenance ("central lab" vs "CRF") is recorded anywhere. Streaming `_observed_values` for large datasets (it currently holds every distinct value of every column in memory).

LIFT AND REUSE. This is the most lift-worthy module in the batch: the idea "emit the standards body's own interchange format and let their generator do the serialization" applies to any regulated-output pipeline. Genericise by splitting plan-walking from DDS-shaping; the interface is `plan_to_dds(plan, measure: Callable[[domain], observed]) -> dict`, with all measurement behind the callable.

M7.4 `services/define_pipeline.py` — define.xml via CDISC's own generator

The first define.xml this system ever produced contained `Length="__PLACEHOLDER__"`. The generator wrote it quite happily; only the XSD caught it, because `Length` is `xs:positiveInteger`. That incident is why this module exists in the shape it does: every run validates against CDISC's own schema, and `ok` is false when validation fails — "producing a define.xml that a regulator would reject is worse than producing none, and the failure is silent."

WHAT IT DOES. Three functions. `available()` — can the stage run at all, and if not, exactly why (missing virtualenv, missing generator script). `generate(dds_json, define_xml)` — runs CDISC's `define_generator.py` as a subprocess in its own virtualenv, turning M7.3's `dds.json` into `define.xml`.

`validate(define_xml)` — validates the result against `define2-1-0.xsd`, also in a subprocess. None of them raise; all return report dicts.

WHY IT EXISTS. A subprocess rather than an import, deliberately: the generator needs odmlib and xmlschema, which the ClinCoder service does not carry (verified on this host — `xmlschema` 4.3.2 imports only inside `/root/cdisc360i_p2/.venv/bin/python`, not under the service's `/usr/bin/python3`). The process boundary keeps CDISC's dependency tree out of the Flask app and makes "which generator version ran" a recordable fact: the report carries `cdisc_repos.REPOS["dde"]["commit"]` (`096b2bfe3`, pinned at tag `clincoder-pinned-20260726` under `/root/cdisc360i_p2` — see `services/cdisc_repos.py`, which exists because these repos were once found in three places at five different commits).

INPUTS AND OUTPUTS. Input: the path to `dds.json` and a destination path. Output: a report dict — on the Cardiology run that produced the on-disk artifact, the file is 86,418 bytes and its opening element (verified) is:

```
<ODM FileOID="ODM.DEFINE21.86cb2d49-..." ODMVersion="1.3.2" FileType="Snapshot"
     Originator="360i Define-XML Team" SourceSystem="odmlib"
     xmlns="http://www.cdisc.org/ns/odm/v1.3"
     xmlns:def="http://www.cdisc.org/ns/def/v2.1" def:Context="Other">
```

Inside it, the AE running example appears as `ItemGroupDef OID="IG.AE"` with `ItemDef OID="IT.AE.AETERM"` at `Length="12"`, under `MetaDataVersion ... def:DefineVersion="2.1.0"` with `def:Standard ... Name="SDTMIG" Version="3.4"`.

HOW IT WORKS. `generate` first gates on `available()`, then:

```
#| skip-check  (verbatim excerpt from services/define_pipeline.py, generate)
        proc = subprocess.run(
            [VENV_PY, os.path.basename(gen), "--template",
             os.path.abspath(dds_json), "--define", os.path.abspath(define_xml)],
            cwd=os.path.dirname(gen), capture_output=True, text=True,
            timeout=TIMEOUT)
```

with `VENV_PY` overridable via `CDISC360I_PY` and `TIMEOUT` via `DEFINE_TIMEOUT` (default 600 s). If the file exists afterwards, `validate` spawns a one-liner in the same venv — `xmlschema.XMLSchema(xsd).validate(path)` — and the final `ok` is the validation verdict, not mere file existence. `scripts/build_mapping_plan.py` wires the whole arc: `dds_export.write(plan, .../dds.json)` then `define_pipeline.generate(dds, .../define.xml)`.

NOW THE NAMESPACE TRUTH THE MODULE HEADER OF THIS STUDY PACK ASKED ABOUT. There are two define.xml producers in this codebase, and they do not agree.

The other road: `services/define_xml.py`. This is the older arm, still live — `scripts/batch_run.py` imports `generate_define` from it and calls it per domain (step 6 of a batch run), and `scripts/review_gate.py` smoke-tests it. It does not generate XML itself either; it adapts the mapper's spec rows (`target` / `label` / `type` / `core` / `ct` / `method`) into the `study_meta` dict consumed by `generate_define_xml` in a different external tool, `/root/define-maker` (path overridable via `DEFINE_MAKER_HOME`). Its one substantive fix: SDTMIG Core was never applied upstream, so every ItemRef came out `Mandatory="No"` including STUDYID/USUBJID — `CORE_TO_MANDATORY` maps `REQ` → `"Yes"` and nothing else. Its `generate_define(spec, domain, study_id, out_path)` honours a "never raises" contract, returning `{"ok": False, "error": ...}` on any failure so a broken define cannot block a review run.

Here is the defect, stated from what the code and artifacts actually contain today:

- `services/define_xml.py` lines 30–31 declare `ODM_NS = "http://www.cdisc.org/ns/odm/v2.1"` and `DEF_NS = "http://www.cdisc.org/ns/def/v2.1"`, and `/root/define-maker/core/define_generator.py` line 24 confirms the generator really writes `xmlns` = `.../odm/v2.1` (verified by grep).
- **There is no `odm/v2.1` namespace in the Define-XML 2.1 standard.** Define-XML v2.1 is built on ODM 1.3.2, whose namespace is `http://www.cdisc.org/ns/odm/v1.3` — exactly what the CDISC generator's real output uses (`ODMVersion="1.3.2"`, `xmlns=".../odm/v1.3"`, verified in `data/mapping_plans/Cardiology/define.xml`). So the define-maker arm emits documents in a namespace no published schema binds, and no XSD validation exists on that arm to catch it.
- `services/define_xml.py` carries its own honest LIMITATIONS block (lines 310–326) listing what Pinnacle 21 would still flag: no `Purpose="Tabulation"`, no `def:ArchiveLocationID` / `def:leaf`, `def:Origin` written as element text where 2.1 wants `<def:Origin Type="Collected"/>`, no MethodDef (so derivation `logic` lands in Description), and text double-escaped by the generator's `_escape()` so `&` serialises as `&amp;`.

One further observed drift on the good arm: `dds_export` declares `"not_populated": ["annotatedCRF -- no CRF available this corpus"]`, yet the emitted define.xml contains `<def:AnnotatedCRF><def:DocumentRef leafID="LF.acrf"/>` — the CDISC generator inserts the reference on its own. Whether a matching `def:leaf` exists in the file was not checked; the XSD passed, but XSD validity does not prove the leaf reference resolves.

THE PACKAGES. Stdlib `subprocess` + `os` here; the XML work all happens on the other side of the boundary (odmlib serializes, xmlschema validates). This is the correct call versus vendoring lxml + a schema into the service: the schema, the generator, and the model version all travel together in the pinned checkout.

BRIDGE. For the programmer who has run Pinnacle 21 Community over a hand-built define: `validate()` is only the schema check — roughly P21's "XML is well-formed and schema-valid" line, nothing more. It will not catch a dangling CodeListRef target, a wrong Origin, or an ItemGroup that doesn't match the data (that last one is M7.5's job). Do not read `"ok": true` as "P21-clean".

TRY IT YOURSELF. A minimal, correctly-namespaced Define-XML skeleton with stdlib ElementTree, parsed back the way M7.5 will:

```
import xml.etree.ElementTree as ET

ODM = "http://www.cdisc.org/ns/odm/v1.3"
DEF = "http://www.cdisc.org/ns/def/v2.1"
ET.register_namespace("", ODM)
ET.register_namespace("def", DEF)
root = ET.Element(f"{{{ODM}}}ODM", ODMVersion="1.3.2", FileType="Snapshot")
study = ET.SubElement(root, f"{{{ODM}}}Study", OID="STD.DEMO")
mdv = ET.SubElement(study, f"{{{ODM}}}MetaDataVersion", OID="MDV.1",
                    Name="Demo", **{f"{{{DEF}}}DefineVersion": "2.1.0"})
ig = ET.SubElement(mdv, f"{{{ODM}}}ItemGroupDef", OID="IG.AE", Name="AE",
                   Repeating="Yes")
ET.SubElement(ig, f"{{{ODM}}}ItemRef", ItemOID="IT.AE.AETERM", Mandatory="Yes")
item = ET.SubElement(mdv, f"{{{ODM}}}ItemDef", OID="IT.AE.AETERM",
                     Name="AETERM", DataType="text", Length="12")
xml_bytes = ET.tostring(root, encoding="utf-8", xml_declaration=True)
open("mini_define.xml", "wb").write(xml_bytes)
back = ET.parse("mini_define.xml").getroot()
names = [i.get("Name") for i in back.iter(f"{{{ODM}}}ItemDef")]
print("ItemDefs found:", names)   # ['AETERM'] — namespace must match to find them
```

Stdlib `ElementTree` is fine here because you are parsing a file you just wrote yourself. For a define.xml that arrives from outside — a sponsor, a vendor — parse with `defusedxml.ElementTree` instead: stdlib parsers accept external entities (XXE) and nested-entity expansion by default. That is precisely the choice `services/define_recon.py` makes in M7.5.

LIMITS. `generate` trusts the venv: a stale venv with an old odmlib produces whatever it produces, and only the pinned-commit field in the report tells you which code ran. The 600-second timeout is per stage, unmeasured against real large studies. On the legacy arm, everything in the LIMITATIONS list stands, plus the namespace defect above; `batch_run` then feeds those legacy files to `define_recon`, which is the one consumer tuned to that namespace — see M7.5.

HOW TO IMPROVE IT. Retire the define-maker arm and route `batch_run` through DDS → CDISC generator, which would fix the namespace, the Origin shape, and the double-escaping in one move; the blocker is that `batch_run` works per-domain from a spec, while `dds_export` wants a full plan. Add a post-validate link check (every `leafID` / `CodeListRef` resolves) — cheap with ElementTree, and exactly the class of error XSD misses.

LIFT AND REUSE. The pattern to lift is `available()/generate()/validate()` with never-raise report dicts and a pinned-commit stamp in every result. Any wrapper around somebody else's generator — SAS via subprocess, a rendering engine, a compiler — benefits from the same three verbs.

M7.5 `services/define_recon.py` — does the define describe the data

The define.xml was generated from the mapping spec. The dataset was produced by generated code. Those are two different paths out of the same plan, and until this module existed nothing compared them: the define could declare a variable the data never had, or `Length="8"` on a field carrying 40 characters, and every existing check would still pass — `validator.py` only sees the dataset, `define_xml.py` only sees the spec. In regulatory terms: does the submission's metadata actually describe the submission's data. If it doesn't, some sponsor notices first.

WHAT IT DOES. `reconcile(dataset_path, define_path, domain)` parses one define.xml and one produced CSV and runs six cross-checks, returning `{status, findings[], domain, variables_in_define, variables_in_dataset, checks_run}`. `summarize(report)` gives the one-line version for logs (`PASS - AE: 20 data vars vs 20 define vars, reconciled`, or a `FAIL` line with per-check counts). It is layer (c) of the compliance stack the docstring names: (a) per-dataset `validator.py` + CDISC CORE, (b) cross-dataset `cross_dataset.py`, (c) dataset-vs-define — this module.

WHY IT EXISTS. It closes the last gap a two-path pipeline leaves open. `scripts/batch_run.py` line 885 calls it right after generating the per-domain define (`define_recon.reconcile(produced, dx, domain)`), and a recon `fail` feeds the run's overall verdict (line 911).

INPUTS AND OUTPUTS. Inputs: a CSV path and a define.xml path. Output findings each carry `{check_id, severity, variable, message, detail, count}` with detail capped at `_MAX_DETAIL = 10` entries so a wholly-wrong column cannot produce megabyte findings. The checks, in code order: **DR000** either input unusable (Reject, stop — "do not pretend to reconcile"); **DR001** variable in dataset, absent from define; **DR002** in define, absent from dataset; then, for variables present on both sides: **DR003** values longer than the declared `Length` — the docstring's own motivating example; **DR004** `Mandatory="Yes"` variable with empty values (blank, `.`, and similar count as null via `_NULLISH`); **DR005** values outside the declared codelist; **DR006** declared numeric DataType but non-numeric values (tested with

`float(v)`). Status is `fail` if any finding's severity is in `_SEV_FAIL = {"Reject", "Error"}`. For the AE example: if `map.R` dropped `AESER` but the define declares it, DR002 fires with `variable="AESER"`.

HOW IT WORKS. `parse_define` builds three indexes from the XML: codelist OID → set of `CodedValue`s (from both `CodeListItem` and `EnumeratedItem`), ItemOID → Mandatory (read from `ItemRef`, because "Mandatory lives on ItemRef (usage), not ItemDef (definition)" — a distinction hand-rolled define tooling regularly gets wrong), and NAME → `{oid, type, length, mandatory, codelist_oid, codelist_values}` from `ItemDef`. Any parse failure returns `({}, message)` rather than raising. The import at the top is a security decision worth quoting:

```
# defusedxml, not stdlib ElementTree: define.xml frequently supplied BY
# SPONSOR, i.e. it untrusted input arriving outside. ...
try:
    from defusedxml import ElementTree as ET
    _XML_HARDENED = True
except ImportError:  # pragma: no cover
    import xml.etree.ElementTree as ET
    _XML_HARDENED = False
```

(On this host defusedxml is installed system-wide, so `_XML_HARDENED` is True in the running service — measured, not assumed.) The module is stdlib-plus-defusedxml only and never imports engine/codegen/executor: "the grader must not import the thing graded."

THE ONE DEFECT YOU MUST KNOW. Line 38 pins the namespace: `ODM = "{http://www.cdisc.org/ns/odm/v2.1}"`. That matches the legacy define-maker output (M7.4's sidebar) — which is what `batch_run` feeds it, so the pairing works today — but it does **not** match the CDISC pipeline's real output, whose namespace is `odm/v1.3`. Point `reconcile` at `data/mapping_plans/Cardiology/define.xml` and `root.iter(f"{ODM}ItemDef")` finds zero items: `parse_define` returns an empty index without error, `define_vars` is empty, and DR001 flags every dataset variable as undocumented. The failure mode is loud (a wall of DR001 errors), not silent, but it is wrong loud: the report says "your define declares nothing" when the truth is "I parsed with the wrong namespace." The two arms of the define pipeline are namespace-incompatible, and recon is welded to the nonstandard one.

THE PACKAGES. defusedxml with stdlib fallback, `csv`, `collections.Counter`. lxml would add XPath and schema hooks nobody needs here — the queries are all `iter(tag)` — at the cost of a compiled dependency in the grading path.

BRIDGE. This is a homegrown, six-check subset of what Pinnacle 21's define-vs-data checks do (P21 has hundreds). The analogy misleads if it makes you trust a recon PASS as submission-readiness: DR-checks cover presence, length overflow, mandatory population, codelist membership, and numeric-ness — not ISO 8601 date validity, not cross-domain key integrity, not Origin/Method faithfulness, and DR003-DR006 run only on variables present on both sides, so a variable DR001/DR002 already flagged gets no value-level scrutiny at all.

TRY IT YOURSELF. A miniature reconcile, stdlib only, trusted input:

```
import csv, io, xml.etree.ElementTree as ET

NS = "{http://www.cdisc.org/ns/odm/v1.3}"
define = """<ODM xmlns="http://www.cdisc.org/ns/odm/v1.3">
 <ItemGroupDef OID="IG.AE"><ItemRef ItemOID="IT.AE.AETERM" Mandatory="Yes"/>
  <ItemRef ItemOID="IT.AE.AESER" Mandatory="Yes"/></ItemGroupDef>
 <ItemDef OID="IT.AE.AETERM" Name="AETERM" DataType="text"/>
 <ItemDef OID="IT.AE.AESER" Name="AESER" DataType="text"/>
</ODM>"""
data = "USUBJID,AETERM\nDEMO-001,HEADACHE\nDEMO-002,NAUSEA\n"

root = ET.fromstring(define)
declared = {i.get("Name") for i in root.iter(f"{NS}ItemDef")}
rows = list(csv.DictReader(io.StringIO(data)))
present = set(rows[0].keys())
print("DR001 undocumented in define:", sorted(present - declared))
print("DR002 declared but absent:   ", sorted(declared - present))
assert "AESER" in (declared - present)   # the dropped-variable finding
```

LIMITS. Beyond the namespace weld: DR005 compares raw string values, so a codelist mismatch in case ("mild" vs "MILD") is a finding even when a submission reviewer might call it a mapping-step issue; `_read_dataset` loads the whole CSV into memory; and severities are fixed — every value-level finding is an Error, no warning tier.

HOW TO IMPROVE IT. Accept both namespaces: try `odm/v1.3`, fall back to `odm/v2.1`, and report which one matched (two lines, and it un-welds recon from the legacy arm). Emit `_XML_HARDENED` into the report so an unhardened environment is visible in evidence.

LIFT AND REUSE. `parse_define` alone is a reusable Define-XML reader for presence/codelist checks — the right interface is `parse_define(path, namespaces=("odm/v1.3", "odm/v2.1")) -> (items, error)`. The finding dict shape (`check_id` / `severity` / `variable` / `detail` / `count`, detail capped) is a good lingua franca for any rule engine feeding a dashboard.

M7.6 `services/ta_pack.py` — one therapeutic area boxed for review

Four reviewers are about to argue about the Cardiology mappings. What they need on the table is not a server login: it is the raw datasets, the SDTM specification for each domain, and — crucially — one row per raw column saying where that column went. That last artifact is the coverage report, and its invariant is the reason this module can claim nothing was silently dropped.

WHAT IT DOES. `build_pack(ta)` assembles a single dict for one therapeutic area from the Tier B corpus (`corpus/tierB_robustness/<TA>/*.csv`) and the fleet-adjudicated specs (`evidence/raw_inventory/specs/tierB_robustness*.json`): per domain, the raw header, row count, the mapping spec (or a plain statement that none exists), and a coverage row for every raw column. It derives an SV (Subject Visits) spec deterministically from the visit-bearing files, computes a study-level `visit_map`, and stamps full provenance. `write_specs_workbook` and `write_raw_workbook` render the pack as Excel — explicitly "a view of the pack, never a second source of truth."

WHY IT EXISTS. Three deliberate properties, from the docstring: **deterministic** (no model calls, no clock in the payload that matters — same corpus + same specs = byte-identical pack, pinned by `test_pack_build_is_deterministic`); **complete about coverage** (every raw column gets exactly one coverage row, pinned by `test_every_raw_column_appears_exactly_once`); **silent where it does not know** (unmapped columns get a suggested disposition only when a committed rule in `knowledge/coverage_hints.json` fires — no rule, no hint). And one thing a pack is NOT: evidence. The `DISCLAIMER`

constant travels inside every pack: two models agreeing is a confidence signal, nothing here has been executed, Tier B has no value oracle.

INPUTS AND OUTPUTS. Input: a TA name — `build_pack` raises `UnknownTA` (carrying the list of TAs that do exist) for anything not both on disk and registered in `corpus/MANIFEST.json` (the manifest filter keeps E2E fixtures like a dropped-in SYNTH01 from masquerading as one of the registered Tier B areas; a missing manifest fails open). Output: the pack dict — `ta`, `corpus`, `disclaimer`, `totals` (`domains`, `spec_rows`, `raw_rows`, `raw_columns`, `unmapped_columns`), `domains`, `provenance`. For AE: the domain entry has `raw_file: "ae_raw.csv"` (domain inferred by `_domain_from_filename`), `raw_columns` straight from the header, `mappings` from the adjudicated spec with `source_ref`/`derived_from` annotations, and a coverage row like `{"column": "event_term", "status": "MAPPED", "consumed_by": ["AETERM"], "hint": None}`.

HOW IT WORKS. The build order matters and is itself a lesson: `provenance.git_snapshot()` is taken **before** reading a single input ("stamping it afterwards is how an artifact ends up naming a commit created during its own build"), and taken again at the end so `drift_note` can report if the tree moved mid-build. Then per CSV: `_read_csv` (plain `csv.reader`, blank-preserving — `""` stays `""`, never NaN), `attach_spec`, `annotate_sources`, `coverage_for`. The clever, careful part is the text analysis in `annotate_sources`: a DERIVED spec row (like AESEQ) names its inputs only in prose, so `_mentions(col, rule)` finds raw columns referenced in the rule text — whole-token regex, never substring, so `date` cannot match inside `start_date`:

```
#| skip-check  (verbatim excerpt from services/ta_pack.py, _mentions)
    return re.search(r"(?<![A-Za-z0-9_])" + re.escape(column) + r"(?![A-Za-z0-9_])",
                     text, re.IGNORECASE) is not None
```

And `_near_negation` handles rules like "Collection time, **not** analysis_date, defines LBDTC": a column whose every mention sits in a negating sentence is surfaced as uncertain, never counted as coverage. The docstring records why it doesn't try to be smarter — on the four real cases in the Cardiology specs, a resolving heuristic would have been wrong on three, and "between the two available errors, a false UNMAPPED costs a reviewer thirty seconds; a false MAPPED hides a dropped column, which is the one thing this report exists to prevent." `derive_sv` builds the SV spec from `VISIT_DATE_COLUMN` — a committed per-domain map (VS→`assessment_date`, LB→`collection_date`, …) because the naive "first column containing 'date'" picks LB's `analysis_date`, the day the lab ran the sample, not the day the subject was seen. VISITNUM's ordering by first-observed date is labelled an ASSUMPTION inside the rule text itself, because the trial design (TV) is absent from the corpus. Finally `pack_id` fingerprints content while excluding timestamps and provenance — "the id answers 'is this the same deliverable?', and git_head separately answers 'what made it?'."

THE PACKAGES. Stdlib for the build (`csv`, `json`, `re`, `datetime`); openpyxl only in the two workbook writers, imported inside the functions so the pack itself never needs it. No pandas anywhere in the module — blank-preservation is the stated reason for hand-rolled CSV reading (pandas' default NA coercion is exactly the behaviour being avoided; the repo's `CLAUDE.md` records an R-side bug of the same species).

BRIDGE. A TA pack is the review binder you'd assemble for a mapping-review meeting — raw listings, draft specs, an issues list. It is not an eSub package and not a P21 report: nothing in it was executed, and the coverage report answers "did every collected column get a disposition?", which is closer to the annotated-CRF completeness question than to conformance. Where the analogy misleads: in your hand-built binder, an unmapped column with an obvious disposition would just get one written in; here, no committed hint rule means no suggestion, on principle.

TRY IT YOURSELF. The coverage invariant in miniature — one row per raw column, whole-token mention matching for derived inputs:

```
import re

def mentions(column, text):
    pat = r"(?<![A-Za-z0-9_])" + re.escape(column) + r"(?![A-Za-z0-9_])"
    return re.search(pat, text, re.IGNORECASE) is not None

header = ["subject_id", "event_term", "start_date", "date"]
mappings = [
    {"sdtm_var": "AETERM", "raw_var": "event_term", "rule": "Copy verbatim."},
    {"sdtm_var": "AESTDTC", "raw_var": "",
     "rule": "Derive ISO 8601 from start_date."},
]
consumers = {}
for m in mappings:
    if m["raw_var"]:
        consumers.setdefault(m["raw_var"], []).append(m["sdtm_var"])
    else:
        for col in header:
            if mentions(col, m["rule"]):
                consumers.setdefault(col, []).append(m["sdtm_var"])
coverage = [{"column": c, "status": "MAPPED" if c in consumers else "UNMAPPED",
             "consumed_by": consumers.get(c, [])} for c in header]
for row in coverage:
    print(row)
assert len(coverage) == len(header)          # exactly one row per column
assert coverage[3]["status"] == "UNMAPPED"   # 'date' not matched inside 'start_date'
```

LIMITS. Stated in the code and true: `derived_from` reports that a rule mentions a column, not that the generated program reads it — upgrading means parsing the program (labelled Phase 3). The negation heuristic surfaces ambiguity but resolves nothing, by design. `visit_map`'s ordering is an assumption wherever TV is absent. The Excel views carry the pack's disclaimer and dirty-tree warning on the INDEX sheet, but a reader who only opens the AE sheet never sees either.

HOW TO IMPROVE IT. Repeat a one-line disclaimer on every sheet, not just INDEX. Phase 3 as named: derive `derived_from` from the program's AST/parse rather than prose. Consider recording the spec adjudication ids per row so a reviewer can trace a mapping to the fleet run that proposed it.

LIFT AND REUSE. Three lift-worthy pieces: the provenance bracket (snapshot before inputs, snapshot after, drift note), the content-fingerprint-vs-provenance split for artifact identity, and `coverage_for`'s exactly-once invariant with its two pinned tests. Generic interface: `coverage(header: list[str], consumers: dict[str, list[str]]) -> list[dict]` — everything else is corpus-specific glue.

M7.7 `services/bundle.py` — pull-only review zip plus run report

The reviewer sits in RStudio Server on VPS1. The mapper runs on VPS2. VPS2 cannot SSH to VPS1, so nothing can ever be pushed to the reviewer — but VPS1 can reach VPS2 over HTTPS. Delivery is therefore one line the reviewer runs themselves: `download.file("https://clincoder.cloud/sdtm-review/api/bundle/<job_id>", "b.zip"); unzip("b.zip")`. Every design decision in this module follows from that single fact.

WHAT IT DOES. `build_bundle(job_dir, meta, out_zip)` packages one mapper job into a self-sufficient zip: the three generated programs (`programs/map.sas`, `map.R`, `map.py`), the exact input the mapper saw (`input/`), the reference SDTM when one exists (`expected/`), `define.xml`, plus two files it writes itself —

`run.R` (the reviewer's only entry point, embedded in the module as the `_RUN_R` string constant) and `README.md` (generated per job by `_readme`). It returns `{"ok", "path", "members", "bytes", "error"}` and never raises for a missing part — a missing program is omitted from the zip and named in the README, because "a job that failed to emit an artifact is a finding, so score it rather than discarding it."

WHY IT EXISTS. Without it, review means giving reviewers server access, absolute paths, and a verbal explanation of which file is which. With it, the AE bundle unzips to `AE_<dataset>_<job>/` (each token sanitized by `_safe`, which prevents a dataset id containing a slash from writing outside the bundle root) and `source("run.R")` does everything: stages `input/` and `map.R` into a `work/` directory, runs the program there, writes `out.csv`, and — when an oracle exists — prints a per-column PASS/FAIL diff.

INPUTS AND OUTPUTS. Input: a job directory and `meta = {"job_id","domain","dataset_id","ta","tier","has_oracle","study_id"}`. The finder functions are tolerant by design: `_find_program` accepts `<DOMAIN>_map.R`, `<domain>_map.R`, `map.R`, or any `*_map.R` ("the bundler is not coupled to one naming era"); `_find_input` prefers an explicit `input/` dir, then `*_raw.*` files, then any data file that is not the mapper's own output; `_find_define` accepts three spellings of `define.xml`. Output: the zip, whose README carries a job metadata table, the fetch/run instructions, the contents list, the "Not included in this bundle" section, and the five-dimension scoring rubric (spec, code, CT, derivations, define.xml — each 1-5, comment required when any score is ≤ 2 or the verdict is `reject`).

HOW IT WORKS. `build_bundle` gathers `staged` pairs (arcname, source path), builds the README before opening the zip (so the members list in the README matches what will be written), then writes everything with `zipfile.ZipFile(..., ZIP_DEFLATED)` — real files via `z.write`, `run.R` and `README.md` via `z.writestr`. The embedded `run.R` deserves study even though it is R, because its contract is enforced from Python: `tests/test_bundle.py` literally asserts it contains no absolute path. It locates itself whether sourced in RStudio or run via `Rscript` (parsing `--file=` from `commandArgs`, walking `sys.frames()` for `ofile`, falling back to `rstudioapi`), guards missing packages with a readable message instead of a raw R error, aliases the shipped input to whatever filename `map.R` hardcodes ("cheaper than making the reviewer edit generated code"), and when no oracle exists it says so in capitals: "NO ORACLE EXISTS FOR THIS DATASET. … YOUR OWN JUDGEMENT IS THE ONLY SIGNAL." The README's bluntest section is about SAS: `map.sas` "is a **deliverable, not a verified artifact**" — there is no SAS runtime on either VPS, the program has never executed anywhere, and the reviewer is told to judge it by reading. Callers: `routes_review.py` (`/api/bundle/<jid>`) and `scripts/review_gate.py`.

THE RUN REPORT: `SERVICES/RUN_REPORT.PY`. The bundle is what a reviewer receives; the run report is what the operator receives about the run that made it. `report_run(kind, ta, extras, start_offset, or_before_balance, wait=False)` is the single choke point (`scripts/batch_run.py` line 1033 calls it at the end of a batch) that builds two renderings of the same usage slice so they cannot drift apart: an **unmasked** record — real model ids, backend route labels from `ROUTE_LABEL`, token counts, and an honest cost line — appended synchronously to `evidence/run_log.jsonl` and dispatched to Telegram (via `/root/deploy_kit/tg_digest.py`, which verifies delivery) and Notion (creds from `/etc/clincoder/notion.env`, with a three-way diagnosis: absent, unreadable, or missing keys); and a **masked** record — tier aliases only, via `services.model_tiers.alias_for` — safe to hand back to an API caller, because clients never see model or vendor names. Two honesty mechanisms are worth copying: `_cost_line` never invents a dollar figure — a real per-call cost is only reported for OpenRouter, and only as a measured balance delta (`openrouter_balance()` snapshotted before the run by the caller, diffed after); and the sinks are best-effort in a daemon thread (`_dispatch_sinks`) so "logging a run must never be why the run itself fails" — with `wait=True` available because a process that exits immediately would

otherwise kill the in-flight thread. Note what the run report is not: it is plain text with `tg_digest.py`'s tiny markup plus a JSONL line — no docx. The Word-generation half of this module pairing lives in M7.2's `docgen.py`.

THE PACKAGES. `zipfile` (stdlib) for the bundle — streaming, deflate, `writestr` for generated members; no reason for anything heavier. `run_report` uses stdlib `urllib.request` for the OpenRouter balance call and `subprocess` for Telegram, importing `requests` only inside the Notion sender. The R script travels as a raw string constant rather than a template file — one file, one review, zero packaging steps to forget.

BRIDGE. The bundle is the double-programming QC transfer package you have zipped by hand a hundred times — programs, inputs, expected outputs, a README saying what to check. Two places the analogy misleads: first, the package scores its own gaps (a missing artifact is a rubric item, not a reason to rebuild the package); second, PASS output from `run.R` is a column-level diff against a synthetic oracle, not an independent double-programming result — the README says so, and the no-oracle case refuses to print any PASS at all. The run report has no clean SAS-world analogue; the closest is a batch log, except this one exists to make spend and model routing auditable, which your `.log` file never did.

TRY IT YOURSELF. A self-sufficient review bundle in miniature (stdlib only):

```
import zipfile, os

root = "AE_demo_job42"
readme = """# Review bundle — AE / demo
Run: python3 check.py   (relative paths only; run from the unzipped dir)
## Not included
- programs/map.sas — not generated for this job. Score that as a finding.
"""
check = """import csv
rows = list(csv.DictReader(open("input/ae_raw.csv")))
print(len(rows), "input rows; columns:", list(rows[0]))
"""
with zipfile.ZipFile("mini_bundle.zip", "w", zipfile.ZIP_DEFLATED) as z:
    z.writestr(f"{root}/input/ae_raw.csv",
               "subject_id,event_term\n001,HEADACHE\n002,NAUSEA\n")
    z.writestr(f"{root}/check.py", check)
    z.writestr(f"{root}/README.md", readme)
with zipfile.ZipFile("mini_bundle.zip") as z:
    print(z.namelist())
    assert all(not n.startswith("/") for n in z.namelist())  # no absolute paths
print("bytes:", os.path.getsize("mini_bundle.zip"))
```

LIMITS. The bundle trusts the job directory's naming conventions; a job that wrote its program under a fourth naming era silently loses that arm (it is listed as missing, which is honest, but wrong-missing). `run.R`'s output collection falls back to "newest CSV that is not an input" when the program overwrote an existing name — a heuristic, flagged as such in its comments. In `run_report`, the Telegram and Notion failure prints go to stdout, i.e. the journal, not to any alerting path — a dead sink is discoverable only by reading logs; and `openrouter_balance` returns `None` on any error, indistinguishable from "not configured."

HOW TO IMPROVE IT. Bundle: write a `MANIFEST.json` with sha256 per member so a reviewer can prove what they reviewed. Run report: a `sinks` field in the returned dict recording delivered/failed per sink, so evidence shows whether the human actually got told.

LIFT AND REUSE. `build_bundle`'s shape is the generic artifact: `(job_dir, meta, out_zip) -> report`, tolerant finders, gaps named in a generated README, an embedded self-locating entry script. For the report, the masked/unmasked split from one usage slice is the reusable idea for any tool where operators and clients see different levels of detail.

What this module still owes you

Honest gaps, so you do not go looking for things that are not there. **THE TWO DEFINE ARMS HAVE NOT BEEN UNIFIED**, and the book documents but does not resolve the consequence: `define_recon` can only parse the legacy `odm/v2.1`-namespaced files, and no automated check today reconciles a produced dataset against the CDISC-generated define.xml. **No Pinnacle 21 run exists** for any define.xml named here — "XSD-valid" is the strongest claim made anywhere in this codebase, and the strongest this book makes. **`study_spec_export.py` is documented as broken, not fixed.** The bundle's `run.R` was read, not executed on VPS1 by this author — its behaviour claims come from the script text and its tests, not a fresh RStudio run. Whether `LF.acrf` in the generated define.xml resolves to a `def:leaf` element was not checked. And all data named here is synthetic (Tier B Cardiology corpus); no claim in this module has been tested against real study data.

Module 8 — Gates and harnesses

This module is about the machinery that decides whether the SDTM Mapper's output is correct, as opposed to merely present. The governing rule was earned the hard way: this project produced false-green evidence more than once — validation tables whose "Result" column read PASS without anything having run — and the design that came out of it is stated in the code itself: a model may write a test; only a RUN may write an evidence file. Every chapter below is a variation on one thesis: **a gate that cannot fail is not a gate**, and each gate here carries either a negative control that makes it fail or an explicit statement of what it does not prove.

A note for the SAS/R clinical programmer: this is your world. The tool literally has `iq_gate.py` and `oq_gate.py` — Installation Qualification and Operational Qualification, exactly as in a computer system validation protocol — and its parity gates are double programming: an independent second implementation, compared cell-for-cell against production output. The difference from a human QC programmer is that here both arms are machine-generated, so the harness must also defend against both arms being wrong in the same way, and against the comparison itself being vacuous.

M8.1 `/root/sdtm_mapper/scripts/ci_gate.py` — grades golden outputs; famously graded the answer key

A CI job runs on every change, prints `CI GATE: PASS (5/5 golden domains conformant >= 95%)`, and everyone relaxes. Then someone deletes the entire mapping engine and the gate still passes — because it never touched the engine. That is not a hypothetical; it is the documented reason `parity_gate.py` (next chapter) exists, stated in its own docstring: "Static conformant files always conformant... It tests answer key, not student."

WHAT IT DOES. `ci_gate.py` is 36 lines. It walks `/root/sdtm_mapper/golden/` — five domain folders (`AE DM DS EX VS`), each holding a raw input CSV, a hand-verified expected output (`ae.csv` etc.), and committed mapper programs — and runs each expected output through `services.validator.validate(mapped, dom)`, the deterministic CDISC SDTMIG conformance checker. A domain passes if the verdict is `Pass` or the conformance percentage clears a threshold (`CI_CONF_THRESHOLD` env var, default 95). Exit 0 means green.

WHY IT EXISTS. It is cheap, offline, LLM-free, and catches one real class of defect: a change to the validator or the golden fixtures that breaks conformance. What it can never catch is a broken mapper, because the files it grades are static. Understanding that scope error — and how it produced a false green — is the whole point of this module. The SAS analogy: it is as if your QC process consisted of re-running PROC COMPARE between the production dataset and a saved copy of the production dataset. Byte-identical forever, informative never.

INPUTS AND OUTPUTS. Input: `golden/<DOM>/<dom>.csv` files (e.g. `golden/AE/ae.csv`, a mapped AE domain from the CDISC pilot study, with columns like `USUBJID, AETERM, AEDECOD, AESEV`). Output: console lines and an exit code — notably, no evidence file. That asymmetry is deliberate in the current design: `oq_gate.py` wraps it and writes the evidence with provenance.

HOW IT WORKS. The whole control flow:

```
for dom in sorted(os.listdir(GOLDEN)):
    dp = os.path.join(GOLDEN, dom)
    if not os.path.isdir(dp):
        continue
    mapped = os.path.join(dp, dom.lower() + ".csv")
    ...
    rep = validator.validate(mapped, dom)
    pct = rep.get("conformance_pct", 0.0); verdict = rep.get("verdict", "?")
    ok = (verdict == "Pass") or (pct >= THRESH)
```

If no golden domain is found at all it exits 2 (`CI FAIL: no golden domains found`) — an early instance of the house pattern that an empty check must not read as a pass.

THE PACKAGES. Stdlib only (`os`, `sys`, `glob`) plus the project's own `services.validator`. No pandas here — the validator does the parsing. Nothing to swap.

TRY IT YOURSELF. The failure class in miniature — a "gate" that validates a static file keeps passing after you break the generator:

```
import csv, io

def generator(broken=False):
    if broken:
        raise RuntimeError("engine deleted")
    return "USUBJID,AETERM\nS-1,HEADACHE\n"

GOLDEN = "USUBJID,AETERM\nS-1,HEADACHE\n"   # committed long ago

def static_gate():          # ci_gate-shaped: grades the committed file
    rows = list(csv.reader(io.StringIO(GOLDEN)))
    return rows[0] == ["USUBJID", "AETERM"]

def executing_gate():       # parity_gate-shaped: runs the generator
    return generator(broken=True) == GOLDEN

print("static gate with engine deleted :", static_gate())      # True (!)
try:
    print("executing gate:", executing_gate())
except RuntimeError as e:
    print("executing gate: FAIL --", e)                        # the truth
```

LIMITS. It proves the golden outputs conform to the IG rules the validator encodes — nothing more. It does not execute any mapper, does not touch the LLM, and its threshold (95%) is a policy number, not a measured one. The project's own layer taxonomy (in `parity_gate.py`'s docstring) demotes it to "Layer" status precisely because it is necessary but radically insufficient.

HOW TO IMPROVE IT. Write an evidence JSON with `executed: true`, timestamp and git sha, like every other gate here — today its result exists only as stdout captured into `evidence/step4_ci_gate.txt`. And add a negative-control fixture: one deliberately non-conformant CSV in a `_mutant/` folder that the gate must FAIL on, run in CI, so a validator that starts approving everything is caught.

LIFT AND REUSE. The reusable idea is the taxonomy, not the script: label every check with what it executes. "Validates stored artifacts" (this), "executes committed code" (M8.2), "executes the live pipeline" (M8.3). Any project that lists its gates in those three buckets discovers its own false-green risk in an afternoon.

M8.2 `/root/sdtm_mapper/scripts/parity_gate.py` — executes shipped mappers, compares cells to golden

Your study has a production AE program and a golden AE dataset that a human verified. The only question that matters is: if I run the program the customer will run, on the raw data, do I get the golden dataset back — every cell? `parity_gate.py` is the script that finally asked that question, after `ci_gate.py` had spent weeks answering an easier one.

WHAT IT DOES. For every domain under `golden/` and for each executable language arm, it copies the raw CSVs and the committed mapper program (`AE_map.py` or `AE_map.R`) into a fresh temp directory, executes it as a subprocess with a 180-second timeout, and compares the produced `ae.csv` to the golden `ae.csv` cell by cell. It then writes `validation_package/PQ_parity_evidence.json` from the actual run — a file that "can and will say fail", as its docstring insists — and exits 0 only if every executed run reproduced golden exactly.

WHY IT EXISTS. The docstring says it plainly: `ci_gate.py` "passes with the engine deleted. It tests the answer key, not the student." So parity in this file means **output parity between an executed program and a human-verified golden dataset** — the software equivalent of double programming where one arm is the frozen, adjudicated truth. It is labelled "Layer 3: OUTPUT-PARITY" and "PQ" (Performance Qualification) in the evidence, completing the IQ/OQ/PQ triad.

INPUTS AND OUTPUTS. Input: `golden/AE/ae_raw.csv` (CDISC pilot AE data — columns like `STUDY, PATNUM, IT.AETERM, AEOUTCOME, IT.AESEV`), `golden/AE/AE_map.py`, `golden/AE/ae.csv`. Output: `validation_package/PQ_parity_evidence.json` containing per-run records like `{"domain": "AE", "lang": "python", "result": "pass", "secs": 1.2, "produced_sha256": ..., "golden_sha256": ...}` plus `langs_skipped`, `git_sha`, and the `sas_arm` block. The evidence on disk right now records `pass: true` across 10 runs (5 domains × python + r), timestamped 2026-07-28.

HOW IT WORKS. Three functions carry it. `LANGS` declares the executable arms — `python` and `r` only. `run_domain(dom, lang)` does isolation and execution; note the failure ladder, each rung named:

```
def run_domain(dom, lang="python"):     # abridged from the original
    ...
    proc = subprocess.run(cfg["cmd"](), cwd=tmp,
                          capture_output=True, text=True, timeout=180)
    if proc.returncode != 0:
        return {"domain": dom, "lang": lang, "result": "fail",
                "reason": "mapper crashed (rc=%d)" % proc.returncode}
    produced = os.path.join(tmp, "%s.csv" % dom.lower())
    if not os.path.exists(produced):
        return {"domain": dom, "lang": lang, "result": "fail",
                "reason": "expected output not produced"}
    ok, diffs = compare(produced, expect)
```

`compare(produced, expected)` checks header equality (distinguishing missing columns, extra columns, and mere order differences), then row count, then every cell through `norm()` — which only strips whitespace: "We do NOT normalise values themselves - a wrong value must stay wrong." Diffs are capped at 10 examples plus a total count. In `main()`, an arm whose runtime is absent is skipped loudly: "`[r] SKIPPED - Rscript not on PATH. This arm proves nothing today.`" — and the skip is recorded in the evidence, because "a silently skipped arm is how PASS starts to lie."

THE SAS POLICY — A CLAIM WITHDRAWN IN WRITING. There is no SAS runtime on this host, so `AE_map.sas` cannot be executed. An earlier version of this file implied three-language parity; the docstring now states that claim "was false, because the SAS arm was never run. It has been removed rather than

softened." What replaced it is a delivery rule, not a verification claim: SAS is emitted from the same mapping spec as R, and per domain the evidence records `sas_deliverable[dom]` as `"release"` only if that domain's R arm passed — `"hold_r_arm_not_passed"` otherwise. If Rscript is missing entirely, `r_pass` is empty and every SAS artifact holds. For a validation-minded reader this is the exact discipline of a protocol deviation: the unexecutable check is documented as unexecuted, not quietly reworded into a pass.

THE PACKAGES. Stdlib throughout — `csv`, `subprocess`, `tempfile`, `hashlib`, `shutil`. No pandas: the comparison is exact strings on purpose, and `csv.reader` cannot silently coerce `"71.0"` to a float the way `pandas.read_csv` would. Swapping to pandas here would add a normalisation layer the gate explicitly refuses to have.

TRY IT YOURSELF. A minimal executed-parity check that fails loudly and writes an evidence file:

```
import csv, io, json, datetime

golden = [["USUBJID", "AESEV"], ["S-1", "MILD"], ["S-2", "SEVERE"]]

def mapper():  # pretend this is the shipped program being executed
    return [["USUBJID", "AESEV"], ["S-1", "MILD"], ["S-2", "MODERATE"]]

produced = mapper()
diffs = []
if produced[0] != golden[0]:
    diffs.append("header differs")
else:
    for i, (p, g) in enumerate(zip(produced[1:], golden[1:]), 1):
        for j, col in enumerate(golden[0]):
            if p[j].strip() != g[j].strip():
                diffs.append(f"row {i} col {col}: produced {p[j]!r}, expected {g[j]!r}")

evidence = {"phase": "parity-demo", "executed": True, "pass": not diffs,
            "timestamp": datetime.datetime.now(datetime.timezone.utc).isoformat(),
            "diffs": diffs}
with open("parity_demo_evidence.json", "w") as fh:
    json.dump(evidence, fh, indent=2)
print(json.dumps(evidence, indent=2))
assert not diffs, f"PARITY FAIL: {diffs}"
```

LIMITS. Honest and self-declared: it covers only the deterministic path — the committed generated programs. `engine.classify_domain` and `generate_mapping_spec` are LLM calls and are explicitly out of scope ("non-reproducible, cost money, cannot be audit evidence"). That scope line is where the second false-green hid, which is the next chapter's story. It also proves nothing about domains outside the five golden fixtures, and its goldens are one study's shape.

HOW TO IMPROVE IT. The golden corpus is the bottleneck: five domains, one study. The sdtm_review fork's `data/synthetic_v4` corpus (11 domains, seeded, with an oracle) is the natural expansion. And the `compare` function is duplicated conceptually in three places across the two repos (here, `codegen_parity` imports this one — correctly; `batch_run._compare_to_oracle` is a separate, key-aligned implementation) — unifying on the key-aligned version would remove a real divergence risk.

LIFT AND REUSE. Genericise `run_domain` as `run_in_isolation(program, inputs, expected_output, cmd) -> result_dict`, and keep two iron rules: the comparison normalises transport (whitespace) but never values, and every skip is recorded in the evidence. That pair of rules is portable to any double-programming pipeline, SAS included.

M8.3 `/root/sdtm_mapper/scripts/codegen_parity.py` — drives the live LLM pipeline, then compares

On 2026-07-17, per this file's own docstring, a council of advisors recommended: swap the code-generation model to a cheaper one, run `parity_gate`, and "if it stays 4/4 you cut inference cost with proof it's safe." The swap was made; parity_gate returned byte-identical results; nothing was learned — because parity_gate runs static committed files that never touch a model. The gate that was supposed to adjudicate the model swap could not see the model. This is the anatomy of the second false-green: not fabricated data, but a true green read as evidence for a claim outside its scope.

WHAT IT DOES. `codegen_parity.py` ("Layer 4: LIVE-PIPELINE parity") drives the real product path for each golden domain: `profile_dataset` (local) → `generate_mapping_spec` (LLM call) → `generate_python` (LLM call) → execute the model's own code in a temp dir → `compare` to golden, cell by cell. It writes `validation_package/codegen_parity_<label>.json` per model label, so different models' runs sit side by side as separate evidence files.

WHY IT EXISTS. It is the only gate that can answer "is this model safe to map with?" Without it, every model swap is judged by gates that pass regardless — the `ci_gate.py` bug "wearing a new hat", as the docstring puts it.

INPUTS AND OUTPUTS. Input: the same `golden/AE/ae_raw.csv` fixtures, plus a `--label` (defaults to the `SDTM_CC_CODE` env var) and optional `--domains AE,DM`. Output: an evidence JSON whose per-domain record carries what the model actually did. The directory holds three real ones today: `codegen_parity_deepseek-v4-pro.json`, `codegen_parity_mimo.json`, and — most instructive — `codegen_parity_MUTANT.json`, a run in which the AE arm failed, recording `spec_model: "commandcode:xiaomi/mimo-v2.5-pro"`, coverage 21/24 targets mapped, and diffs such as:

```
row 1 col AESEV: produced '',                 expected 'MILD'
row 1 col AEREL: produced 'Probably Related', expected 'PROBABLY RELATED'
```

A gate whose evidence directory contains a recorded FAIL is a gate you can trust to mean something when it says PASS. ("It is expected to FAIL sometimes. A gate that has never failed is not known to work" — that sentence is from `iq_gate.py`, but this file is where the project proved it to itself.)

HOW IT WORKS. `run_domain(dom)` is a straight pipeline with named failure stages — `"model produced empty code"`, `"generated code crashed (rc=%d)"`, `"no ae.csv produced"` — and then:

```
sys.path.insert(0, os.path.join(ROOT, "scripts"))
from parity_gate import compare  # noqa: E402
...
ok, diffs = compare(produced, expect)
rec.update(result="pass" if ok else "fail", diffs=diffs[:8],
           secs=round(time.time() - t0, 1))
```

Importing `compare` from `parity_gate` rather than rewriting it is a design decision the comment defends: "two comparison functions eventually disagree, and then neither is evidence."

THE REPORTING RULE. Because LLM output varies run to run "even at temperature 0" (a fact this project measured elsewhere and encodes here as policy), the evidence hard-codes an anti-inflation clause into itself: `"reporting_rule": "n=4. Report as '4 of 4'. A bare percentage from n=4 is not a measurement."` For the statistician in the audience: a single run is an observation, not a rate; the file refuses to let anyone quote it as one.

THE PACKAGES. Project services (`engine`, `codegen`, `privacy`) plus stdlib. `privacy_mode()` is printed and recorded so evidence shows what left the box.

TRY IT YOURSELF. The two-layer distinction, runnable — a static gate stays green across a "model swap"; a live gate notices:

```
import random

GOLDEN = "USUBJID,AESEV\nS-1,MILD"
COMMITTED_PROGRAM_OUTPUT = GOLDEN          # frozen on disk months ago

def model(name):                            # the thing being swapped
    if name == "cheap":
        return "USUBJID,AESEV\nS-1,Mild"    # case defect: not a golden match
    return GOLDEN

def static_parity(_model_name):             # parity_gate-shaped
    return COMMITTED_PROGRAM_OUTPUT == GOLDEN

def live_parity(model_name):                # codegen_parity-shaped
    return model(model_name) == GOLDEN

for m in ("good", "cheap"):
    print(f"model={m:5s} static={static_parity(m)} live={live_parity(m)}")
# static is True for both -- it cannot see the model. live catches 'cheap'.
```

LIMITS. Deliberately does not cover `classify_domain` (the domain is supplied — this measures mapping, not detection). It costs real money per run (two LLM calls per domain), so it cannot be a per-commit gate; it is an on-demand adjudicator. And a single green run of a stochastic pipeline licenses far less than a green run of a deterministic one — the file says so about itself, which is rarer than it should be. Do not run it casually: each invocation spends LLM calls (`#| skip-check` applies to running the real script, not to reading it).

HOW TO IMPROVE IT. Repeated-run support: `--n 5` with per-domain pass counts and a Wilson interval would turn observations into an estimable rate (the house rule elsewhere is "Wilson CI, never bare percentages"). Snapshot-replay of LLM responses, which the parity_gate docstring already names as the missing piece for spec/classify coverage, would make a free deterministic regression layer out of past paid runs.

LIFT AND REUSE. The interface worth stealing: `evidence[label] = run(pipeline, fixtures)` with the label as part of the filename, so competing models accumulate comparable artifacts instead of overwriting one "latest". Plus the reporting_rule field — evidence that carries its own quotation rules travels safely through summaries written by people (or models) who never read the code.

M8.4 `/root/sdtm_mapper/scripts/iq_gate.py` — executes IQ checks the document only asserted

Open `docs/validation_package/04_IQ.md` in this repo and you find an Installation Qualification table whose "Result" column reads PASS on every row. Nothing ran to produce it. The docstring names the incident this rhymes with: on 2026-07-17 "the worker fleet wrote IQ/OQ/PQ files with every check hardcoded `pass` for an app that had never been run end to end." If you have ever audited a CSV package where the executed-by initials were filled in before the test was executed, you know this failure class personally.

WHAT IT DOES. `iq_gate.py` re-implements the six IQ checks as functions that interrogate the live system — `iq1` OS/runtime, `iq2` systemd unit active, `iq3` ExecStart points at the deployed `app.py`, `iq4` port

8812 listening and `/api/health` returning 200, `iq5` `flask`/`pandas`/`openpyxl` importable with versions, `iq6` LICENSE file present — and writes `validation_package/IQ_evidence.json` from what it observed, with `executed: true`, UTC timestamp, git sha, hostname, and per-check `expected` vs `actual`.

WHY IT EXISTS. House invariant, quoted verbatim from the file: "a model may write a test; only a RUN may write an evidence file." The provenance fields exist so a downstream rubric "can tell it apart from a typed file". And the next line is the module thesis in its original wording: "It is expected to FAIL sometimes. A gate that has never failed is not known to work."

INPUTS AND OUTPUTS. Input: the live host (systemctl, ss, curl, the Python import machinery) — no files. Output example, abbreviated from the real `IQ_evidence.json` (on disk now: `"summary": "6/6 checks passed"`):

```
{"phase": "IQ", "document": "SM-IQ-001", "executed": true,
 "git_sha": "...", "host": "...",
 "checks": [{"id": "IQ-4", "expected": "127.0.0.1:8812 listening, /api/health 200",
             "actual": "listening=True /api/health=HTTP 200", "pass": true}, ...]}
```

HOW IT WORKS. A tiny harness pattern worth memorising — every check is `(id, description, expected, fn)` where `fn` returns `(pass, actual)`, and the wrapper converts exceptions into failures with the exception text as the actual:

```
def check(cid, desc, expected, fn):
    try:
        ok, actual = fn()
    except Exception as e:
        ok, actual = False, "check raised: %s" % e
    return {"id": cid, "description": desc, "expected": expected,
            "actual": actual, "pass": bool(ok)}
```

One check deserves special note. `iq3` found that the IQ document was stale — it specifies `/root/sdtm_mapper/app.py`, but production moved to `/srv/prod/sdtm_mapper` on 2026-07-19. The gate accepts either **and records which**, "so the spec can be corrected rather than the gate quietly bent to match." That is the correct handling of a spec/system divergence in any validation regime: surface it, never absorb it.

THE SIBLING: `OQ_GATE.PY`. Same shape, next protocol phase. `05_OQ.md` asserted "CI GATE PASS (4/4)" as prose; `oq_gate.py` executes both halves instead. `run_ci_gate()` runs `scripts/ci_gate.py` as a subprocess (this is where ci_gate's missing evidence file gets written on its behalf, as `OQ_evidence.json`'s OQ-CI check plus captured per-domain lines). `run_api_checks()` exercises the API-key gate via Flask's `test_client` — no port bind — with five checks: no key → 401, invalid key → 401, valid key admitted, `/api/health` open, `/api/demo/*` open. Crucially it isolates state before importing the app:

```
_TMP = tempfile.mkdtemp(prefix="oq_gate_")
os.environ["API_KEY_ENFORCE"] = "1"
os.environ["SDTM_KEYS_FILE"] = os.path.join(_TMP, "api_keys.json")
os.environ["SDTM_QUOTA_FILE"] = os.path.join(_TMP, "api_quota.json")
sys.path.insert(0, APP_DIR)
```

so "a gate run never mutates production api_keys.json or burns real quota" — the env vars must be set pre-import because `apikey.py` reads them at module load. The current `OQ_evidence.json` records 6/6.

THE PACKAGES. Stdlib plus Flask's test client (already a dependency). One fork divergence found while reading: the mapper copy runs commands via `shlex.split(cmd)` with `shell=False`; the sdtm_review

copy of `iq_gate.py` still uses `shell=True`. Same checks, different injection posture — the mapper copy is the newer, safer one.

TRY IT YOURSELF. A two-check IQ gate for any machine, evidence file included:

```
import importlib.util, json, os, platform, datetime

def check(cid, desc, expected, fn):
    try:
        ok, actual = fn()
    except Exception as e:
        ok, actual = False, f"check raised: {e}"
    return {"id": cid, "description": desc, "expected": expected,
            "actual": actual, "pass": bool(ok)}

checks = [
    check("IQ-1", "OS", "any Linux/mac/Windows",
          lambda: (True, platform.platform())),
    check("IQ-2", "deps", "json importable (stand-in for flask/pandas)",
          lambda: (importlib.util.find_spec("json") is not None, "found")),
]
doc = {"phase": "IQ-demo", "executed": True,
       "timestamp": datetime.datetime.now(datetime.timezone.utc).isoformat(),
       "pass": all(c["pass"] for c in checks), "checks": checks}
with open("iq_demo_evidence.json", "w") as fh:
    json.dump(doc, fh, indent=2)
print(doc["pass"], "->", os.path.abspath("iq_demo_evidence.json"))
```

LIMITS. IQ proves installation, not behaviour: all six checks can pass on a service that maps every domain wrongly. OQ's API half proves the gatekeeping works, not that what is behind the gate is correct. Both are snapshots — `IQ_evidence.json` is only as current as its timestamp, and nothing on this box re-runs it on a schedule. And an evidence file with `executed: true` is still just a file; the provenance raises the cost of fabrication, it does not make fabrication impossible.

HOW TO IMPROVE IT. Wire both into a timer (the sdtm_review fork's `conformance_watch.py`, M8.7, is the existing pattern for scheduled evidence with change-only notification). Add a negative control to OQ-API: flip `API_KEY_ENFORCE` off and assert the gate then fails, proving the 401s come from the key gate rather than from a broken route.

LIFT AND REUSE. The `(id, expected, fn) -> {expected, actual, pass}` table plus the pre-import env isolation trick are drop-in for any Flask/systemd app. For a clinical team: this is a machine-executable IQ/OQ protocol — keep the .md protocol as the human-approved specification and generate its results section from the JSON, never by hand.

M8.5 `/root/sdtm_mapper/scripts/visitor_soak.py` — hammers the running site as visitors would

On 2026-07-19, per this file's docstring, the site was shipping a live HTTP 500 on `/api/profile` — the primary visitor action, drag-and-drop a CSV — with all four correctness gates green, because no gate had ever sent the running website a single HTTP request. Mapping correctness and site availability are different claims, and each needs its own gate.

WHAT IT DOES. `visitor_soak.py` ("Layer 6") replays seeded, randomised visitor sessions against a running instance — landing page, demo walkthroughs in order and shuffled, deliberately malformed POSTs, out-of-order step requests, unknown routes — sequentially and then concurrently, and asserts four invariants: no 5xx ever; no response body carrying a `trace` key (a traceback leaking to a visitor);

every JSON response has a status in `OK_STATUS = {200, 400, 404, 413}`; and the landing page actually contains HTML, "200 with an empty body is the failure a status check misses." Evidence goes to `validation_package/PQ_visitor_soak_evidence.json`; the one on disk records `pass: true`, 100 iterations, 0 failures, against `http://127.0.0.1:18812`.

WHY "100 ITERATIONS" IS NOT THE POINT. The docstring contains the best paragraph on soak design in either repo: 100 identical sequential GETs against one warm process "is ONE test executed 100 times" — the requests hit a memoized `_DEMO_CACHE` dict in a single process and are near-perfectly correlated, so "repetition buys ~1 bit of information." The iterations are therefore spent on three axes that genuinely vary the system: variation (seeded scenario draws), concurrency (a thread pool against a threaded Werkzeug server with an unsynchronised module-global `JOBS` dict), and cold start. Any SAS programmer who has "validated" a macro by running it twice on the same input will recognise the fallacy being dismantled.

INPUTS AND OUTPUTS. Input: `--base URL --iterations 100 --concurrency 10 --seed 20260719`, plus `--read-only` (GET-only, safe against production) and `--live-llm` (exactly ONE paid request to prove the model seam is alive — "the hundredth adds nothing"). Output: the evidence JSON, which carries an explicit `claim_supported` ("the site was AVAILABLE…") and a three-item `claim_NOT_supported` list — not correctness ("parity_gate.py owns that"), not an uptime percentage ("these sessions are correlated, not independent Bernoulli trials"), not LLM quality.

HOW IT WORKS. `scenario(base, rng, read_only)` draws one of six session kinds and returns a failure list; `check()` applies the invariants to each response. `main()` splits iterations 70/30 sequential/concurrent (`n_conc = max(1, int(args.iterations * 0.3))`) and runs the concurrent share through `ThreadPoolExecutor`. One guard is pure operational scar tissue:

```
if not args.read_only and ":8812" in args.base:
    print("  REFUSING: %s looks like the LIVE service and --read-only was not set." % args.base)
    print("  Soaking prod with POSTs writes job dirs and audit rows a real visitor would see.")
    sys.exit(1)
```

A test harness that can pollute production is itself a defect; this one refuses.

THE COPY IN SDTM_REVIEW. `/root/sdtm_review/scripts/visitor_soak.py` is byte-identical (verified with `diff` this session) — a straight fork copy, still pointing its defaults at the mapper's port and evidence path. Unlike `iq_gate.py`, which diverged between forks, nobody has adapted this one; if you soak the review app (port 8813) you must override `--base` and `--evidence` yourself.

THE TINY SIBLING: `/ROOT/SDTM_REVIEW/SCRIPTS/E2E_SMOKE.PY`. Where the soak asks "does the site stay up under varied traffic", the smoke asks "does one real end-to-end path still work" — 87 lines, six checks against `http://127.0.0.1:8813`: absolute-path and traversal requests to `/api/profile` are refused with 400 (S1/S2); profiling catalogue dataset `tierA:CDISCPILOT01:DM` opens a review run (E1); a valid human review persists (E2); and then the check that makes the file worth studying — E3, a **negative control**: an unexplained low score must be refused with 400, "Without this, E2 alone would pass even with validation removed." E4 asserts every aggregate row carries its `n` alongside the mean — the Wilson-CI house rule surfacing at the API layer.

THE PACKAGES. Stdlib `urllib.request` in both, not `requests` — zero extra dependency for a script that must run on any box, at the cost of clunkier error handling (`HTTPError` is caught and read for its body). Swapping to `requests` / `httpx` buys retries and connection pooling the soak deliberately doesn't want (a retry would hide a flaky 500).

TRY IT YOURSELF. A seeded mini-soak against any URL, with the correlated-samples lesson built in (uses only stdlib; point it at something harmless):

```
import json, random, urllib.request, urllib.error

BASE = "http://127.0.0.1:9"   # deliberately dead port: watch it fail loudly
def hit(path):
    try:
        with urllib.request.urlopen(BASE + path, timeout=3) as r:
            return r.status, r.read()[:80]
    except urllib.error.HTTPError as e:
        return e.code, b""
    except Exception as e:
        return 0, str(e).encode()[:80]

rng = random.Random(20260719)          # seeded: rerunnable, diffable
failures = []
for i in range(5):
    path = rng.choice(["/", "/health", "/nope", "/api/demo"])
    status, body = hit(path)
    if status == 0 or status >= 500:
        failures.append({"path": path, "status": status, "why": body.decode()})
print(json.dumps({"pass": not failures, "failures": failures}, indent=2))
assert not failures, "soak found availability failures"
```

LIMITS. Self-declared and worth repeating: this is an availability signal, not correctness — "A green soak must never be read as 'the tool works'." The cold-start axis is described in the docstring but there is no restart logic inside the script itself; restarting the target mid-run is an operator action. And the assertion set is invariant-based (no 5xx, no leak), so a page that returns well-formed wrong answers sails through.

HOW TO IMPROVE IT. Latency is measured per request (`secs` in `req()`) but never asserted or summarised — recording p95 per endpoint in the evidence would catch slow-degradation regressions for free. The e2e smoke's negative-control pattern (E3) belongs in the soak too: one scenario that should 500 on a debug-only route would prove the 5xx detector itself works.

LIFT AND REUSE. Genericise `scenario` into a table of `(kind, weight, steps)` and keep three transferable rules: seed everything, spend iterations on variance axes rather than repetition, and make the evidence file name the claims it does not support in the same breath as the one it does.

M8.6 `/root/sdtm_review/services/parity_harness.py` — owns what both arms must agree on

The sdtm_review fork ran R-generated and Python-generated programs from the same spec across nine Cardiology domains, and the arms disagreed on all nine. When the disagreements were classified (the table is in the module docstring), the biggest classes were not about patients at all: 5,566 cells where `--STRESN` was written `71` in one arm and `71.0` in the other — the same number, serialised twice; 3,200 cells where `--SEQ` was numbered `8,15,22` versus `2,3,4` — both valid orderings of an ordinal SDTM only requires to be unique within USUBJID; and three domains where one arm carried `SUBJID` and the other did not. Here parity means something different from M8.2: **cross-language agreement between two independently generated arms**, with no golden truth to referee — so the harness must first remove the disagreements that are merely mechanical, or the real ones drown.

WHAT IT DOES. `parity_harness.py` normalises both arms' produced CSVs identically, in place, using the same Python code after both have run — which "makes these classes identical by construction rather

than by luck." Every change is counted and reported, because "a harness that silently rewrote delivered datasets would be indistinguishable from one that hid a defect."

WHY IT EXISTS. Formatting, ordinal assignment and column selection are "decisions that have one right answer per study and no business being drawn from a language model twice." Deciding them once, deterministically, in the harness, converts the parity diff from noise into signal. A QC SAS programmer does exactly this when the PROC COMPARE spec says "compare with FUZZ, after sorting both by the key, restricted to the shell's column list" — the harness is that comparison protocol, written down as code.

INPUTS AND OUTPUTS. Input: two CSV paths (`r_csv`, `py_csv`), the domain, and optionally the spec-declared column list, natural keys and blankable variables. Entry point:

```
def normalise_pair(r_csv, py_csv, domain,
                   expected_columns=None, keys=None, blank_vars=None) -> dict:
    """Normalise both arms identically, then say what changed in each."""
```

Output: a report per arm — `numbers_canonicalised`, `seq_renumbered`, `columns_added/dropped`, `blanked_unenforceable`, `unit_identity_fixed`, `test_decoded`, `y_or_null_enforced`, `stresc_standardised`, `testcd_too_long`, `uncoded_cleared`, `datetime_harmonised` — with `summarise(reports)` rolling a whole therapeutic area into one Counter. For AE: `AESEQ` gets re-derived from the natural key in both arms; an `AEDECOD` column that is a total copy of `AETERM` gets cleared (see below).

HOW IT WORKS — THE MECHANICAL FIXES. `canonical_number` gives every numeric string one text form (`71.0 → 71`, `0.500 → 0.5`, `1e2 → 100`) and leaves non-numbers — `"71 bpm"`, `"<0.1"`, dates — strictly alone. `renumber_seq` re-derives `--SEQ` from the IG natural key in both arms. `align_columns` forces the spec's column set: a column the spec expects but an arm omitted is added empty ("a true statement"); a column no arm should carry is dropped. The spec, not either arm, is the authority.

HOW IT WORKS — THE DECIDABLE DEFECTS. The more interesting functions repair violations that are decidable without preferring an arm, each grounded in a named real case: `enforce_unit_identity` fires only when `--ORRESU` equals `--STRESU` and the values still differ — the LB PLAT case where Python wrote `LBSTRESN = 441700` against `LBORRES = 441.7` under the same declared unit, "wrong by its own declared unit, which is what makes this decidable without a sponsor conversion table." `enforce_test_decode` fixes `--TEST` via the published CT decode of `--TESTCD` — R wrote "White Blood Cell Count", Python "White Blood Count", and both are wrong; the published term is **Leukocytes**. `enforce_y_or_null` reads CDISC's own variable descriptions (regex on "should be 'Y' or null") rather than a hardcoded list. `enforce_uncoded_dictionary` clears `AEDECOD` only when it copies `AETERM` on every populated record (≥5), because real coding sometimes legitimately returns the verbatim — "Headache" is a genuine MedDRA PT. `enforce_testcd_length` **reports** a `--TESTCD` over 8 characters and never truncates: "Truncating would invent a code nobody chose."

WHAT IT REFUSES TO TOUCH. EPOCH, `--DY` and the DM date columns are real disagreements — in the `MHSTDY = -5716` case one arm fabricated a study day from a reference date that does not exist in the extract. `blank_unenforceable` empties only variables the enforced spec (`spec_enforce`, via `unenforceable_variables()`) marks Not Submitted for want of study context, and the docstring marks the boundary in bold terms: blanking a column because the arms disagree "would be the worst thing in this file: it would turn every disagreement into agreement and destroy the signal the parity check exists to give." That sentence is the entire philosophy of QC normalisation in one line.

THE PACKAGES. Stdlib `csv` / `re` / `collections.Counter`, plus project services (`ct_index`, `knowledge`, `ig_reference`, `spec_enforce`) resolved lazily inside functions so a missing pack degrades to "no change" rather than a crash. No pandas — rows are dicts, mutation is explicit, and nothing can coerce types behind the harness's back.

TRY IT YOURSELF. The signal/noise split on a toy AE pair:

```
import re
_NUM = re.compile(r"^[+-]?(\d+\.?\d*|\.\d+)([eE][+-]?\d+)?$")

def canon(v):
    v = (v or "").strip()
    if not _NUM.match(v):
        return v
    f = float(v)
    return str(int(f)) if f == int(f) else repr(f)

r_arm  = [{"USUBJID": "S-1", "AESEQ": "8", "AESTDY": "3.0",  "AESEV": "MILD"}]
py_arm = [{"USUBJID": "S-1", "AESEQ": "1", "AESTDY": "3",    "AESEV": "SEVERE"}]

for arm in (r_arm, py_arm):
    for row in arm:
        row["AESTDY"] = canon(row["AESTDY"])   # mechanical: one text form
        row["AESEQ"] = "1"                     # mechanical: re-derived ordinal
diffs = [(k, a[k], b[k]) for a, b in zip(r_arm, py_arm)
         for k in a if a[k] != b[k]]
print("surviving diffs:", diffs)
assert diffs == [("AESEV", "MILD", "SEVERE")], diffs
# The formatting noise is gone; the REAL clinical disagreement survives.
```

LIMITS. The harness rewrites delivered datasets in place — the highest-trust operation in either repo — and its safety rests entirely on the "count and report everything" rule plus the tests that pin what it must NOT do (`tests/test_parity_harness.py` is written mostly as prohibitions: "'<0.1' comparator result must survive verbatim", "must not truncate", "only the named column may be blanked"). When both arms are wrong the same way, cross-arm parity is structurally blind — that is what the oracle comparison (M8.7) and the CT-grounded enforcers exist for. And several enforcers are quietly no-ops when a codelist fails to resolve, which is a graceful degradation that also means "green" can include "unchecked".

HOW TO IMPROVE IT. Return the per-enforcer "could not check" state explicitly (the `gates.py` tri-state `pass/fail/unchecked` pattern is already in this repo and is exactly right). A `--dry-run` mode that reports counts without writing would make the in-place mutation auditable before the fact, not just after.

LIFT AND REUSE. The design rule generalises to any double-programming setup, human or machine: sort the diff into (a) mechanical — take it from both arms and derive it once; (b) decidable-by-rule — repair with the rule cited; (c) genuine — never touch, surface loudly. The function-per-defect-class structure, each docstring naming the real incident that motivated it, is also the best incident-log format in either repo: the code is the postmortem.

M8.7 `/root/sdtm_review/scripts/batch_run.py` — orchestrates eleven domains into graded evidence

Run the whole pipeline over every domain of the synthetic study, and at the end there is a directory a skeptic can audit: every generated program, every produced dataset, every check's verdict, SHA-256

of every artifact, the token bill per model, and one manifest that says how many domains actually passed. The first full LLM run under this harness, `data/batch_runs/full11_parity_fixed` (2026-07-21), records **0 of 11 domains passed** — and that honest zero is the baseline the memory files call "baseline 0/11". A harness that could report that number about its own product is the most trustworthy artifact in either repo.

WHAT IT DOES. `batch_run.py` (1,048 lines — the fleet harness) runs, per domain: (1) mapping spec — deterministic from the knowledge pack, or via the live engine under `--llm` mode; (2) R program generation through the guarded LLM cascade; (3) contained execution of that R; (4) cell-for-cell grading against the oracle in `data/synthetic_v4/expected/`; (5) a generated-and-executed testthat suite; (6) Define-XML 2.1; (7) `services/validator.py` conformance; (8) dataset-vs-its-own-define reconciliation; and once study-wide, (9) cross-dataset checks. With `--parity` it adds the Python arm (M8.6's subject matter) as step 5b. Everything lands under `--out` (default `data/batch_runs/latest`): `domains/<DOM>/` working dirs, `study/`, `checkpoint.json`, and `RUN_MANIFEST.json`.

WHY IT EXISTS. Its docstring draws the line against its sibling `robustness_sweep.py`: the sweep is a benchmark that "writes only to –scratch and answers 'how good is the mapper'"; this runner answers "here is the submission package, and here is the evidence it is correct." And it states the module's law as its own contract: "EVIDENCE RULE: only a run writes an evidence file. Every status in the manifest comes from something that actually executed… Where a check could not run, the manifest says so explicitly rather than recording a pass."

INPUTS AND OUTPUTS — THE AE RUNNING EXAMPLE. Input: `data/synthetic_v4/raw/raw_ae.csv` plus the corpus `MANIFEST.json` (which supplies `study_id="SYNTH-001"` — threaded in deliberately, because the raw CSVs carry no study column and a model left to invent one produced `STUDYID = "STUDY123"`, misaligning every USUBJID and voiding the entire grade). Output for AE in the real `full11_parity_fixed` run — this is what a **failing** AE run leaves behind:

```
domains/AE/
  AE_map.R  AE_mapping_spec.json  AE_postprocess.json  AE_reference.csv
  ae.csv  ae.xpt  define_AE.xml  test-ae.R  raw_ae.csv  pyarm/
RUN_MANIFEST.json (AE excerpt):
  status: fail
  oracle: 331 cell(s) differ, aligned_on STUDYID+USUBJID+AEDECOD+AESTDTC
  gates:  fail
  py_parity: fail -- natural key does not align between arms (156 unshared keys)
```

A passing domain has the same files but `"status": "pass"`: oracle `mismatches: 0`, testthat `>0` passed / 0 failed, define recon clean, `gates: pass`. The pass/fail decision is explicit in `process_domain`:

```
hard_fail = (
    result["steps"]["execute_r"]["status"] == "fail"
    or result["steps"]["testthat"]["status"] in ("fail", "error")
    or not result["steps"]["oracle"]["match"]
    or result["steps"]["define_recon"]["status"] == "fail"
    or result["steps"].get("gates", {}).get("status") == "fail"
)
result["status"] = "fail" if hard_fail else "pass"
```

HOW IT WORKS — THE DEFENSES, EACH WITH ITS SCAR. Nearly every helper exists because of a dated, named failure:

- Import hygiene. `_ROOT` is resolved from `__file__`, never hardcoded — a hardcoded `/root/sdtm_review` once made worktree runs import the shared tree's `services/`, so "agents measuring a fix they had just written were grading unmodified code" (found 2026-07-21; it invalidated every

"after" number that workflow produced). `SDTM_DISABLE_OPENROUTER=1` is set before `services.llm` imports, because the key resolves at module load.

- Spec-source gate. `_check_spec_sources` verifies every source column a spec names is resolvable — from the raw data or from an SDTM target defined by an earlier spec row (transitive resolvability; an earlier raw-only version false-flagged legitimate derive-from-derive rows and "disguised a gate defect as a codegen failure"). Fails by column name, before any R runs.
- Program hygiene. `_looks_like_r` catches the model answering with prose instead of code (observed on 2 of 11 domains); `_r_input_contract` restricts the generated program to exactly three file names — its raw input and its own outputs — because AE's program once read `dm.csv`, which is both a broken dependency and "the shape a grading-integrity escape would take, since the reference copy lives in the same directory." Retries are always a fresh regeneration, never a patch — "a patched program is no longer the artifact the customer downloads."
- Containment and the sealed answer key. All generated code runs through `services.sandbox`, and the runner fails closed (`SandboxUnavailable` → "refused to execute generated R without containment"). The corpus containing `expected/` is bind-mounted read-only **only** for the trusted `--no-llm` passthrough, never for LLM output: "a hostile generated program that could read expected/ could copy it as its own output and score a false zero mismatches. Sealing the answer key from untrusted code is a GRADING-INTEGRITY boundary, distinct from the containment boundary."
- Key-aligned grading. `_compare_to_oracle` aligns rows on the domain's natural key from the knowledge pack, not file position — positional zip once inflated 614 real DM differences to 1123 (509 phantom mismatches, "45% of the reported total, every one of them an artifact of row order"). If the keys don't align, that is the finding and no cells are counted.
- The 0F lesson. `_run_testthat` returns a `parsed` flag because testthat, past its failure cap, aborts without printing its summary line; the first parser reported that as "0 failed", i.e. a catastrophically wrong dataset displayed as clean. "A misleading number in an evidence file is precisely the failure mode that produced the fabricated-evidence incident." Now: no summary line → `status: "error"`, never `0F`.
- Money honesty. `_credit_deltas` reports the OpenRouter balance before/after only when actually measured, and explicit non-numbers otherwise — z.ai and OAuth are `flat_rate`, CommandCode is `not_queryable` with the reason spelled out — "never a fabricated delta."

RESUMABILITY AND IDEMPOTENCY. After every completed domain the full results list is rewritten to `checkpoint.json`; `--resume` reloads it and skips domains already at `"status": "pass"`, so a crashed or interrupted fleet run re-does only failures. Domains run concurrently via `ThreadPoolExecutor` (`--workers`, default 4), each in its own `domains/<DOM>/` directory, so parallel arms cannot trample each other's files.

THE STANDING WATCH: `/ROOT/SDTM_REVIEW/SCRIPTS/CONFORMANCE_WATCH.PY`. The batch run measures once; the watch re-measures on a schedule and speaks only when a number moves. Per therapeutic area it runs three detectors over the executed outputs: ClinCoder's 17 CLIN structural rules (the watch's own docstring still says 16 — the M5.3 drift), CDISC CORE's 430 SDTMIG v3.4 rules (~66 s per study), and a dead-source-value detector (`services/value_domains`) for the silent-blank class no conformance rule reports — the motivating case being `AEOUT` blank on all 114 records while four clean source values sat unmapped, invisible because AEOUT is Permissible. Evidence is written every run to `evidence/conformance/_watch/<TA>/watch_<stamp>.json` plus `latest.json` (real files exist for Cardiology, 2026-07-26 onward); notification via `tg_digest.py` fires only when one of four `TRACKED` counters changes, and the process exits 2 when a counter went up, so cron sees regressions without parsing JSON. Its two self-restraints are the design: "It reports. It does not fix" (a job that repairs

cannot report honestly about its own repairs), and "Silence is the default" (a watch that speaks hourly trains its reader to ignore it).

THE PACKAGES. Stdlib for orchestration (`concurrent.futures`, `subprocess`, `csv`, `hashlib`, `urllib`), project services for everything domain-shaped. R-side: `testthat` executed via `Rscript`. Nothing heavier — no Celery, no Airflow; a thread pool and a checkpoint file cover an 11-domain fleet, and the simplicity is what makes the evidence auditable.

TRY IT YOURSELF. A resumable mini-fleet with a checkpoint and a manifest:

```
import json, os, random, time

DOMAINS = ["AE", "DM", "VS", "LB"]
CKPT = "fleet_checkpoint.json"

def process(dom, rng):                 # stand-in for spec->code->run->grade
    time.sleep(0.05)
    return {"domain": dom, "status": "pass" if rng.random() > 0.4 else "fail",
            "mismatches": 0 if rng.random() > 0.4 else rng.randint(1, 400)}

done = {}
if os.path.exists(CKPT):
    done = {r["domain"]: r for r in json.load(open(CKPT))["results"]
            if r["status"] == "pass"}
    print("resume: skipping", sorted(done))

rng = random.Random(20260721)
results = list(done.values())
for dom in DOMAINS:
    if dom in done:
        continue
    results.append(process(dom, rng))
    json.dump({"results": results}, open(CKPT, "w"), indent=1)  # after EVERY domain

passed = sum(r["status"] == "pass" for r in results)
json.dump({"domains_passed": passed, "domains_total": len(results),
           "domains": results}, open("RUN_MANIFEST_demo.json", "w"), indent=1)
print(f"passed {passed}/{len(results)} -- rerun me: only failures re-execute")
```

LIMITS. The oracle is the synthetic SYNTH-001 corpus — grading transfers to real studies only as far as the corpus's shapes do. The hard-fail rule treats a parity failure as evidence, not a run failure ("it does not say WHICH arm is wrong"), so `--parity` findings need a human. Thread-level parallelism means a hung LLM call occupies a worker for its full timeout. And `checkpoint.json` keys on domain+status only: a code change between resume legs silently mixes two versions' results into one manifest — the manifest records one git state nowhere.

HOW TO IMPROVE IT. Stamp the git sha and dirty-flag into both `checkpoint.json` and `RUN_MANIFEST.json`, and refuse `--resume` across a differing sha (the exact worktree-import incident class, one directory over). Promote the per-domain result dict to a documented schema — three consumers (checkpoint, manifest, run_report) already depend on its shape by convention.

LIFT AND REUSE. The portable core is the evidence rule plus four mechanisms: checkpoint-after-every-unit, fail-closed sandboxing with a separately-argued answer-key seal, key-aligned comparison that refuses to count cells when keys don't align, and a manifest that distinguishes `pass` / `fail` / `error` / `unchecked` everywhere. Interface to copy: `process_unit(unit, meta, out_root) -> result_dict that never raises`.

M8.8 `/root/sdtm_review/tests/test_gates.py` — the suite that makes gates falsifiable

`tests/test_gates.py` opens with the module thesis as its literal docstring: "A gate that cannot fail is not evidence, so every gate has a negative control that MAKES it fail; and 'could not check' must surface as 'unchecked', never as a pass." This chapter uses that file as the door into the whole suite — 59 test files under `/root/sdtm_review/tests/` (the mapper fork has only 4: `test_codegen_prompt.py`, `test_llm_cli_refusals.py`, `test_privacy.py`, `test_sec_scan_scope.py` — the review fork is where the testing culture lives).

WHAT IT DOES. For `services/gates.py` — the millisecond, fail-by-name post-run gates batch_run invokes as step 3c (`not_submitted_created`, `required_nonblank`, `ct_conformance`, `key_integrity`) — the tests build a miniature DM in `tmp_path` and then break it one way per test: drop a spec target, blank a Required variable, write `SEX="Male"` against the CT, duplicate the natural key, write the wrong `STUDYID`. Each asserts not just `status == "fail"` but that the offending name or value appears in the detail — `assert g["detail"]["SEX"] == ["Male"]`. One test monkeypatches the CT loader to return nothing and pins the tri-state:

```
def test_ct_unreadable_is_unchecked_not_passed(tmp_path, monkeypatch):
    monkeypatch.setattr(ct_mod, "load_codelists", lambda path=None: {})
    rep = _run(tmp_path, GOOD)
    assert rep["gates"]["ct_conformance"]["status"] == "unchecked"
```

"could-not-measure must never read as measured-clean" (gates.py's own words) is thereby a tested property, not a comment.

WHAT THE WIDER SUITE COVERS. Reading across the representative files: `test_core_engine.py` drives a captured real CORE report (`tests/fixtures/core_report_sample.json`) through the normaliser — offline by design, "CORE takes ~66 s... a suite that invoked it would stop being run" — and guards hardest that `execution_error` never becomes a pass: on the first real run 177 of 430 rules could not evaluate, and a normaliser that counted those as successes "would have reported 62% conformance instead of the truth." Unknown statuses degrade to `execution_error` ("fail safe"), and the pass-rate denominator excludes unevaluated rules. `test_fleet_map_validate.py` documents its own origin — "There were NO tests on validate() before this file. That is how the defect below shipped mid-flight" — and pins the fatal-vs-row-level split (one bad mapping row must not discard 27 good ones; rejected rows are dropped, never repaired) plus a fleet-integrity property: no routing tier may ask the same underlying model twice, or "'both models agreed' becomes one model agreeing with itself."
`test_parity_harness.py` is written almost entirely as prohibitions on the in-place rewriter (M8.6). `test_codegen_prompt.py` (21 offline tests) pins the prompt-as-code defects from the 2026-07-20 0/2 probe. Beyond these: security (`test_security_paths.py`, `test_mapping_route_security.py`), sandbox behaviour, define.xml and recon, spec enforcement, study context, review scoring, and LLM plumbing (`test_llm_usage.py`, `test_llm_cli_refusals.py`) — the seams around the model, never the model.

WHY IT EXISTS. Every load-bearing test file here was written after a specific shipped defect, and most say so with a timestamp. This is regression-driven testing in its purest form: the suite is a ledger of everything that has already gone wrong once.

INPUTS AND OUTPUTS. Run with `python3 -m pytest tests/ -x -q` from `/root/sdtm_review`. Everything is offline: fixtures are inline dicts, `tmp_path` CSVs, or one captured JSON report. No test spends a token or needs the service running (`e2e_smoke.py` covers that seam, outside pytest).

HOW IT WORKS. Two mechanics worth stealing: tests import script-files without packaging via `importlib.util.spec_from_file_location("_fleet_map", .../scripts/fleet_map.py)`, so even `scripts/` code is testable; and fixtures are minimal by construction (a 4-variable DM pack, two subjects) so a failure message is readable in one screen.

THE PACKAGES. `pytest` with `tmp_path` and `monkeypatch` only — no plugins, no fixtures framework, no mocking library beyond monkeypatch. The cost of that austerity is repetition (`_write`/`_run` helpers re-appear per file); the benefit is that any Python beginner can read every test.

TRY IT YOURSELF. The negative-control discipline in one runnable file:

```
def ct_gate(rows, codelist):
    if not codelist:                       # could not check != clean
        return {"status": "unchecked"}
    bad = sorted({r["SEX"] for r in rows if r["SEX"] not in codelist})
    return {"status": "fail", "detail": bad} if bad else {"status": "pass"}

good = [{"SEX": "M"}, {"SEX": "F"}]
CT = {"M", "F", "U"}

assert ct_gate(good, CT)["status"] == "pass"
assert ct_gate([{"SEX": "Male"}], CT) == {"status": "fail", "detail": ["Male"]}
assert ct_gate(good, set())["status"] == "unchecked"   # never "pass"
print("gate has a working negative control -- it is allowed to say PASS now")
```

LIMITS — THE HONEST CHARACTERIZATION. What the suite pins down: deterministic logic around every seam — gate verdicts, normalisers, validators, routing, security paths, prompt text. What it does not and cannot pin: the LLM outputs themselves (every model call is out of scope; the flakiness of generation is measured by paid harness runs like `flaky_dm_1..5` under `data/batch_runs/`, not by pytest), the R runtime behaviour of generated programs (testthat inside batch_run owns that, per run), true end-to-end HTTP (delegated to `e2e_smoke.py` and `visitor_soak.py`), and CORE itself (only its report-normaliser is tested). No coverage percentage is measured anywhere in either repo, and consistent with house rules I will not invent one — the honest statement is: the seams are dense with regression tests; the stochastic core is deliberately tested by executed evidence instead.

HOW TO IMPROVE IT. A CI entry point that runs pytest and the deterministic gates in one command with one exit code (today `evidence/step4_*.txt` files suggest a manual step sequence). Property-based tests (hypothesis) for `canonical_number` and `_compare_to_oracle` — both are parsers of adversarial text and currently tested only on enumerated examples.

LIFT AND REUSE. The reusable convention is the docstring-as-incident-report: every test file names the defect, the date, and the mechanism it now forbids. A new engineer reading `tests/` learns the project's failure history faster than from any wiki — and the checker that audits this study pack would find every one of those symbols real.

What this module still owes you

Debts, stated plainly. **COVERAGE NUMBERS:** no line/branch coverage is measured in either repo, and none was invented here; "59 test files" counts files, not coverage. **The LLM layer is gated only by paid, low-n runs:** `codegen_parity` evidence is n=4-or-fewer per label, single-run observations by its own reporting rule; snapshot-replay tests for `generate_mapping_spec`/`classify_domain` are named as the missing piece in `parity_gate.py`'s docstring and still do not exist. **SAS remains delivered, never executed:** `sas_deliverable` is a delivery decision; no gate anywhere runs a `.sas` file, and nothing in this module

can tell you the SAS translation is faithful beyond the shared-spec argument. **The historical false-greens are reconstructed from the code's own docstrings** (the 2026-07-17 hardcoded-pass incident, the model-swap non-measurement, the worktree self-grading trap) — the design guards against these classes today, but the original incident artifacts were not re-audited this session. **Fork drift is live:** `visitor_soak.py` is byte-identical across forks while `iq_gate.py` has silently diverged (`shell=True` vs `shlex`), and nothing gates the gates' own divergence. **And the deepest debt:** the oracle itself. Every cell-level grade in the review fork traces to `data/synthetic_v4/expected/` — a synthetic corpus the project generated. The gates prove the pipeline reproduces that truth; whether that truth matches what a study statistician would sign remains, as ever, a human's signature away.

Module 9 — Web and ops layer

This module covers how the SDTM Mapper and SDTM Review codebases turn Python functions into services a browser can reach: two Flask apps (`/root/sdtm_mapper/app.py` and `/root/sdtm_review/app.py`), the 63 KB `routes_ta.py` route file, the audit and watermark stamps applied to every request, the `sec_scan.py` security gate, and the four systemd units that keep it all running on ports 8812, 8813, 9101 and 9113 of this host. If you come from SAS, the core mental shift is this: a batch job runs once, writes a log, and exits — a web service is a program that never exits, and every browser click is a tiny batch job dispatched into it. Everything in this module is about what that shift costs and buys: request/response instead of submit/log, in-memory job state instead of a WORK library, and systemd instead of cron.

THE BRIDGE, EXPLICITLY. Before the chapters, here is the translation table this module keeps returning to. Each row is where a SAS-server instinct lands in this stack — and the last column is where that instinct misleads:

SAS-server concept	This stack	Where the analogy breaks
Submit a program, read the log	`POST` a request, read the JSON response	The response is for the caller; the log (`service.log`) is separate and the caller never sees it. A 200 response with `"ok": false` inside is normal, not a contradiction.
WORK library (session-scoped scratch)	The `JOBS` dict + `output/<jid>/` directory	WORK dies with your session on purpose; `JOBS` dies whenever the service restarts — including for other people's crashes. Disk artifacts in `output/` survive; the in-memory session does not.
`%include` shared macro files	Flask blueprints (`register_blueprint`)	An `%include` failure kills the job loudly. A guarded blueprint failure here boots a "healthy" app with a missing feature (M9.2).
Batch scheduler / cron	systemd units (M9.6)	cron starts things; systemd also keeps them alive (`Restart=always`) and injects their environment. There is no equivalent of "the job finished" — a service is only ever up or down.
The LOG's NOTE/WARNING/ERROR discipline	HTTP status codes + JSON error bodies	Status codes are for machines and are deliberately vague to strangers: `_fail()` returns "The server failed to handle this request" while the real traceback goes server-side only. On a public URL, a detailed error is a leak.
Program header block (`/* author, date */`)	`audit.py` rows + watermark headers (M9.4)	The SAS header says what the author claims; the audit row records what actually ran — including which LLM. The fixed-model-attribution bug in `api_execute` is exactly the gap between those two.
One user = one SAS session	One process serves everyone concurrently	This is the deepest shift. Two reviewers clicking Execute at once share the same process, the same `JOBS`, the same demo job ids — which is why `_job_lock(jid)` exists, and why the race it fixes was found by a concurrency soak test, never by sequential testing.
Metadata-driven access (PROC PWENCODE, metadata ACLs)	An issued API key resolved server-side (M9.3)	The key proves possession, not identity claims: the server looks up who the key belongs to and ignores anything the request body says about who is asking. Fail closed: no key, no write.

Keep the table in reach; every chapter's "Limits" section is mostly one of these rows biting.

M9.1 `/root/sdtm_mapper/app.py` — one file wires the whole mapping pipeline

You double-click nothing. There is no `sas -sysin ae_map.sas` here. Instead a process started weeks ago is sitting on port 8812, blocked on `accept()`, and when a browser posts `ae_raw.csv` to `/api/profile`, a function named `api_profile` wakes up, runs for a few seconds, returns a JSON blob, and goes back to sleep. That standing process is this file.

WHAT IT DOES. `app.py` is the entire HTTP surface of the SDTM Mapper: it accepts a raw clinical dataset, classifies which SDTM domain it is (AE, DM, VS...), generates a mapping specification, generates SAS/R/Python programs from that spec, executes the Python and R arms, validates the result, and serves every artifact back as a download. Fourteen routes (`grep -c '@app.route'` says 14), one file, 427 lines.

WHY IT EXISTS. The alternative is what clinical programmers actually do today: email a raw dataset to a programmer, wait days, get back a spec and a program. This file compresses that loop to minutes and makes it self-serve. Without it, the `services/` modules (engine, codegen, executor, validator — covered in other modules) are a library with no door: callable only by someone with shell access to the box.

INPUTS AND OUTPUTS. Input: multipart file uploads (CSV/SAS7BDAT) or JSON bodies like `{"job_id": "a1b2c3d4e5f6", "domain": "AE"}`. Output: JSON. A successful `/api/profile` on a synthetic AE file returns roughly `{"job_id": "3f9c...", "raw_name": "ae_raw.csv", "profile": {...}, "classification": {"domain": "AE", "label": "Adverse Events", ...}}`. Files come back through one route, `/api/download/<jid>/<fname>`, as attachments (the spec .xlsx, the generated `AE_map.sas`, the built `ae.csv`/`ae.xpt`).

HOW IT WORKS. The file has three layers. First, module-level wiring that runs once at boot:

```
#| skip-check
app = Flask(__name__, static_folder=STATIC, static_url_path="")
app.config["MAX_CONTENT_LENGTH"] = 50 * 1024 * 1024
JOBS = {}
```

`JOBS` is the whole session model: an in-process Python dict keyed by a 12-hex-char job id. Second, per-request hooks: an `@app.after_request` that stamps `X-Response-ID` and `X-ClinCoder-Watermark` headers on every response (see M9.4), and an `@app.before_request` API-key gate from `services.apikey` that is OFF unless `API_KEY_ENFORCE=1`. Third, the routes, which form a strict pipeline the front-end walks in order — this is the AE running example, verbatim from the code:

1. `POST /api/profile` — upload `ae_raw.csv`; `profile_dataset()` + `classify_domain()` run; a `jid` is minted and `JOBS[jid]` created; `audit.log(jid, "classify", ...)`.
2. `POST /api/spec` — `generate_mapping_spec()` builds the AE mapping spec; `write_excel`/`write_word` produce `AE_mapping_spec.xlsx`/`.docx` into `output/<jid>/`.
3. `POST /api/code` — `generate_all_code()` writes `AE_map.sas`, `AE_map.R`, `AE_map.py`; the response carries `"verification": {"sas": "unverified", ...}` because generation is not execution.
4. `POST /api/execute` — under `_job_lock(jid)`, `run_python_program()` builds `ae.csv`/`ae.xpt`, then `run_r_program()` runs the R arm and compares cell-for-cell; SAS is only ever released on the strength of R agreeing with Python (there is no SAS runtime on this box).
5. `POST /api/validate` — `validator.validate()` checks the built `ae.csv` against SDTMIG rules and writes `AE_conformance_report.xlsx`. The mapper (not the review fork) additionally has `POST /api/validate_core`, which runs the CDISC CORE engine (its docstring says ~10–25 s).
6. `GET /api/download/<jid>/ae.xpt` — collect the artifact.

Every handler resolves its job through one helper, `_job(request)`, which returns `(data, None)` for unknown ids so the route can answer 400 instead of a KeyError-turned-500. Errors funnel through `_err(e)`, which returns the traceback only when `SDTM_DEBUG=1` — the comments record that both helpers exist because scanners and stale browser tabs used to turn crashes into traceback disclosures on the public URL. There is also a demo-replay layer (`/api/demo/<domain>/<step>`) that serves pre-computed JSON snapshots from `demo_snapshots/` for live presentations, no LLM calls.

THE PACKAGES. Flask, and only Flask, for the web layer — `pandas` appears in routes only to count columns for `/api/library`. Flask fits because this is a small-team tool with server-rendered static HTML and JSON endpoints; FastAPI would buy async and automatic OpenAPI docs, but every heavy route here is CPU/subprocess-bound (LLM calls, executing generated programs), so async buys little, and there is no published API contract to document. Django would impose an ORM and an app structure on what is deliberately one file. Cost of a swap to FastAPI: rewrite all 14 handlers' request parsing (`request.get_json(silent=True)` → Pydantic models), replace `send_from_directory` calls, and re-test the error-hiding behavior — days of work for features this app does not use.

TRY IT YOURSELF. The SAS-programmer bridge in 25 lines: a Flask app you can exercise without ever starting a server, using `test_client()` — the same trick this repo's own tests use.

```
import io
from flask import Flask, request, jsonify

app = Flask(__name__)
JOBS = {}

@app.route("/api/profile", methods=["POST"])
def api_profile():
    f = request.files.get("file")
    if f is None:
        return jsonify({"error": "no file"}), 400
    header = f.stream.readline().decode().strip().split(",")
    jid = f"job{len(JOBS):04d}"
    JOBS[jid] = {"raw_name": f.filename, "columns": header}
    return jsonify({"job_id": jid, "n_cols": len(header)})

client = app.test_client()
ae = b"USUBJID,AETERM,AESTDTC\n001,HEADACHE,2026-01-02\n"
resp = client.post("/api/profile",
                   data={"file": (io.BytesIO(ae), "ae_raw.csv")})
assert resp.status_code == 200
assert resp.get_json()["n_cols"] == 3
print("uploaded:", resp.get_json(), "| jobs held in memory:", JOBS)
```

LIMITS. `JOBS` is a plain dict: every job dies on restart (`Restart=always` means any crash silently wipes all in-flight sessions), it grows without bound until restart, and it confines the app to exactly one process — you cannot scale to two workers because worker B has never heard of worker A's job ids. `app.run(host="127.0.0.1", threaded=True)` is Flask's development server in production; it works at this traffic level but is single-process and has none of gunicorn's worker recycling. `/api/download/<jid>/<fname>` is unauthenticated: anyone who learns or guesses a 12-hex job id can fetch that job's artifacts (`send_from_directory` does prevent path traversal). These are observed properties of the code, not measured failure events.

HOW TO IMPROVE IT. Move `JOBS` to SQLite (the repo already uses SQLite for audit and review — reuse the pattern) keyed by jid, so restarts stop destroying sessions and a second worker becomes possible; then front it with gunicorn (`gunicorn -w 2 app:app`) instead of `app.run()`. Add the job id's owner to `JOBS` and check it in `/api/download`.

LIFT AND REUSE. The reusable shape is the pipeline-over-jobs pattern: `POST` steps that each take `{"job_id": ...}`, one `_job()` resolver that turns unknown ids into 400s, one `_err()` that hides tracebacks behind a debug flag, one `_job_lock(jid)` for per-job serialization, and one download route per job directory. That five-piece kit ports unchanged to any "upload, then run N stages, then download" tool. Genericise by making the stage list data (`STAGES = [...]`) instead of copy-pasted handlers.

M9.2 `/root/sdtm_review/app.py` — the fork that grew four blueprints

Open this file and the first line lies to you: `"""ClinCoder SDTM Mapper — Flask application."""` — but you are in `/root/sdtm_review`, a different repo, serving a different port. That docstring is a fossil. sdtm_review began as a copy of sdtm_mapper, and this file is the clearest record of what happened next: the pipeline core stayed nearly identical, and four new route modules were bolted on around it.

WHAT IT DOES. Same mapping pipeline as M9.1 (also 14 `@app.route` handlers, 576 lines, default port 8813 instead of 8812), plus the registration of four Flask blueprints — `routes_review`, `routes_mapping`, `routes_study`, `routes_ta` — which together add the team-review layer: run scoring, per-variable mapping adjudication, study browsing, and the TA spec review covered in M9.3.

WHY IT EXISTS. The mapper answers "turn this one dataset into SDTM". The review fork answers "let four people examine, comment on, and overrule what the fleet produced across a whole therapeutic area". Those needs arrived after the mapper shipped, and forking the repo (rather than growing the original) kept the interview-proven mapper stable. The price of that choice is two diverging copies of everything — `services/watermark.py` is still byte-identical across the repos (verified with `diff`), while `app.py` has drifted: the review fork gained `/api/rerun_r` and lost the mapper's `/api/validate_core`; the mapper's health endpoint reports 7 domains on SDTMIG v3.4, the review fork 21 domains on SDTMIG v4.0 (both verified live with `curl /api/health`).

INPUTS AND OUTPUTS. Identical shapes to M9.1 for the pipeline routes. The interesting new output is boot-time stderr: each blueprint registration is wrapped so a broken module produces a line like `[ta] blueprint NOT registered: <error>` instead of a dead app.

HOW IT WORKS. A Flask blueprint is a batch of routes defined in another file and merged into the app at boot — the SAS analogy is `%include`, except each include is wrapped in its own recovery block. The wiring is four near-identical stanzas:

```
#| skip-check
try:
    from routes_ta import bp as _ta_bp
    app.register_blueprint(_ta_bp)
except Exception as _e:
    print(f"[ta] blueprint NOT registered: {_e}", file=sys.stderr)
```

The comments above each stanza are load-bearing documentation. The `routes_mapping` one records that the blueprint was withdrawn once (2026-07-21) for returning tracebacks to clients and accepting reviewer identity from the request body, and reinstated only after both were fixed and pinned by `tests/test_mapping_route_security.py`. The `routes_ta` one states the design rule the whole fork follows: writes fail closed — with no server-side identity every write is a 403, so registering a blueprint can never silently start accepting unattributed records. The `routes_review` stanza additionally calls `_review.init_db()` (idempotent, creates `data/review.db` — the repo-root `review.db` is a zero-byte stray, per M6). Route ordering does not matter here because there are no overlapping paths: the app's own

routes live under `/api/*`, the blueprints under `/api/datasets`, `/api/mapping/*`, `/api/study/*`, `/api/ta/*` (10, 11, 11 and 40 handlers respectively, counted by `grep -c '@bp.route'`).

One more difference from the mapper matters operationally and is stated in `routes_ta.py`'s docstring: `sdtm-review.service` runs this file **straight from the git tree**, so "assume anything committed here is one restart away from being live." The mapper, by contrast, serves from a promoted release directory (see M9.6).

THE PACKAGES. Flask blueprints are the stdlib-of-Flask answer to "my routes file is too big" — no new dependency, no framework change. The alternative inside Flask would be multiple `app` objects behind `werkzeug.middleware.dispatcher.DispatcherMiddleware`; blueprints are simpler because all routes share one config and one before/after-request stack — which is exactly how the watermark and API-key hooks in this file automatically cover all four blueprints' routes too.

TRY IT YOURSELF. Guarded blueprint registration, self-contained — one good blueprint, one that explodes on import, and proof the app survives:

```
from flask import Flask, Blueprint, jsonify

app = Flask(__name__)

ta = Blueprint("ta", __name__)

@ta.route("/api/ta/list")
def ta_list():
    return jsonify({"tas": ["Cardiology", "Oncology"]})

def register(app, loader, name):
    try:
        app.register_blueprint(loader())
    except Exception as e:
        print(f"[{name}] blueprint NOT registered: {e}")

register(app, lambda: ta, "ta")
register(app, lambda: 1 / 0, "broken")   # a bad module must not kill boot

client = app.test_client()
assert client.get("/api/ta/list").get_json() == {"tas": ["Cardiology", "Oncology"]}
assert client.get("/api/nope").status_code == 404
print("app booted with 1 of 2 blueprints; still serving")
```

LIMITS. The guarded-registration pattern trades a loud failure for a quiet one: if `routes_ta.py` has a syntax error, the service boots "healthy" and every `/api/ta/*` URL 404s; the only evidence is one stderr line in `logs/service.log`. Nothing probes at boot that all four blueprints actually registered. The fork-divergence cost is real and already documented inside the repos themselves: `/root/sdtm_review/CLAUDE.md` warns "The knowledge packs differ between forks… Assume nothing about domain metadata across the two forks." And the stale module docstring means anyone grepping for "SDTM Review" will not find this file.

HOW TO IMPROVE IT. Add the blueprint roster to `/api/health` — `{"blueprints": ["review", "mapping", "study", "ta"]}` built from `app.blueprints.keys()` — so a failed registration is visible to a monitor, not just buried in stderr. Fix the docstring. Longer term, the honest fix for the fork is to make the pipeline core a shared package both repos import, but that is a migration project, not an edit.

LIFT AND REUSE. Take the guarded-registration function itself (the `register()` above is the whole idea) plus the fail-closed rule as an interface contract: a route module may fail to load, but it may never

load in a state that accepts unattributed writes. Any multi-module Flask app can adopt both in an afternoon.

M9.3 `/root/sdtm_review/routes_ta.py` — forty routes for reading, judging, recording

Four reviewers have the Cardiology tab open at once. One is paging through the raw AE dataset, one is reading the generated AE program line by line, one is arguing in a comment thread anchored to `AETERM`, and one just clicked "rebuild plan". Every one of those clicks lands in this file — 63 KB, 1,582 lines, 40 route handlers, no second blueprint, no split.

WHAT IT DOES. `routes_ta.py` is the TA (therapeutic-area) Spec Review API: it serves the adjudicated mapping specs, the raw datasets, every generated program and its executed output, five different conformance readouts, and an append-only ledger of comments, proposals and mapping decisions — for one TA at a time, to a small group of named reviewers.

WHY IT EXISTS. The mapper produces specs; nobody should trust them unreviewed. This file is the review room: it makes every raw column accountable (what mapped it, what decided that, what is still open) and makes every human remark attributable and permanent. Without it, review happens in emailed Excel files and the "why" of every mapping change evaporates.

INPUTS AND OUTPUTS. Reads are `GET` with query params: `/api/ta/raw?ta=Cardiology&domain=AE&offset=0&limit=100&fc=AESER&fv=Y` returns `{"ta": "Cardiology", "domain": "AE", "columns": [...], "rows": [...]}` — paged, filterable, searchable server-side. Writes are `POST` with JSON: `/api/ta/comment` takes `{"ta": "Cardiology", "domain": "AE", "anchor": "AETERM", "body": "..."}` and returns the recorded event. Two routes return HTML instead of JSON: `/ta` (serves `static/ta_review.html`) and `/ta/mapper` (server-rendered by `services.mapper_view`); `/api/ta/plan/graph` sends a pre-built HTML file; the Excel exports return `.xlsx` attachments via `send_file`.

HOW IT WORKS. The file is organized as labelled sections, in order — this is its answer to "where are the blueprints?": comment-rule sections instead.

- **Guards** (lines 55–120): `_fail()` logs the exception and returns `{"error": <ExceptionName>, "detail": "The server failed to handle this request."}` — the traceback never leaves the box. `_reviewer_identity()` resolves who is calling from `X-API-Key` header or `ta_key` cookie (the cookie exists because iframes and `<a href>` downloads cannot carry headers), looks the key up in the key file, and returns `"apikey:<email>"` or `None`. `_require_reviewer()` turns `None` into a 403; `_require_reader()` applies the same gate to reads unless `TA_REVIEW_OPEN_READS=1`. Its docstring documents a subtle real bug avoided: the sibling `routes_mapping` honors an `SDTM_REVIEWER` env fallback, and because the shared env file sets `SDTM_REVIEWER=bhanuji.d@gmail.com`, honoring it here would authenticate every anonymous public visitor as Bhanu. So this file takes the key path only, pinned by a test.
- **Loading** (127–160): `_pack()` and `_raw()` are module-dict caches over `ta_pack.build_pack()` and CSV parsing — the comment gives the reason as a hot path ("four clients polling every 3s" against 8,840 CSV rows, the docstring's number, not mine). `_int()` validates paging params into `(value, error_response)` pairs.
- **Reads** (168–816): `whoami`, `list`, `pack`, `raw`, `distinct` (value counts capped at `DISTINCT_CAP = 200` with the true total alongside), `feed` (the polling stream), `programs` (cached generated code, with line comments marked `stale` when the program digest changed under them), `output` (the executed dataset — cache-only; the page can never trigger execution), `ct`, `knowledge`, `compliance`, `compliance/summary`, `core` (stored CDISC CORE results; `stale: true` means nobody ran it),

`dead_values` (source values that map to nothing — defects no conformance rule reports), `plan`, `plan/graph`, `plan/download`, `registry`, `thread`. The only read-section POSTs are deliberate-cost actions: `core/run` (~66 s per its docstring) and `knowledge/rebuild`.

- **Writes** (824–921): `ping` (presence), `comment`, `propose`, `resolve` — each begins `who, err = _require_reviewer()`. A comment on a program line also records the `program_digest` it was written against, so regeneration cannot silently re-point remarks at new code.
- **Excel** (929–957): `/api/ta/export/<kind>` for `specs` or `raw` workbooks.
- **Mapper view** (971–1534): `/ta/mapper` plus `/api/ta/mapper/{list,triage,llm_propose,rebuild,plan,workbook,master,decide,changelog}`. `decide` is the heaviest handler (~150 lines): actions `map/comment/lock/unlock/swap`, refusing any change without a written reason ("the ledger records why a mapping changed, not only that it did"). `rebuild` is single-flight per TA via an `O_EXCL` lockfile with a 300-second staleness break. `llm_propose` is double-gated: reviewer identity and `SDTM_MAPPER_LLM=1`, and it masks model vendor names into tiers before anything reaches the client.
- **Study graph** (1542–1567): the synthetic study's dependency graph as JSON and as an iframe-able HTML page.

THE PACKAGES. Flask primitives only — `Blueprint`, `jsonify`, `request`, `send_file`, `send_from_directory` — plus stdlib `csv`, `json`, `os`, `time`, `logging`. The pattern worth naming: heavy service imports (`spec_bridge`, `mapping_review`, `mapping_graph`) happen inside handlers, so a broken service breaks one endpoint, not the module import — the route-level version of M9.2's guarded blueprints.

WHAT A 63 KB ROUTES FILE COSTS. Honestly: less than you'd guess, and the bill is specific. What it does not cost is correctness or navigability — the section comments, uniform guard prologue (`who, err = ...; if err: return err` appears ~35 times) and uniform error shape make any single handler readable in isolation. What it does cost: (1) merge congestion — every feature touches this file, so parallel branches collide here first; (2) test-target bloat — you cannot import "just the mapper-view routes"; importing the module drags in twelve services via the top-level `from services import (...)`; (3) the guard prologue is copy-paste discipline, and discipline eventually misses a route (nothing structural prevents an ungated handler). How I'd split it: three blueprints along the existing comment seams — `ta_read.py` (reads + exports), `ta_ledger.py` (writes), `ta_mapper.py` (mapper view + study graph) — sharing one `ta_guards.py` for `_require_reader` / `_require_reviewer` / `_fail`. That is a mechanical move; the URL space and behavior would not change. A `before_request` hook on each blueprint could then replace the 35 copy-pasted prologues with one gate per blueprint.

TRY IT YOURSELF. The file's signature move — server-resolved identity with fail-closed writes — in 24 lines:

```
from flask import Flask, Blueprint, jsonify, request

KEYS = {"sdtmm_abc123": {"email": "ramesh@example.com", "revoked": False}}
bp = Blueprint("ta", __name__)

def identity():
    key = (request.headers.get("X-API-Key") or "").strip()
    meta = KEYS.get(key)
    return f"apikey:{meta['email']}" if meta and not meta["revoked"] else None

@bp.route("/api/ta/comment", methods=["POST"])
def comment():
    who = identity()
    if not who:                       # fail CLOSED: no identity, no write
        return jsonify({"error": "unattributable"}), 403
    body = request.get_json(silent=True) or {}
    return jsonify({"ok": True, "reviewer": who, "anchor": body.get("anchor")})

app = Flask(__name__)
app.register_blueprint(bp)
c = app.test_client()
assert c.post("/api/ta/comment", json={"anchor": "AETERM"}).status_code == 403
ok = c.post("/api/ta/comment", json={"anchor": "AETERM"},
            headers={"X-API-Key": "sdtmm_abc123"})
assert ok.get_json()["reviewer"] == "apikey:ramesh@example.com"
print("unattributed write refused; keyed write recorded:", ok.get_json())
```

LIMITS. The `_pack_cache` / `_raw_cache` dicts never invalidate: if the corpus CSVs change on disk, the running process serves stale data until restart. Raw datasets load fully into memory per (ta, domain). The per-route gating is convention, not construction — a new route added without the prologue ships open, and only review catches it. The key lookup reads the key file on every request (`_load_keys(KEYS_FILE, {})`) — fine at four reviewers, measured by nobody at forty.

HOW TO IMPROVE IT. The three-blueprint split above; a `@bp.before_request` identity gate with an explicit allowlist of open endpoints (`whoami`, the UI page) so "gated" becomes the default instead of the convention; mtime-based cache invalidation in `_raw()`.

LIFT AND REUSE. Three things travel well: the `(value, error_response)` validator convention of `_int()`, which keeps handlers flat; the cache-only read, CLI-only compute rule (`programs`, `output`, `core` never trigger generation — a page load must never spend model money or minutes); and the stale-comment digest trick — stamp any annotation with a hash of the artifact it annotates, and mark it detached when the hash moves.

M9.4 `/root/sdtm_review/services/audit.py` and `watermark.py` — every run recorded, every response stamped

Six months from now someone asks: "the AE spec you delivered in August — which model generated it, at what temperature, from which exact input file?" In SAS-shop terms this is the question the program header, the LOG file and the versioned raw snapshot answer together. Here, both repos answer it with 43 lines of SQLite.

WHAT IT DOES. `audit.py` appends one row per pipeline stage — classify, spec, execute, rerun_r, validate, auto_gate — into `logs/audit.db`, recording timestamp (UTC), job id, stage, raw filename, SHA-256 of the input file, domain, model id, temperature, IG version, coverage %, row count, repair count, and a

JSON detail blob. `watermark.py` (byte-identical in both repos, verified by `diff`) stamps every HTTP response with two headers and offers a comment banner for generated programs.

WHY IT EXISTS. Two different threats. Audit: regulatory traceability — the docstring says "21 CFR Part 11 direction", and the comments in `app.py` show it being taken seriously: the execute route was fixed because it hardcoded `"anthropic/claude-sonnet-4.6"` as the model while `commandcode:mimo-v2.5-pro` actually ran — "an audit record that states the wrong model is worse than none." Watermark: IP leakage — if a generated `AE_map.sas` shows up somewhere it shouldn't, the `rid` in its banner ties it to a specific response from this service.

INPUTS AND OUTPUTS. `audit.log("3f9c...", "execute", raw_name="ae_raw.csv", input_path=src, domain="AE", model="deepseek/deepseek-v4-pro", temperature=0.0, n_rows=114, detail={"ok": True})` → one row. `audit.recent(25)` → list of dicts, served raw at `GET /api/audit`. Watermark output, observable on any response from ports 8812/8813: `X-Response-ID: ccx-<20 hex>` and `X-ClinCoder-Watermark: CCWM-<16 hex>`.

HOW IT WORKS. `audit._conn()` opens `logs/audit.db` and runs `CREATE TABLE IF NOT EXISTS runs(...)` every call — no migrations, no schema version, boring and effective. `log()` computes `file_sha256(input_path)` by streaming 8 KB chunks, inserts, commits, closes. The whole body sits in `try/except: pass`. `recent(n)` selects the last n rows ordered by id descending. There is no UPDATE or DELETE statement anywhere in the file — "append-only" is true of the code, though nothing stops an operator with sqlite3 from editing the file (Part 11 would want tamper-evidence this does not have). Watermark is three functions:

```
#| skip-check
def response_id():
    return "ccx-" + uuid.uuid4().hex[:20]

def watermark_tag(seed=""):
    return "CCWM-" + hashlib.sha256((str(seed) + str(time.time())).encode()).hexdigest()[:16]
```

plus `code_banner(domain, rid)`, which renders a `/* ... */` header naming vendor, domain, rid and watermark for embedding in generated programs. The stamping itself lives in each `app.py`'s `@app.after_request` hook (M9.1), which is why all blueprints inherit it for free.

THE PACKAGES. All stdlib: `sqlite3`, `hashlib`, `uuid`, `time`, `json`. SQLite over a JSONL log because `recent()` wants ordered reads and the review layer already lives in SQLite; over a real database because there is exactly one writer process. No ORM — thirteen columns do not need SQLAlchemy.

TRY IT YOURSELF. Both ideas in one runnable file — append-only audit plus response stamping:

```
import sqlite3, hashlib, uuid, tempfile, os, time
from flask import Flask, jsonify

db = os.path.join(tempfile.mkdtemp(), "audit.db")

def log(job_id, stage, **kw):
    c = sqlite3.connect(db)
    c.execute("CREATE TABLE IF NOT EXISTS runs(ts TEXT, job_id TEXT, stage TEXT, model TEXT)")
    c.execute("INSERT INTO runs VALUES(?,?,?,?)",
              (time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime()),
               job_id, stage, kw.get("model", "")))
    c.commit(); c.close()

app = Flask(__name__)

@app.after_request
def stamp(resp):
    resp.headers["X-Response-ID"] = "ccx-" + uuid.uuid4().hex[:20]
    return resp

@app.route("/api/execute")
def execute():
    log("demo-ae", "execute", model="deepseek/deepseek-v4-pro")
    return jsonify({"ok": True, "domain": "AE"})

r = app.test_client().get("/api/execute")
assert r.headers["X-Response-ID"].startswith("ccx-")
rows = sqlite3.connect(db).execute("SELECT stage, model FROM runs").fetchall()
assert rows == [("execute", "deepseek/deepseek-v4-pro")]
print("stamped:", r.headers["X-Response-ID"], "| audited:", rows)
```

LIMITS. Stated plainly. (1) `log()` swallowing every exception means the audit trail fails silent: a locked database or full disk produces a pipeline that works perfectly and records nothing — for a Part 11-direction artifact that is the wrong failure mode, and nothing currently detects it. (2) The watermark tag is weaker than its docstring's "tamper-evident" suggests: `watermark_tag()` hashes `seed + time.time()`, so it is not reproducible — given a leaked artifact you can read its rid and tag but cannot recompute or verify the tag; it is a unique marker, not a MAC. A real tamper-evident tag would be `hmac.new(server_secret, rid)`, verifiable later. (3) Response IDs are generated but not persisted anywhere — the audit DB does not store the rid, so the two layers cannot be joined after the fact. All three are code-reading findings, not incidents.

HOW TO IMPROVE IT. Route `log()` failures to `log.exception` at minimum, and have `/api/health` report the last successful audit write. Replace `watermark_tag` with an HMAC keyed by an env secret. Add `rid` as a column so a leaked banner joins back to the run row.

LIFT AND REUSE. `audit.py` is already the generic artifact — the only SDTM-specific thing in it is the column list. Lift interface: `log(run_id, stage, **fields)` + `recent(n)` over one SQLite table, `CREATE IF NOT EXISTS` inline, callers never crash on audit failure (but do record that it failed). The after-request stamp is two lines in any Flask app.

M9.5 `/root/sdtm_review/scripts/sec_scan.py` — the security gate, and the posture it guards

Before a release is promoted, something has to answer "is there a hard-coded key in this tree, and does any dependency carry a known CVE?" — and answer it the same way every time, offline, in CI. That is this script: 166 lines, three checks, one verdict line: `SEC_SCAN: PASS` or `SEC_SCAN: FAIL`.

WHAT IT DOES. Runs three checks over the repo: `bandit` (static analysis of the app's own Python), `pip-audit` (known vulnerabilities in declared dependencies), and a regex sweep for hard-coded secrets. Fails on High/Critical findings, on any secret hit — and, unusually and correctly, on a missing scanner: "a gate that cannot test must not pass."

WHY IT EXISTS. These repos call paid LLM APIs, so live keys exist near the code; both repos have git-tracked `.env` files in their history of mistakes (the mapper's old unit leaked `ZAI_API_KEY` in a mode-644 git-tracked service file — the review unit's comments say so explicitly). The gate makes "no secrets in the tree" a machine-checked invariant instead of a hope.

HOW IT WORKS. `find_secret_hits()` walks the tree, skipping `SKIP_DIRS` (`.git`, `output`, `data`, `logs`…) and testing each source line against seven compiled patterns — the generic `api_key = "..."` shape, PEM headers, AWS `AKIA...`, `sk-...`, `ghp_...`, bearer tokens. The bandit arm is the most instructive part: it parses bandit's JSON and gates on severity counts, because the previous version substring-matched `"High:"` in text output — which also matches `"High: 0"` in the metrics footer, so the arm could never pass regardless of the code. `run_tool()` returns stdout and stderr separately for the same class of reason: bandit's deprecation warnings on stderr were corrupting the JSON when merged. `BANDIT_EXCLUDE` skips `./output` and `./quarantine` — LLM-generated per-request artifacts that no commit could ever fix — while deliberately scanning `golden/` because that code ships. pip-audit runs twice: gating on `requirements.txt` (the app's own declared deps), informational on the system interpreter, whose CVEs (in packages the app neither imports nor ships) are remediated by the pinned customer container instead.

THE SECURITY POSTURE IT SITS INSIDE — TRUTHFULLY. What I verified on this host today:

- Network: all five services bind `127.0.0.1` only (`ss -tlnp` confirms 8812, 8813, 9101, 9104, 9113). Public reach is via nginx: `/etc/nginx/sites-enabled/clincoder_root` proxies `clincoder.cloud/sdtm-mapper/` → 8812 and `/sdtm-review/` → 8813. The two `-test` ports and 9104 have no nginx stanza in that file — loopback only.
- Auth on the mapper (8812): **effectively none.** The `apikey.gate()` hook protects six routes only when `API_KEY_ENFORCE=1`, and the env file does not set it (verified: the variable is absent from `/etc/clincoder/sdtm_mapper.env`). So `/api/profile` through `/api/auto` are open on the public URL; what limits abuse is the model-spend allowlist inside `/api/auto` and the LLM spend guard (another module's territory).
- Auth on the review app (8813): **real, and on.** `GET /api/ta/whoami` on the live service returned `{"attributable": false, "open_reads": false, "reviewer": null}` — reads and writes both require an issued key. Keys are minted by `scripts/issue_reviewer_key.py`, stored chmod-600, revocable by flipping a JSON flag.
- One cross-repo coupling worth knowing: `services/apikey.py` in the **review** repo defaults `KEYS_FILE` to `/root/sdtm_mapper/api_keys.json` — the other repo's key store — and `/root/sdtm_review/api_keys.json` does not exist. The two apps therefore share one credential file unless `SDTM_KEYS_FILE` overrides it.
- Secrets: runtime secrets live in `/etc/clincoder/sdtm_mapper.env` (mode 600, root-owned, verified with `stat`), loaded via systemd `EnvironmentFile=`, holding key names like `ZAI_API_KEY`, `OR_TG_TOKEN`, `SDTM_REVIEWER` (values redacted here). The exception is the staging unit — see M9.6.
- What public exposure would mean: for the mapper, an anonymous visitor can consume LLM quota and disk via the open pipeline routes, and fetch any job's artifacts by knowing a 12-hex jid; the review app's specs and ledger — the actual product output — are key-gated.

THE PACKAGES. `subprocess`, `re`, `json`, `os`, `sys` — stdlib only; the scanners themselves (`bandit`, `pip-audit`) are external CLIs invoked as list-argv (no shell). That is the right shape: the gate must not import the app it judges.

TRY IT YOURSELF. The secret-grep arm, miniaturized and self-checking:

```
import re

PATTERNS = [
    re.compile(r'(?i)(password|secret|api[_-]?key|auth[_-]?token)\s*[:=]\s*["\'][^"\']{6,}["\']'),
    re.compile(r'sk-[A-Za-z0-9]{20,}'),
    re.compile(r'-----BEGIN (RSA|EC|OPENSSH|PRIVATE) KEY-----'),
]

def scan(text, name):
    hits = []
    for i, line in enumerate(text.splitlines(), 1):
        for pat in PATTERNS:
            if pat.search(line):
                hits.append(f"{name}:{i}: {line.strip()[:60]}")
    return hits

clean = 'KEY = os.environ["ZAI_API_KEY"]  # resolved at runtime, not stored'
dirty = 'api_key = "sk-aaaaaaaaaaaaaaaaaaaaaaaa"'
assert scan(clean, "app.py") == []
assert len(scan(dirty, "config.py")) >= 1
print("gate verdict:", "FAIL" if scan(dirty, "config.py") else "PASS")
```

LIMITS. The gate's blind spot is exactly where this host's one real finding lives: it scans the repo tree, and `/etc/systemd/system/` is outside it — so the inline API key in `sdtm-mapper-test.service` (M9.6) is invisible to `sec_scan.py` even though `.service` is in its `SOURCE_EXTS`. Regex secret detection also misses anything base64-wrapped or split across lines, and `bandit` severity gating at High/Critical means Medium findings accumulate unexamined. The gate is also opt-in: nothing in the systemd or promote path forces it to run before a deploy — that discipline lives in the CI scripts (`ci_gate.py` and friends), which are another module's story.

HOW TO IMPROVE IT. Add `/etc/systemd/system/sdtm-*.service` (readable as root) to the scan targets — the one place a secret actually leaked to. Print a count of Medium findings so drift is visible. Wire the script into `promote.sh` so a FAIL blocks promotion mechanically.

LIFT AND REUSE. The whole script is app-agnostic already — the only project-specific lines are `BANDIT_EXCLUDE` and `SKIP_DIRS`. The two lift-worthy principles: a missing scanner is a failing gate, and parse scanner reports as data, never grep their prose.

M9.6 `/etc/systemd/system/sdtm-review.service` — four units keep the fleet alive across reboots

The box reboots at 3 a.m. for a kernel update. No one is logged in. By 3:01 both apps are serving again, on the right ports, with the right secrets loaded — because four small text files told systemd exactly what to run. In SAS-server terms, systemd is the scheduler, the autoexec, and the operator who restarts a hung session, in one.

WHAT IT DOES. Four unit files (all verified today with `systemctl cat` and `systemctl list-units 'sdtm-*'`; a fifth pair, `sdtm-agent`/`sdtm-agent-test` on 9104, belongs to another codebase) define the production and staging services:

Unit	WorkingDirectory	Port	Restart	Secrets
`sdtm-mapper.service`	`/srv/prod/sdtm_mapper/current` (release symlink)	8812	`always`, 3 s	`EnvironmentFile=/etc/clincoder/sdtm_mapper.env`
`sdtm-review.service`	`/root/sdtm_review` (the git tree)	8813	`always`, 3 s	same EnvironmentFile
`sdtm-mapper-test.service`	`/root/clinical_apps_test/sdtm_mapper`	9101	`on-failure`, 5 s	**inline** `Environment=ZAI_API_KEY=408d...nVi7` (redacted)
`sdtm-review-test.service`	`/root/clinical_apps_test/sdtm_review`	9113	`always`, 3 s	same EnvironmentFile

All four run `Type=simple`, `ExecStart=/usr/bin/python3 <tree>/app.py`, `WantedBy=multi-user.target` (that last line is what "survives a reboot" means — the unit is pulled in when the machine reaches normal multi-user operation). All five listeners were confirmed live with `ss -tlnp` and answered `/api/health` today.

WHY IT EXISTS. `Restart=always` + `RestartSec=3` is the difference between "the app crashed at 3 a.m." being an incident and being a three-second blip. The port assignment per unit (`Environment=SDTM_PORT=8812` etc.) is why one codebase can run four times on one host. And `EnvironmentFile=` is the secrets story: the unit comments themselves record the incident that shaped it — the old mapper unit inlined the GLM key in a mode-644 file that `systemctl cat` would show to any user; the fix moved secrets to `/etc/clincoder/sdtm_mapper.env`, mode 600, root-owned. The staging mapper unit still carries the old mistake — a real, live API key inline in a world-readable unit file. That is a finding, not a design.

INPUTS AND OUTPUTS. Input: the unit file text plus the env file's sixteen variables (`ZAI_API_KEY`, `SDTM_CC_*`, `SDTM_MODEL_*`, `OR_ALLOWED_MODELS`, `OR_MAX_CALL_USD`, `OR_TG_TOKEN`, `SDTM_REVIEWER`, `SDTM_MAPPER_LLM`, … — names verified, values redacted). Output: a running process and logs. Note where the logs actually go: every unit except `sdtm-mapper-test` sets `StandardOutput=append:<tree>/logs/service.log`, so `journalctl -u sdtm-review` shows only start/stop events — the request log (the lines Flask prints per hit) lives in `/root/sdtm_review/logs/service.log`, and prod-mapper's in `/srv/prod/sdtm_mapper/shared/logs/service.log`, deliberately in `shared/` so it survives release swaps. Both claims verified by reading the logs today.

HOW IT WORKS — TWO DEPLOYMENT MODELS SIDE BY SIDE. The two production units embody opposite philosophies, and the comments in the units say so out loud. The mapper serves from a promoted release:

```
# Runs from the promoted release symlink - NEVER the git working tree.
# Before 2026-07-17 this pointed at /root/sdtm_mapper, which meant every file
# saved during development was instantly live on a public URL.
WorkingDirectory=/srv/prod/sdtm_mapper/current
```

`current` is a symlink (today: `releases/20260728_013025_3f312c2`), swapped atomically by `promote.sh` (on this host: `/root/deploy_kit/promote.sh`) — editing `/root/sdtm_mapper` can no longer change production. The review service runs straight from the git tree: fast iteration, at the price `routes_ta.py`'s docstring states — anything committed is one restart away from live. **The staging**

mirror pattern, as it actually is on this host: the `-test` units do not share the prod trees. They run separate copies under `/root/clinical_apps_test/` (a directory holding test copies of fifteen apps), and those copies have drifted — `diff` shows staging `app.py` differs from both `/root/sdtm_mapper/app.py` and `/root/sdtm_review/app.py`. So "staging mirrors" is aspiration; "older parallel copies on loopback-only ports" is fact. They are useful as crash-test dummies (`on-failure` restart on the mapper-test lets a genuinely broken build stay down where you can see it), but a green test on 9113 does not prove the tree behind 8813.

THE AE JOURNEY, END TO END ACROSS THIS LAYER. A user maps and reviews the synthetic AE domain like this, in order: (1) browser → `https://clincoder.cloud/sdtm-mapper/` → nginx → 8812; `POST /api/profile` with `ae_raw.csv`, then `/api/spec`, `/api/code`, `/api/execute`, `/api/validate`, `GET /api/download/<jid>/ae.xpt` (M9.1) — every stage audit-logged (M9.4), every response watermarked. (2) Offline, an operator builds review artifacts on the box (`scripts/build_ta_programs.py --domain AE`, `build_ta_verified.py`), because page loads never generate. (3) Reviewers → `https://clincoder.cloud/sdtm-review/` → 8813 with an issued key: `GET /ta`, `/api/ta/pack?ta=Cardiology`, `/api/ta/raw?ta=Cardiology&domain=AE`, `/api/ta/programs`, `/api/ta/output`, then `POST /api/ta/comment` and `/api/ta/propose` against `AETERM`, `POST /api/ta/resolve` to adjudicate, and `GET /api/ta/export/specs` for the signed-off workbook (M9.3). Two services, one artifact trail.

THE PACKAGES. None — this is the layer where the platform replaces code, the highest rung there is. systemd gives you the restart loop you would otherwise write in a shell `while true`, log capture, boot ordering (`After=network.target`), and environment injection. The alternatives (supervisord, Docker) add a dependency to solve problems this host does not have.

TRY IT YOURSELF. No systemd needed — the one piece worth internalizing is how `EnvironmentFile=` + `Environment=` reach your code as plain `os.environ`. This mirrors the exact contract the four units rely on:

```
import os, tempfile

envfile = tempfile.NamedTemporaryFile("w", suffix=".env", delete=False)
envfile.write("ZAI_API_KEY=408d...redacted\nSDTM_REVIEWER=bhanuji.d@gmail.com\n")
envfile.close()

def load_environment_file(path):
    """What systemd's EnvironmentFile= does, minus quoting edge cases."""
    out = {}
    for line in open(path):
        line = line.strip()
        if line and not line.startswith("#") and "=" in line:
            k, v = line.split("=", 1)
            out[k] = v
    return out

os.environ.update(load_environment_file(envfile.name))
os.environ["SDTM_PORT"] = "8813"          # the unit's Environment= line
port = int(os.environ.get("SDTM_PORT", "8813"))   # app.py's last line, verbatim logic
assert port == 8813 and os.environ["SDTM_REVIEWER"].endswith("@gmail.com")
print(f"app would bind 127.0.0.1:{port} with {len(load_environment_file(envfile.name))} injected
vars")
```

LIMITS. `Restart=always` masks crash loops: a boot-time exception means a restart every 3 seconds forever, and with stdout appended to a file rather than the journal, `systemctl status` shows "active (running)" moments after each restart — you must read `service.log` to notice. Everything runs as root (`User=root` explicit on mapper-test, implicit elsewhere); a compromise of the public Flask process is a

compromise of the box. In-memory `JOBS` (M9.1) means every `Restart` also silently ends all user sessions. The staging drift is unmeasured — nobody can currently say how far behind the test copies are.

HOW TO IMPROVE IT. Add `User=clincoder` with a dedicated account and directory permissions; add `StartLimitBurst` / `StartLimitIntervalSec` so a crash loop becomes a visible failed unit; replace the review service's git-tree serving with the mapper's promote pattern (the mechanism already exists on the box); rebuild the `-test` trees from prod releases on a schedule so "staging" regains meaning; move the mapper-test key into the 600 env file — a one-line fix to the only cleartext secret found in this whole layer.

LIFT AND REUSE. The unit file is the reusable artifact — the prod-mapper one is a near-perfect template: release-symlink WorkingDirectory, port via `Environment=`, secrets via root-600 `EnvironmentFile=`, logs to a `shared/` path that outlives releases, `Restart=always`. Copy it, change four paths, and any Flask app on any Linux box inherits the whole ops story.

What this module still owes you

Honest gaps, so you know what you have and have not been shown. **UNREAD INTERNALS:** this module traced the web layer's calls into `ta_pack`, `ta_review`, `mapping_review`, `core_engine` and friends but did not read those services — their data formats (the pack dict, the ledger schema, `plan.json`) are asserted here only as the routes see them. **The other three blueprints:** `routes_review.py`, `routes_mapping.py` and `routes_study.py` (10, 11 and 11 handlers) got one paragraph of role each, not a chapter; `routes_mapping`'s docstring alone — the withdrawn-and-reinstated security story — is worth your own read. **Unmeasured claims:** every latency number quoted (CORE ~66 s, rebuild ~6 s, validate_core 10–25 s) comes from the code's own docstrings, not from measurements taken for this module; the four-reviewers-polling-every-3s load figure likewise. **Not exercised:** no write endpoint was called against the live services (this module held a read-only line — no comments posted, no rebuilds triggered, nothing restarted), so the write-path behavior is verified from code, not from requests. **nginx:** only the two `location` stanzas were read, not the TLS, rate-limiting or header configuration around them — the true public attack surface is therefore only partially characterized. **And the fork question** — whether mapper and review should reconverge on a shared package — is raised twice here and answered nowhere; that is a decision with a budget, not a chapter.

What this book does not cover

- **The frontend.** Templates, JavaScript and CSS under either repo are out of scope by design — the premise of the book is that the UI is the part that will be rewritten first.
- **Deployment beyond this host.** nginx, TLS and DNS for the public services are covered only where a module touches them; the full ops runbook lives outside this book.
- **The other ClinCoder tools.** The Validator wrapper, TLF Studio, Define-Maker and ADaM Agent are later volumes in this series. Where a module recurs across tools, the later volume documents only the delta.
- **Model quality itself.** This book documents how the AI layer is built and gated, not how good any particular model is at SDTM mapping — those numbers live in the evidence directories the gates write, and they change with every model swap.

Colophon

Written by module-scoped agents reading the source tree directly, then gated: every fenced Python block parsed and its imports resolved against the installed interpreter, every backticked path and service name tested against the host, and a cross-module consistency review before assembly. Built with pandoc and WeasyPrint on VPS2. Basis commits are printed on the cover and in the front matter.