PharmaSUG China 2026 - Paper AA-218

A Dual-Harness Workflow to Wrap AI into Observable, Reusable Clinical Table Delivery: From SAP Intent to Shell-Aligned Execution

Wenjun Zhu, Jiangsu Hengrui Pharmaceuticals Co., Ltd.

ABSTRACT

Statistical programming teams working in a single therapeutic area repeat a familiar pipeline for every study: read the Statistical Analysis Plan (SAP), pick and adapt table shells, bind them to Analysis Data Model (ADaM) variables, run analyses, and produce accurate results. Modern coding agents (Claude Code, GitHub Copilot, Codex, etc.) can also cope with these routines and are multipotent—they generate code, execute it, edit files, and increasingly assemble deliverables end-to-end inside a single session. What they do not, by themselves, provide is the domain validation gates, audit trails, and institutional memory that regulated table, listing, and figure (TLF) production requires.

This paper treats that gap as a control problem rather than a prompting problem. A large language model is a powerful but stochastic actuator; a regulated pipeline demands deterministic, explainable outcomes. The workflow presented here reconciles the two by wrapping the model in a harness—a feedback controller that supplies the reference signal (the sponsor's table shell, treated as the executable spec of the analysis), the sensors (deterministic validation gates), the comparator (shell-aligned checks and human confirm/reject decisions), and the recorded state (an append-only store in which every confirm, reject, and patch becomes structured data). Within this loop, evaluation separates into two arms—semantic SAP-to-shell selection and deterministic shell-aligned execution—connected by a typed `spec[]` handoff placed exactly where statistical programmers have always handed work off to one another, so the boundary that made human review accountable makes agent-assisted review accountable as well.

Two properties of this design travel beyond the specific system. First, the control discipline is model-agnostic: models will keep getting stronger, coding agents and IDEs will keep advancing, and the AI landscape may look different every few months, but wrapping a stochastic component in an explicit loop of reference, measurement, feedback, and recorded state is a strategy worth borrowing wherever non-deterministic AI must operate inside a regulated pipeline. Second, the loop's by-product can outvalue its immediate output: because every review decision passes through a gate, daily operations leave behind not chat transcripts but a compounding, attributable record of judgment. The workflow already runs on real oncology SAP and shell-library material, and the pattern is adoptable today—auditability arrives immediately; institutional memory accrues with use.

1. INTRODUCTION

Clinical table delivery is a multi-step pipeline, not a one-shot generation task. It spans SAP interpretation, shell adaptation, ADaM variable binding, execution, and QC. Success is measured by accuracy under human and statistical-programmer review—and even where AI takes over parts of the pipeline, the same observability, evaluability, and reusability requirements apply.

Stated in control-engineering terms, the tension is simple. The pipeline's newest component—the large language model—is a stochastic actuator, while the pipeline's acceptance criteria are deterministic; no amount of prompting removes that mismatch, because it is a property of the component, not of its instructions. The classical engineering response to a useful-but-unreliable component is not to discard it but to close a loop around it: give it a reference to track, measure its output against that reference, feed the error back, and record every state transition. “Wrapping AI in a harness” is that response applied to TLF production. How to build such a loop—observable, evaluable, reusable—is the main question this paper explores.

1.1 THE PIPELINE SPLITS BEFORE ARCHITECTURE

AI works the same way humans work—from the perspectives of cognition and task decomposition. The familiar TLF pipeline naturally splits into two loops with different convergence goals:

Part A — From SAP to selected shells

- A.1 Read the SAP, especially analysis requirements, and decide which table shells apply.
- A.2 Adapt shell templates to study-specific populations, arms, and endpoints.

Part B — From selected shells to validated tables

- B.1 Bind each report to ADaM variables and analysis logic.
- B.2 Run analyses and compare output to shells and regulatory expectations.
- B.3 Revise specifications (and sometimes algorithms) until tables meet analysis requirements.

Part A hinges on semantic plausibility judged by experts; Part B hinges on structural and numeric validation against sponsor shells. Treating the entire pipeline as a single large-language-model (LLM) mapping task collapses incompatible evaluators and produces chat transcripts instead of auditable artifacts. (Figure 1. The TLF pipeline's natural split—Part A (semantic shell selection) and Part B (structural + numeric validation), connected by the `spec[]` handoff.)

Figure 1. The TLF pipeline splits into two convergence loops

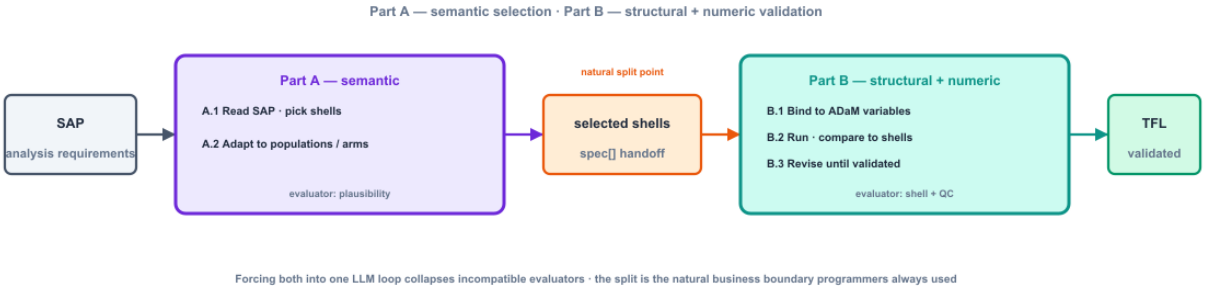

Figure 1. The TLF pipeline's natural split—Part A (semantic shell selection) and Part B (structural + numeric validation), connected by the `spec[]` handoff.

Beyond the pipeline's structural split, a second problem surfaces in everyday work: knowledge that should accumulate across studies but does not. Statistical programmers in a single therapeutic area and compound family encounter recurring, granular questions that people have always recorded in personal notes, team wikis, and email threads:

- **A baseline table needs a non-standard gene-stratification row** that was solved once in an earlier study.
- **The same compound program adds an analogous non-standard table** across Phase II and Phase III protocols.
- **A shell revision discussed in Study A reappears in Study B**—*who decided the fix, and what was the approved pattern?*

AI can now summarize these scattered records, but the loop—AI summarizes, a human reviews, the human revises—is still not smooth. The records stay scattered; the revisions stay manual. A workflow that wants to be useful across studies must therefore carry memory, not just conversation. Underneath sits a quieter observation: every one of these recurring questions was settled, at some point, by a person exercising judgment—and the judgment, not the conversation in which it happened, is the asset worth keeping. A design goal of this work, developed in Section 5, is to capture each such decision as a structured, attributable record, so that review effort compounds across studies instead of evaporating with the session.

1.2 CODING AGENTS ARE MULTIPOTENT—AND WHAT A HARNESS ADDS FOR TLFS

Modern coding agents are multipotent. They generate SAS/R/Python, execute it, edit files across a repository, run tests, and increasingly assemble deliverables end-to-end inside a single session. For an individual table, a capable coding agent can already do a great deal of the work. What TLF production additionally requires is **operational closure around the artifact lifecycle**: domain validators (spec versus shell gold index, preflight rules, deterministic run QC), append-only audit (snapshots, confirm/reject reasons, replayable validation logs), and per-table QC (quality checking) telemetry. Coding agents provide process inspection, code review, and git history—useful, but not the same artifact lifecycle.

These tools excel at **exploratory programming** and **prototype velocity**, and for pipeline-style batch work they can also save tokens by keeping the entire loop—generation, execution, inspection, revision—inside one tool environment rather than switching between services. They do not, by themselves, encode session-isolated spec drafts, quality-checking evidence, shell cross-functional review, or institutional memory distilled from production review. The harness described here adds those layers without replacing coding agents for ad-hoc scripting.

1.3 WHY A WEB PLATFORM INSTEAD OF A DESKTOP APP

A domain-specialized desktop agent can inject business context per user and may feel stronger for a single session. Regulated table workflows, however, require **organizational** properties:

- **Collaboration:** the same SAP job, spec basket, and shell review queue can be handed off across programmers without re-explaining decisions in chat.
- **Auditability:** append-only stores record snapshots, confirm/reject events, and validation logs for full lifecycle replay.
- **Shared memory:** layered memory lets a prior study's shell fix become the next project's default starting point.
- **Central model routing:** backend configuration keeps model roles behind one service boundary, on-premise when required.

Copilot remains excellent for exploratory scripts; regulated tables belong on a harness platform with a control plane, audit, and shared memory.

1.4 CONTRIBUTIONS

This paper addresses the problems above through four contributions:

1. **A control-engineering harness pattern for regulated AI adoption.** One harness pattern, two evaluator arms, one shared control plane: the design closes a feedback loop around a stochastic model—reference (the shell-derived spec), measurement (deterministic gates), correction (human-labeled retry), and state (append-only audit)—with a global / project / table memory stack extending the loop across studies. The pattern is deliberately model-agnostic: a reusable discipline for making non-deterministic AI usable in a regulated pipeline, transferable beyond this system.
2. **The shell-as-spec principle.** Everything upstream of a TLF is ultimately in service of the analysis, and the table shell is the sponsor-facing spec of that analysis. Taking this literally turns table generation from prompt-driven to spec-driven: the unit of AI output becomes a typed, diffable, replayable artifact rather than prose.
3. **A typed `spec[]` handoff placed at the human handoff point.** The machine's split coincides with the boundary statistical programmers have always used—delivering selected shells between semantic selection and numeric execution. Because the two workflows are isomorphic at this boundary, the accountability structure of human review transfers intact to agent-assisted review, and semantic and numeric evaluation stop contaminating each other.
4. **Memory as a compounding record of judgment, reported candidly.** Append-only audit plus layered memory turn daily confirm, reject, and patch decisions into institutional capital that the next study inherits. We separate what runs today from what is still landing, and keep the self-growth direction explicitly staged behind the evidence the harness accumulates.

2. SYSTEM CONTEXT AND DESIGN GOALS

2.1 THE SHELL IS THE SPEC OF THE ANALYSIS

Everything upstream of a TLF—the protocol, the SAP, the data collection plan, the shell library—is, in the end, in service of analysis. The table shell is the spec of that analysis: the sponsor-facing statement of what rows, columns, statistics, and populations the analysis must produce. This is more than an encoding convenience; it is the observation on which the rest of the architecture stands, and three consequences follow. First, analysis becomes spec-driven rather than prompt-driven: the unit of AI output is a typed, diffable, versionable artifact, so review means inspecting a structured diff against a known layout rather than re-reading a conversation, and rerunning means re-executing a snapshot rather than re-rolling a generation. Second, the spec becomes the natural intermediate review point between SAP interpretation and numeric execution—simultaneously an upstream deliverable (what the SAP arm hands off) and a downstream constraint (what the execution arm must satisfy). Third, in control terms, the spec is the reference signal: the explicit target that downstream validators compare output against—without it, there is nothing to close the loop on. (Figure 2. Typed `spec[]` sits between Part A (shell selection) and Part B (numeric execution)—upstream deliverable and downstream constraint.)

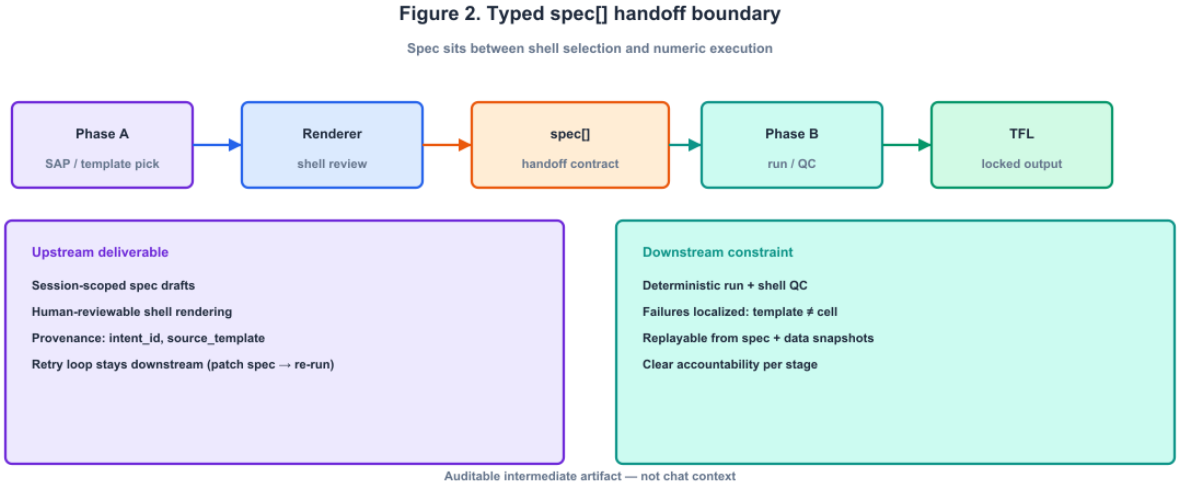

Figure 2. Typed `spec[]` sits between Part A (shell selection) and Part B (numeric execution)—upstream deliverable and downstream constraint.

2.2 TARGET WORKFLOWS, REQUIREMENTS, AND NON-GOALS

The platform targets TLF workflows in which shell-spec logic is reused across studies and therapeutic areas and repeatedly adapted to new protocol contexts. In practice the shell library carries the domain density of oncology reporting—hierarchical adverse-event summaries and their treatment-related mirrors, graded-toxicity views, exposure and laboratory displays, oncology-specific efficacy tables—so what the workflow automates is the reuse of genuinely domain-specific structure, not generic tabulation. Before the architecture is laid out, two practical requirements shape the design.

Determinism and traceability. Given the same spec and data snapshot, reruns must produce identical output, and every patch, reject reason, and promotion event must be recoverable after the fact. In regulatory submissions, the ability to explain *how* a table was produced is as important as the table itself.

Scoped AI flexibility. LLM assistance must be both recordable and flexible enough not to suppress the agent's autonomous capability—just as organizational design should not suppress individual employee capability. The engineering balance is struck at the surface level: each UI surface binds the agent to one evaluator flow and one bounded tool set, so scope is enforced structurally rather than by prompt; within that scope, the agent is free to choose its tools, sequence its actions, and propose edits. Any stage can

disable its agent without affecting the rest of the workflow, so teams can turn off the agent for SAP matching while keeping it active for spec review, or vice versa.

These two requirements also imply clear non-goals for the current stage. The system does not attempt fully autonomous SAP-to-final-table generation, does not replace deterministic statistical execution with opaque model inference, and does not collapse all orchestration into a single universal agent. It must also deliver useful outcomes before any model fine-tuning takes place—if value depends on a future training run, adoption stalls. Future stages may absorb more autonomy as production experience accumulates—and may eventually form a self-improvement loop in which the agent learns from its own confirmed and rejected iterations. At the current stage, however, the goal is more modest: grasp AI capability, teach the agent to work the way humans work with the same memory humans rely on, and be more useful than a human doing the same task.

3. ARCHITECTURE: ONE PATTERN, DUAL EVALUATORS

3.1 THE HARNESS PATTERN

A harness is defined here as an explicit lifecycle with stage boundaries, deterministic checks, human decision gates, and append-only state capture. In control terms, it is the closed loop around the stochastic actuator: the shell-derived spec supplies the reference; deterministic validators are the sensors; the diff against the shell gold index and the human confirm/reject gate form the comparator; bounded, labeled retry is the corrective action; and the append-only store is the recorded state. The agent is a scoped worker inside this lifecycle, not the source of final truth. Both workflow arms follow the same convergence pattern (Figure 3. Convergence state machine—both arms share the pending → auto_check → human_review → confirmed | rejected loop.):

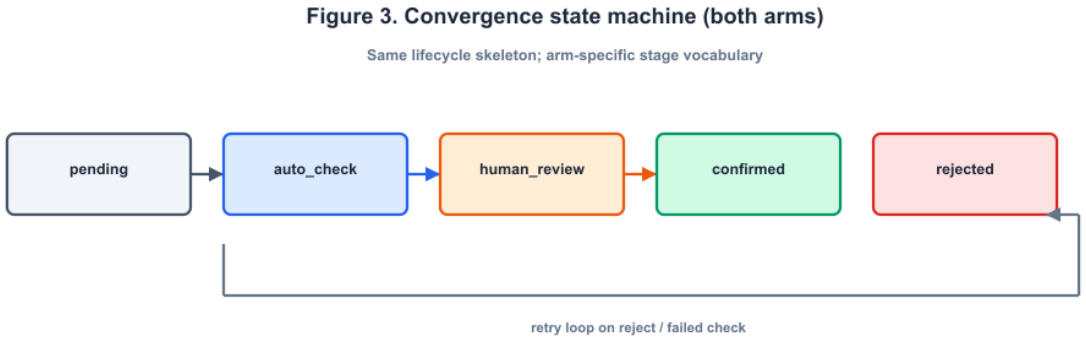

Figure 3. Convergence state machine—both arms share the pending → auto_check → human_review → confirmed | rejected loop.

The vocabulary differs across arms—one tracks job stages, the other tracks per-spec review status—but the loop is the same: automated gates, human review, labeled retry. An agent without a harness produces suggestions. An agent inside a harness produces **labeled iterations**—each confirm, reject, or patch becomes data. In loop terms: without the harness, the error signal escapes as chat; inside it, the error signal is captured, attributed, and reused.

3.2 TWO ARMS WITH SEPARATE EVALUATORS

The first arm processes SAP material into reviewable intent items: it converts SAP sections into structured chunks, retrieves candidate templates from the spec library, presents ranked candidates with supporting context, and lets the reviewer apply explicit actions—pick, skip, clone, or create. Its evaluator is **semantic**: embedding match quality, confidence thresholds, expert pick or skip.

The second arm consumes `spec[]` records and drives table output toward convergence: it enriches spec fields where deterministic rules permit, dispatches to the appropriate analyzer pattern via a registry, renders both placeholder and computed outputs for side-by-side comparison, runs shell-alignment and QC checks, and accepts structured patch, reject, or confirm decisions. Its evaluator is **structural and**

numeric: spec versus shell gold index, deterministic run QC, enrichment validation. (Figure 4. Semantic vs structural/numeric evaluators—different failure modes, separated by the `spec[]` split point.)

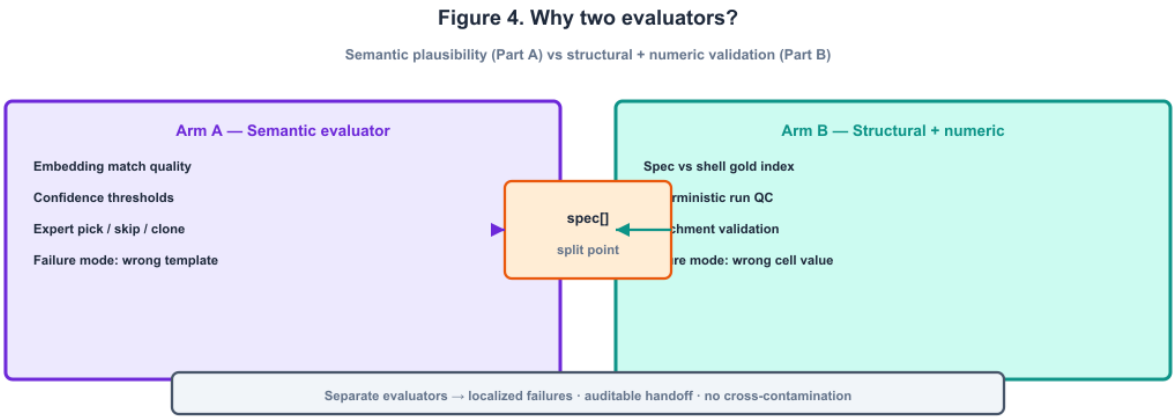

Figure 4. Semantic vs structural/numeric evaluators—different failure modes, separated by the `spec[]` split point.

Splitting evaluation is not a stylistic preference. The two phases require different evaluators, and forcing both into one loop collapses incompatible evaluators and makes a wrong template pick and a wrong cell value indistinguishable in the audit trail. These are workflow-structure problems, not model-quality problems.

3.3 ONE BOUNDARY, TWO WORKFLOWS: THE SPLIT IS ISOMORPHIC TO HUMAN PRACTICE

The place where the machine pipeline splits was not invented for the machine. The SAP-to-shell versus shell-to-output boundary is the split point traditional statistical-programming practice already uses: programmers have always handed off selected, adapted shells as the deliverable between interpreting the SAP and producing validated numbers. The harness draws its stage boundary on exactly that line, and the coincidence is the point—the machine's division of labor is isomorphic to the human one.

This isomorphism does real work. First, it inherits an accountability structure the profession has already settled: the artifact exchanged at the boundary is one reviewers know how to inspect, so agent-assisted review requires no new review theory—the same handoff that made human review accountable makes agent review accountable. Second, it makes failure attribution legible: when a table is wrong, the boundary tells you whether to question the interpretation or the execution, which is exactly the question a lead programmer would ask of a human team. Third, it keeps humans and agents interchangeable at the boundary: either phase can be performed by a person, by an agent, or by a person supervising an agent, without renegotiating the contract between the phases. A workflow that wants human oversight to stay meaningful should split where human judgment already knows how to attach; a workflow that splits anywhere else forces reviewers to learn the machine's seams instead of letting the machine inherit theirs.

3.4 THE `SPEC[]` HANDOFF BOUNDARY

The `spec[]` boundary is the central engineering contract. Each handoff item carries enough provenance for deterministic continuation:

```
{
  "table_id":"t-adae-teae-soc-pt-study042",
  "intent_id":"4.4.1.2-teae-soc-pt",
  "section_ref":"4.4.1.2",
  "action":"cloned_from_library",
  "source_template":"t-adae-teae-soc-pt",
  "session_path":"outputs/session_specs/{session_id}/..."
}
```

The boundary isolates semantic and numeric failures into separate evaluation contexts; it makes any run replayable from its spec snapshot, dataset snapshot, and analyzer version; and it lets promotion policies be enforced at the artifact level rather than depending on conversational context. The handoff is simultaneously an upstream deliverable (a typed list of session-scoped spec drafts ready for shell review) and a downstream prompt scaffold (compact structured context injected into the execution arm without pasting full JSON into every system prompt). (Figure 5. Arm A delivers typed `spec[]`; the same shell-container layout renders in FRAME mode for human shell review and in LIVE mode for locked TFL output.)

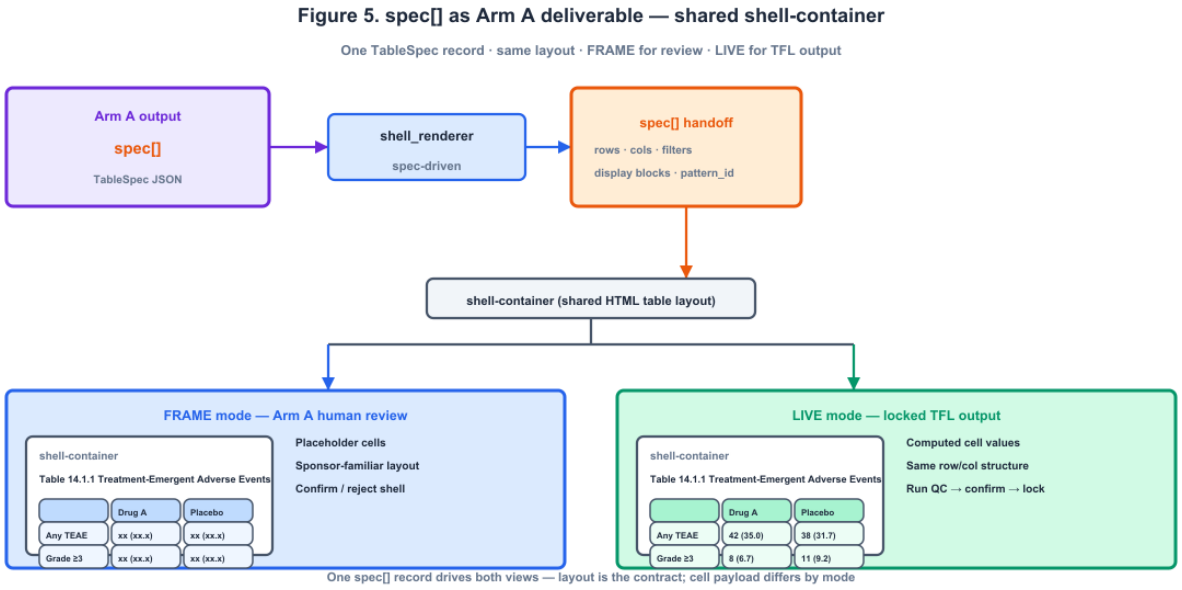

Figure 5. Arm A delivers typed `spec[]`; the same shell-container layout renders in FRAME mode for human shell review and in LIVE mode for locked TFL output.

3.5 SHARED CONTROL PLANE

Both arms run on the same control-plane infrastructure: session isolation, append-only review events, spec snapshots, run-level audit records, and training-data export paths. This shared layer yields a single audit story across the entire pipeline without forcing evaluator unification. Keeping the criteria separate while sharing the lifecycle skeleton is the core architectural choice.

4. AGENT ROLE IN REGULATED WORKFLOWS

The agent is not a single omniscient assistant but a set of stage-specific workers, each scoped to one phase of the harness lifecycle. In the SAP arm, the agent helps rank candidate shells, drafts clone operations, and explains why a template fits a section. In the execution arm, it proposes spec patches, diagnoses validation failures, and suggests the next review action. This stage-by-role division has three benefits: each agent carries only the context its stage needs, so the context window stays small and noise stays out of the audit trail; accountability maps cleanly to the stage that produced a decision; and any stage can disable its agent without affecting the rest of the workflow. In control terms, each surface closes its own small loop—one evaluator flow, one bounded tool set, one place for the error signal to land—rather than one large loop no reviewer can inspect.

The agent does not replace the traditional UI—it sits beside it. Each existing review surface (intent list, handoff basket, spec editor, shell comparison) keeps its structured controls; the agent panel is an optional side channel that can propose edits, summarize context, and pre-fill forms. Reviewers act on the structured surface; the agent accelerates the choke points. This keeps the harness usable even when the agent is off, and it keeps humans on sponsor-familiar screens rather than inside a chat window. (Figure 6.

Agent dialog running beside the traditional review surface—structured controls remain primary; the agent panel is an optional side channel scoped to that surface's evaluator flow and tool allowlist.)

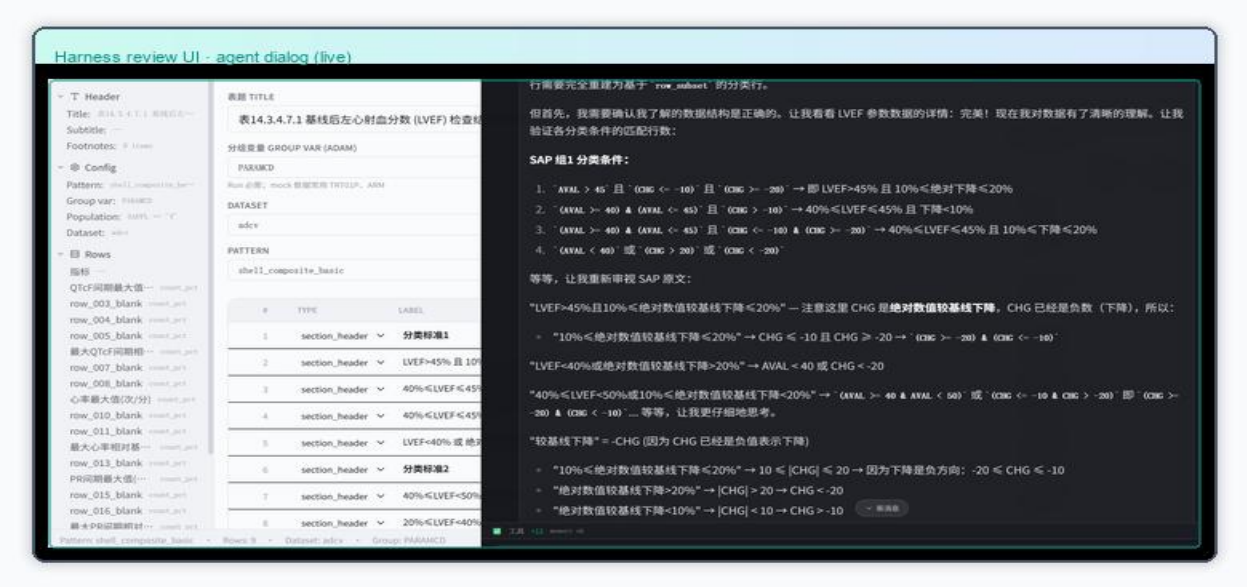

Figure 6. Agent dialog running beside the traditional review surface—structured controls remain primary; the agent panel is an optional side channel scoped to that surface's evaluator flow and tool allowlist.

Context injection follows the same stage scoping. A frozen memory block per session (global conventions, trial defaults, recent table-level notes) is built once for prefix-cache stability and invalidated lazily after writes. Tool responses carry a compact summary for the LLM context alongside a full payload for audit, so the agent sees enough to choose its next step without the full JSON flooding the prompt. The tool surface is the contract between the agent and the harness: same shape across both arms, different allowlists per surface. In its completed form, each tool represents one clear operation and carries enough neighboring context that the agent can choose its next step without excessive round-trips.

5. MEMORY, AUDIT, AND INSTITUTIONAL LEARNING

The observation planted in Section 1.1 matures here. In an established therapeutic area, the scarce resource is not table-generation capacity but settled judgment—which shell fits this endpoint, which fix was approved last study, which deviation was deliberate. Conventional practice produces that judgment continuously and stores it nowhere durable. The harness changes the economics of keeping it: because every confirm, reject, and patch must pass through a gate anyway, recording the decision as structured, attributable data costs nothing extra at review time—and what accumulates is not a log but a compounding record of judgment. The layers below serve that end from two directions: memory makes prior judgment retrievable at the point of decision; audit makes it defensible after the fact.

5.1 THREE-LAYER MEMORY

Layer	Scope	Purpose
Global	Cross-trial	Conventions shared across the therapeutic area
Project / trial	One trial identifier	Trial-specific defaults and prior fixes
Table / session	One spec or session	Per-spec distilled notes from recent review

Recall builds a frozen memory block per session for prefix-cache stability and invalidates lazily after writes. Memory content is fenced and sanitized against prompt injection. General coding agents rely on static project rules files or ephemeral chat—neither captures confirm/reject-distilled institutional knowledge indexed by table identifier. (Figure 7. Three-layer memory stack—cylindrical scopes (Global → Project → Table), merged into a frozen agent prefix per session with lazy invalidation after writes.)

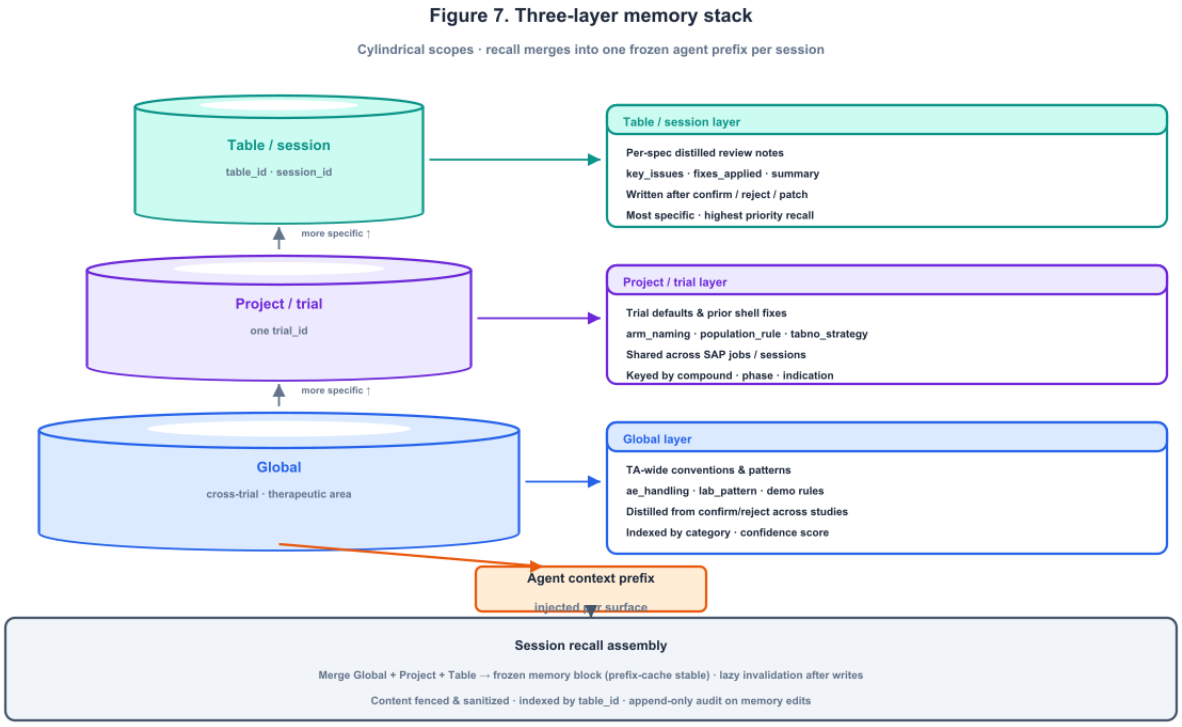

Figure 7. Three-layer memory stack—cylindrical scopes (Global → Project → Table), merged into a frozen agent prefix per session with lazy invalidation after writes.

5.2 APPEND-ONLY AUDIT AND TRAINING STORES

Audit is **append-only** on both arms: review events, spec snapshots, enrichment runs, agent conversations, convergence summaries, and validation logs are never deleted. The SAP arm additionally emits slice-level confirm/skip labels as training examples for future embedding reranker fine-tuning. What the store captures is decisions, not just outputs—who confirmed, what was rejected, and for what stated reason—precisely the fraction of institutional knowledge that personal notes and email threads have always lost.

The two arms currently use different persistence schemas—a pragmatic compromise, not a final design. A unified workflow state machine remains future work.

5.3 TRAINING FLYWHEEL

The upgrade path is deliberately staged (Figure 8. Training flywheel—Phase 1 accumulates production labels today; Phase 2 and 3 upgrade prompts and retrieval only after enough labeled evidence exists.). Phase 1 (current) relies on production review alone: every confirm, reject, patch, and SAP slice decision appends structured labels to the audit and training stores—no model change required. Phase 2 begins once rejection and patch reasons cluster, using those patterns to auto-select prompt templates while humans retain the confirm gate. Phase 3 fine-tunes the embedding reranker only after labeled slices accumulate across many SAP jobs, closing the confidence feedback loop on memory so improved retrieval feeds the next production cycle.

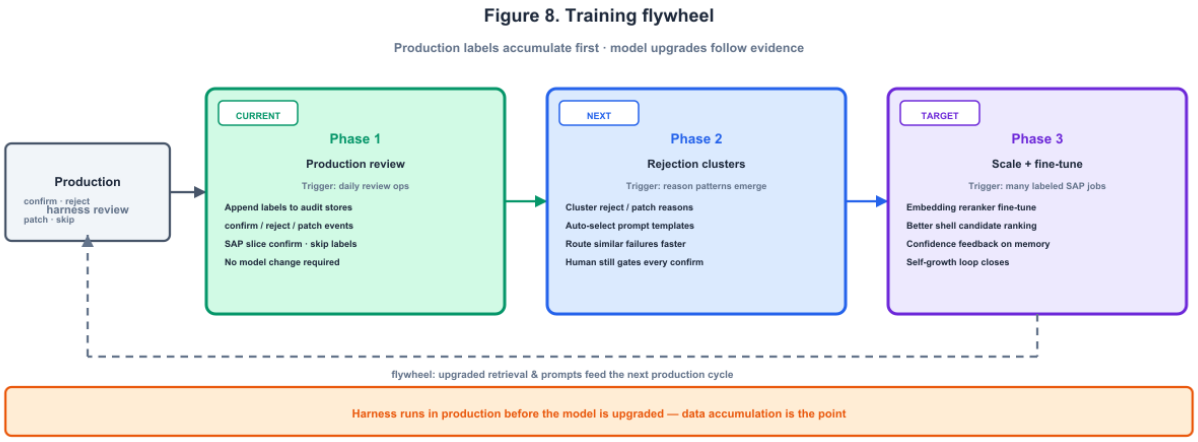

Figure 8. Training flywheel—Phase 1 accumulates production labels today; Phase 2 and 3 upgrade prompts and retrieval only after enough labeled evidence exists.

The harness **runs in production before the model is upgraded**—data accumulation is the point. The flywheel's fuel is the recorded judgment of the review process itself, not additional model capability.

6. SPEC, SHELL, AND ALGORITHM CO-EVOLUTION

The shell, the spec, and the analysis pattern are not independent artifacts—they co-evolve under harness pressure. A sponsor revises a shell; the parser updates the corresponding spec; enrichment maps new labels to ADaM variables; the spec runs against the registered pattern. When the pattern handles the change, the cycle closes at the spec layer. When it cannot—when specs repeatedly fail validation on a class of tables—the gap becomes the trigger for algorithm graduation.

The three layers move at different speeds. Shells change with protocol amendments; specs change every session; patterns change only when a documented gap accumulates enough evidence to justify a human-authored extension. This asymmetry is deliberate: patterns are the slowest-moving layer because they are the shared contract across many studies, and a premature pattern change ripples into every future spec.

The graduation mechanism is human-gated, not autonomous. A new row label in the shell triggers the parser to update the spec and enrichment to map it to an ADaM variable; a new statistic row adds a spec field, and if no registered pattern produces it, the gap is documented; a population arm change patches the spec and re-renders the shell; a shell template revision triggers re-parse, archive diff, batch enrichment, validation, and a review campaign. When validation failures cluster on a class of tables, a programmer implements the algorithm extension, registers the new pattern, and future specs reference the graduated pattern (Figure 9. Pattern graduation under harness pressure—spec changes validate against the registered pattern; repeated failures document a gap; a human extends the algorithm and registers the new pattern, which future specs reference.).

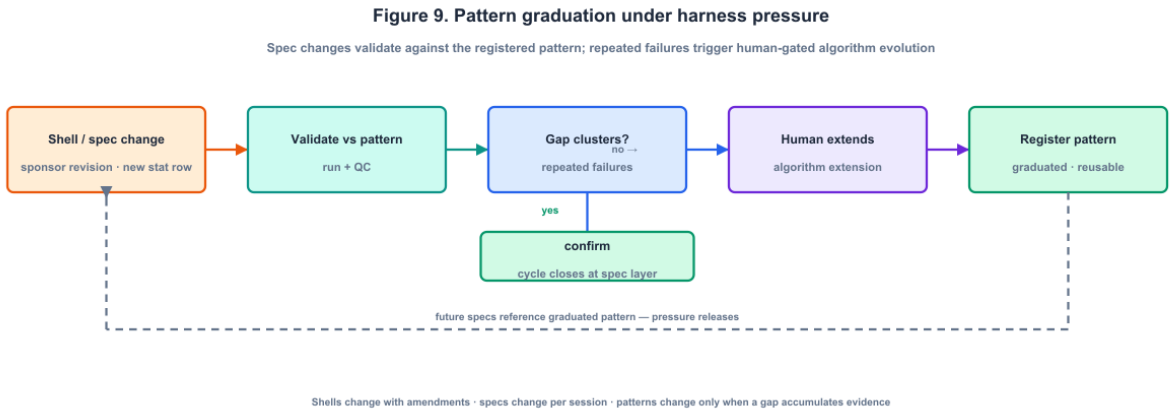

Figure 9. Pattern graduation under harness pressure—spec changes validate against the registered pattern; repeated failures document a gap; a human extends the algorithm and registers the new pattern, which future specs reference.

This is **algorithm evolution through harness pressure**—not autonomous code generation, and not prompt-time improvisation. The harness provides the pressure and the evidence; the human provides the code. The pattern registry grows only as fast as production review can justify, which is what makes it safe to reuse across studies. Because each graduation decision can propagate across future studies, this co-evolution loop must pass explicit governance gates before promotion.

7. GOVERNANCE AND TRUST BOUNDARIES

These controls are the release gates for the co-evolution loop in Section 6: they determine when a spec change remains session-local, when it can be promoted, and how every promotion stays inspectable and replayable.

Control	Purpose
Session-isolated writes	Drafts cannot corrupt the base library without an explicit promote
Archive-before-overwrite	Shell re-parse preserves prior spec versions
Expression safety	Population filters screened before evaluation
Rate limits	Per-session LLM caps prevent runaway loops
Validation gate	No confirm without passing automated QC (or explicit waiver)
Append-only audit	Deletion prohibited; inspection-ready history
Truth gate	Agent cannot silently dismiss diff warnings

The agent operates **inside** these walls regardless of prompt content. Together, these controls allow downstream inspection not only of *what output was produced* but also of *how the decision was made, by whom, and what alternatives were considered*.

8. DISCUSSION

8.1 DUAL EVALUATORS VS. UNIFIED "SAP-TO-TABLE" AGENT

A single agent spanning SAP prose and numeric output must carry incompatible context, use incompatible evaluators, and blur accountability. The `spec[]` handoff keeps concerns separable while preserving traceability. The architectural claim is that the two arms instantiate **one harness pattern** with **dual evaluators**, not two unrelated chatbots. The “dual” in this paper’s title refers to exactly this: two evaluator arms of a single harness pattern, not two separate harnesses.

8.2 COEXISTENCE WITH CODING AGENTS

Task	Tool
Exploratory analysis, new method prototype	General coding agents
Environment and repo maintenance	General coding agents
SAP intent review, template pick, batch clone	This system (Arm A)
Spec validation, execution, confirm	This system (Arm B)
Shell-form comment accumulation, centralized review and evaluation	This system
SAP matching training export	This system training store

Coding agents optimize **authorship velocity**. This system optimizes **convergence velocity with evidence**, with intermediate-artifact evaluation points that match the traditional TLF workflow. The two are complementary, not competitive.

8.3 WHY ENGINEERING-FIRST SEQUENCING MATTERS

The project demonstrates a practical adoption sequence: first stabilize the harness lifecycle and control-plane boundaries, then instrument review and failure signals, then harden deterministic execution, and only then invest in retrieval, prompt, and model upgrades. This ordering avoids a common trap—over-investing in model sophistication before the workflow can reliably capture whether the model's output was actually useful.

8.4 LIMITATIONS

We report the following limitations honestly, because they define the real roadmap and because a systems paper that hides its gaps is less useful to readers considering adoption.

- **Long-round context stability.** The agent loop does not yet carry a context budget. In long review rounds, tool results accumulate, prior information is not compressed, and the model's effective context share degrades as rounds increase. This is the most pressing engineering risk: it does not show up in short demos but surfaces in real batch work. It is a workflow-structure problem, not a model-quality problem, and it is addressed first in the roadmap below.
- **Confidence feedback loop deferred.** When a reviewer confirms or rejects a spec, the system should learn which recalled memory rules helped the decision and which led it astray—confirming should reinforce the helpful rules, rejecting should weaken the unhelpful ones. This feedback loop is not yet implemented. The delay is deliberate: the system cannot yet trace which recalled rules informed a given decision, heuristic-based reinforcement risks corrupting unrelated rules, and the sample volume is still too low to separate signal from noise. This is the key gap on the self-growth path.
- **Mixed runtime architecture.** Due to the special nature of some statistical analysis models in clinical research—survival analysis, Kaplan-Meier, and certain PK models have mature R implementations that Python ecosystems have not fully matched—some analyses run in R via a sidecar service while the rest run in Python. Both stacks coexist with the friction that implies, and no clearly superior unified architecture has emerged yet.

8.5 FUTURE WORK

We organize the roadmap into three themes, ordered by dependency rather than by calendar.

Theme 1 — Context stability. Add context-budget awareness to the agent loop (token or character threshold), trigger history compression when exceeded, and replace old tool results with compact summaries while keeping only the most recent full payload. Introduce a structured continuation mechanism so that reaching the round cap returns a resumable signal instead of a silent failure. This theme is the prerequisite for trustworthy long-horizon batch work.

Theme 2 — Architecture convergence. Extract a unified workflow state machine from both arms so the convergence loop shares one vocabulary while evaluators stay stage-specific. Promote the handoff agent into its own engine module, lift batch orchestration out of routers into the engine layer, and converge the two persistence schemas toward a single audit export.

Theme 3 — Self-growth. Instrument agent runs with the telemetry needed to associate each confirm or reject with the recalled rules that informed the decision. Once that association is precise and sample volume is sufficient, close the confidence feedback loop so confirmations boost the rules that helped and rejections decay the rules that hurt. Complete the analyzer batch loop and the optional LLM sanity review. The longer-horizon target is an agent that progressively builds and converges more of the workflow on its own, on top of the same harness skeleton—justified by the labeled iterations the harness accumulates, not by model marketing claims.

9. CONCLUSION

This work argues that robust AI adoption for regulated clinical TLF delivery is fundamentally a control- and systems-engineering problem, not a prompt-engineering problem. The main conclusions are:

- **Harness plus dual evaluators provides stage-appropriate reliability for production TLF workflows.** With one shared control plane and two evaluator-specific arms, the system can run semantic intent selection and structural/numeric validation in the same lifecycle, localize failures to the responsible stage for faster correction, and maintain end-to-end traceability through one convergence loop.
- **The typed `spec[]` boundary is the natural split point that stabilizes the end-to-end workflow.** Upstream, it converts SAP interpretation into a reviewable and handoff-ready artifact; downstream, it constrains execution to deterministic, shell-aligned validation against explicit table intent; system-wide, it reduces cross-stage coupling, makes failures diagnosable by stage, and preserves replayability and governance from snapshots rather than conversational context. Therefore spec[] is not just an interface, but the control boundary that turns a multi-agent process into an auditable workflow
- **Audit and memory are first-class infrastructure for cumulative performance.** Three-layer memory plus append-only review telemetry transforms operational decisions (confirm/reject/patch) into reusable institutional signal, enabling measurable cross-study transfer rather than repeated cold starts.

The intended message of this paper is a control-engineering one. Models will continue to evolve; the requirement for explainable, replayable, human-accountable decisions will not. The durable asset is therefore not any particular model but the loop built around it—reference, measurement, feedback, and recorded state—which lets a team treat each future model as a swappable component inside an unchanged discipline. A harness built on explicit boundaries, durable memory, and auditable iteration gives teams something rare and durable: not just faster output today, but a compounding record of judgment that future teams can inherit, defend, and improve.

REFERENCES

1. CDISC. *Analysis Data Model (ADaM) Implementation Guide.*
2. International Council for Harmonisation. *E3: Structure and Content of Clinical Study Reports.*
3. International Council for Harmonisation. *E9(R1): Addendum on Estimands and Sensitivity Analysis in Clinical Trials.*

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Contact the author at:

Wenjun Zhu

Jiangsu Hengrui Pharmaceuticals Co., Ltd.
86-17721285930
Wenjun.zhu@hengrui.com