PharmaSUG China 2026 - Paper AA203-2026

AI-Augmented SDTM Workflow: A Methodological Framework

Yuqi Chen, Muqu Lv, Zhuozhao Zhan, Akeso,Inc.

ABSTRACT

In clinical trial data submissions, the Study Data Tabulation Model (SDTM) accelerates drug approval through standardized structures. Traditional SDTM generation is time-consuming, relying heavily on manual Case Report Form (CRF) annotation, specification writing, and datasets programming. To address these challenges, this paper presents an AI-augmented workflow framework that automates the end-to-end mapping process. This framework integrates programmatic execution, Large Language Model (LLM) automation, and human oversight: (1) Programmatic execution handles foundational tasks, including CRF reconstruction and domain specification template initialization; (2) Building on the structured CRF and specification template, LLM automates the manual efforts: drafting annotations, completing variable conversion rules, and generating and debugging executable code; (3) Rigorous review mechanisms ensure the quality of AI-generated content, including collaborative AI-human refinement and mandatory human approval. Compared to conventional processes, this framework significantly reduces manual effort while ensuring CDISC compliance and methodological consistency across projects.

Keywords: SDTM, LLM, workflow automation, CRF annotation, specification, code generation

INTRODUCTION

Clinical trial data standardization represents one of the most critical and time-consuming phases in pharmaceutical development. The Clinical Data Interchange Standards Consortium (CDISC) Study Data Tabulation Model (SDTM) standard provides a universal framework for organizing and presenting clinical trial data, accelerating drug approval through standardized structures. However, the transformation from raw source data to SDTM-compliant datasets requires substantial expertise and manual effort. Organizations typically dedicate substantial time to annotation, specification, and coding activities to deliver a high-quality SDTM package.

The traditional SDTM workflow involves three primary phases: Case Report Form (CRF) annotation, to map source fields to SDTM variables; SDTM specification development, where analysts define conversion rules for each domain variable; and code generation, where programmers implement the specifications in executable scripts. Each phase demands meticulous attention to regulatory requirements, control terminology, and domain-specific conventions, making the end-to-end mapping process highly labor-intensive. Manual SDTM transformation processes encounter several fundamental challenges:

1. Annotation inconsistency: Human annotators may interpret similar CRF structures differently, leading to inconsistent domain or variable mappings across forms and studies. Historical institutional knowledge often resides in individual experts rather than systematic, reusable references.

2. Specification fragility and implementation overhead: The translation of business logic into technical specifications is inherently error-prone. When authoring conversion rules in natural language, analysts are prone to discrepancies between the described logic and actual intent—such as referencing non-existent variables or articulating imprecise transformation conditions. These ambiguities significantly amplify the cognitive load required to navigate complex inter-variable relationships, key derivation logic, and terminology constraints. This "specification fragility" directly translates into high code generation overhead, as programmers must bridge the gap between flawed documentation and executable code. The subsequent debugging cycles and rework not only demand specialized programming expertise but also create a compounding maintenance burden that complicates future maintenance efforts.

To address these challenges, this paper presents an AI-augmented workflow framework that automates the end-to-end SDTM mapping process. The proposed methodology integrates three core pillars to ensure efficiency, accuracy, and CDISC compliance:

1. Programmatic execution for foundational tasks: Automating the initial setup through deterministic programming. This includes CRF reconstruction to create structured source inputs and domain specification template initialization to establish standardized frameworks for downstream processing;

2. LLM automation for manual-intensive efforts: Building upon the structured CRF and initialized specification templates, Large Language Models (LLMs) are deployed to automate traditionally manual tasks. This encompasses drafting CRF annotations, completing complex variable conversion rules, and generating as well as debugging executable programming code;

3. Rigorous human oversight and review mechanisms: Ensuring the reliability and quality of AI-generated content through structured governance. This involves collaborative AI-human refinement loops for iterative improvement and mandatory human approval gates at critical milestones to guarantee regulatory compliance and methodological consistency across projects.

FRAMEWORK OVERVIEW

WORKFLOW ARCHITECTURE

The workflow architecture is designed as follow:

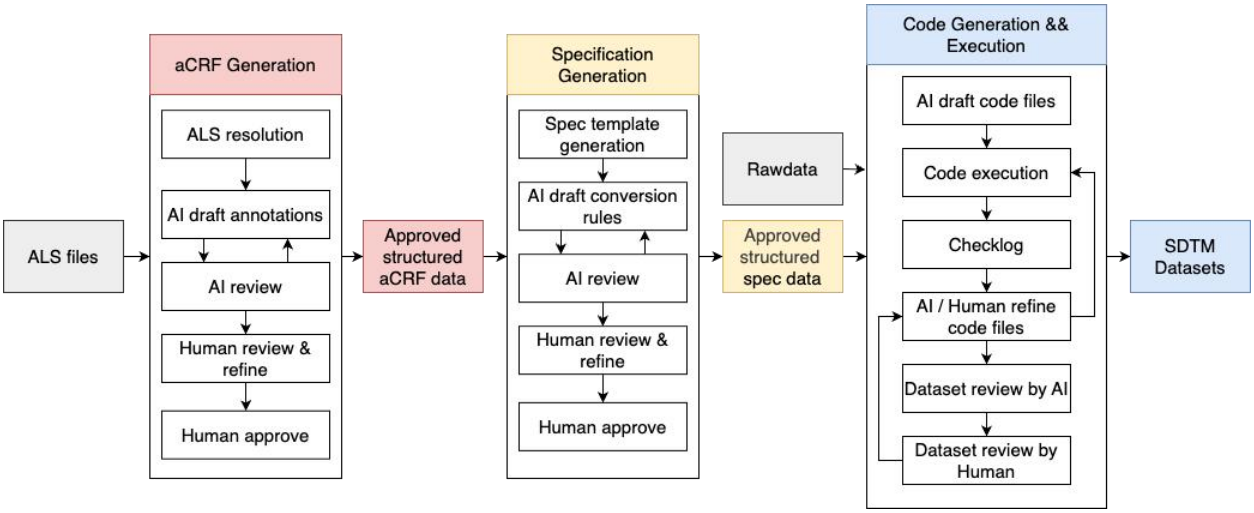

Figure 1. Overview of the Workflow

Compare to the traditional workflow:

1. It starts with the Architect Loader Spreadsheet (ALS) file as the input of the whole workflow instead of the PDF format CRF file;

2. All the manually drafting were replace with AI;

3. Output of each stage will be persistence into database instead of multiple files, and call via API by downstream stages;

4. Human review/ refine the results of each stage on unified web UI instead of reading / modifying files;

5. Each stage could be automatically launched by human if the items in previous stage have been approved by human.

FUNCTIONAL MODULES ARCHITECTURE

Based on the three stages of workflow, the functional modules in this framework are designed as follow:

Modules	Active Stages	Responsibilities
ALS Resolver	aCRF	Resolve ALS files and reconstruct CRF objects
Annotation Agent	aCRF	Agent for generate / review annotation information in CRF
CRF Controller	aCRF / Spec	Read / Manage the CRF data
Spec Agent	Spec	Agent for generate / review spec and conversion rules
Spec Controller	Spec / Coding	Read / Manage the Spec data
Coding Agent	Coding	Agent for generating or refining code files
Code Controller	aCRF	Read / Manage the code data
Standard Data Controller	aCRF / Spec	Read / Manage the standard data, e.g. SDTM metadata / control terminology
Stage Controller	aCRF / Spec / Coding	Manage the status and transition of the stages in the workflow

Table 1. Functional modules inside the framework

BASIC INFRASTRUCTURE

The Basic Infrastructure comprises with Standard Data Controller and Stage Controller, serve as the foundational infrastructure that orchestrates the workflow and provides centralized, standardized knowledge across all three stages (Annotation, Specification, and Coding).

STAGE CONTROLLER

The Stage Controller acts as the central state machine and execution engine for the entire SDTM automation pipeline. Its primary responsibilities include:

- **Stage Gating and Dependency Resolution**: It strictly enforces the sequential dependencies between stages. A subsequent stage (e.g., Specification Generation) is programmatically blocked from initiation until all prerequisite artifacts (e.g., aCRF) from the preceding stage have achieved an explicit "Approved" status via human review.
- **Granular Event Logging and Observability**: It maintains a comprehensive, time-series event log tracking the lifecycle of every individual artifact (e.g., specific CRF forms, SDTM domains). This ensures full traceability and aids in debugging;
- **Pipeline Monitoring Dashboard**: To provide human operators with real-time visibility into the pipeline's health, a dedicated monitoring dashboard is strongly recommended. This UI allows users to inspect the granular status of all workflow items, identify bottlenecks, and manually trigger control actions such as Start, Pause, Stop, or Rerun for specific tasks or entire stages, ensuring human oversight over the autonomous execution.

STANDARD DATA CONTROLLER

Standardized reference data, specifically SDTM metadata and Controlled Terminology, is critical for validating annotation outputs and guiding the generation of conversion rules. The Standard Data Controller acts as the centralized data provider and semantic anchor for the AI agents. It manages two primary categories of reference data:

- **SDTM Domain Metadata**:
    1. **Standard Domains**: The official domain definitions, classes, structures, and variable specifications strictly derived from the targeted SDTMIG;
    2. **Custom/Non-Standard Domains**: Metadata definitions for custom domains, dynamically aggregated from historical project repositories or internal company-specific data standards.

- **Controlled Terminology**:
    1. **Non-Extensible Codelists**: The strict, non-modifiable codelist officially published by CDISC(e.g., "ACN" for AEACN, "OUT" for AEOUT);
    2. **Extensible Codelists**: CDISC published extensible codelist (e.g., "UNIT" for --ORRESU, "LOC" for --LOC), enriched and extended with sponsor-specific or historical project-specific terms to accommodate real-world clinical data nuances.

ACRF GENERATION STAGE

The annotated Case Report Form (aCRF) Generation Stage utilizes the ALS file as input and exports structured CRF data to the subsequent stages of the workflow.

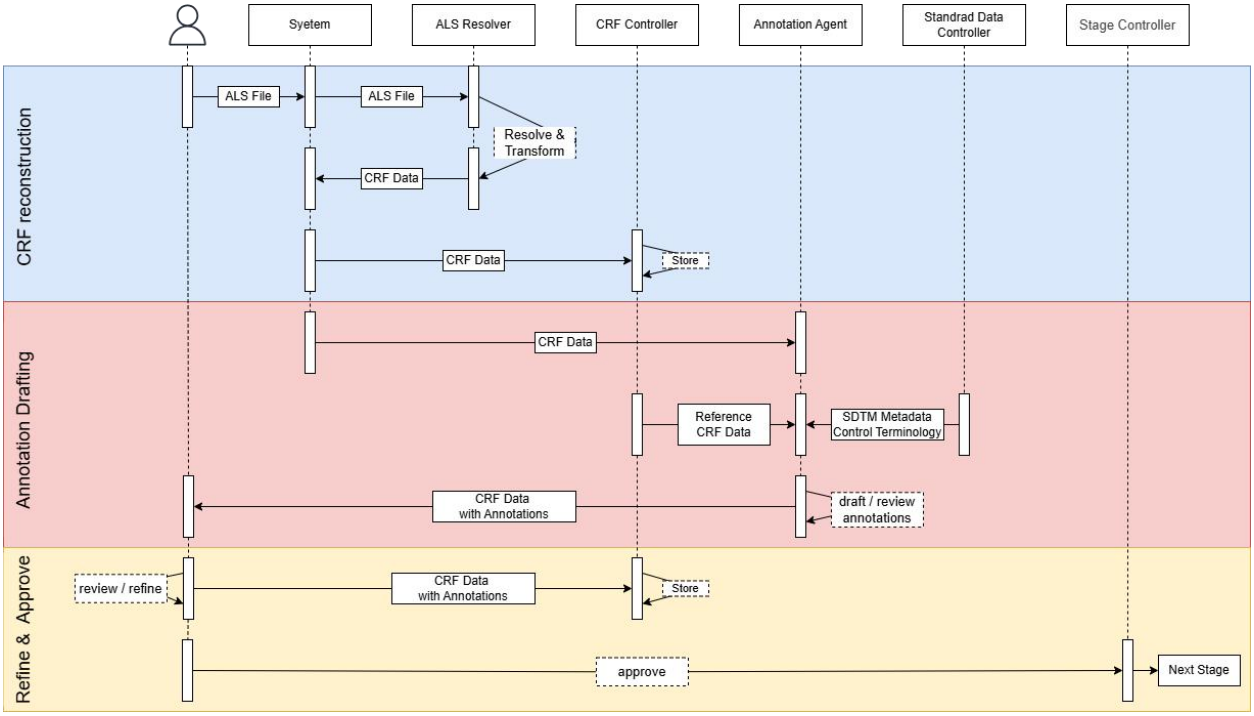

Figure 2. Workflow Design of aCRF Generation Stage

This stage comprises three primary steps:

- CRF Reconstruction: Parsing the ALS file that exported by Electronic Data Capture (EDC) systems and restructuring into a unified data model;
- Annotation Drafting: An AI-driven Annotation Agent drafts and reviews CRF annotations by referencing:
    1. **Historical structured aCRF data**: To ensure consistency in annotation methodologies across studies;
    2. **SDTM Metadata**: To validate that variables declared in the annotation text (excluding supplemental qualifiers) exist within the specified target SDTM domains.
- Refinement and Approval: Human reviewers evaluate, refine, and approve the annotated CRF via a dedicated UI.

ALS RESOLVER

The ALS Resolver is responsible for processing various ALS files exported from different EDC systems and reconstruct them into a unified, structured CRF data format. Architecturally, the ALS Resolver is

designed as an abstract base class with standardized interfaces, allowing specific implementation logic for different EDC vendors (e.g., Medidata RAVE, Taimei eCollect).

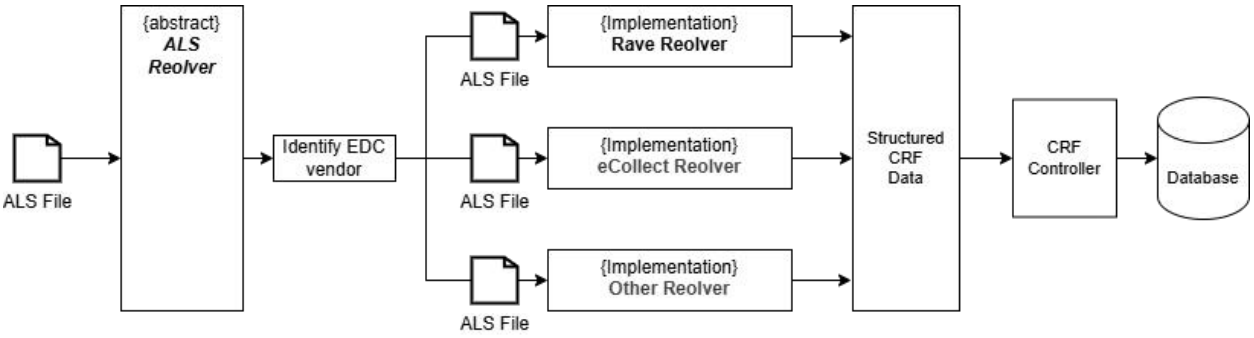

Figure 3. Implement of ALS Resolver

Reason For CRF Reconstruction

The ALS Resolver accepts ALS files rather than PDF exports from EDC systems. This design choice is fundamental to reconstruction accuracy for the following reasons:

- Direct System Mapping: ALS files contain actual item identifiers (Item OIDs) and binding relationships that are typically lost in PDF rendering. Item OIDs serve as critical anchors for downstream SDTM mapping;
- Complete Metadata Retention: ALS retains item data types (e.g., free text, dropdown), format patterns (date formats, numeric constraints), and codelist/option values that PDF exports obscure.

Design of the Structured CRF Data Model

Utilizing a unified structured CRF data model decouples the core framework from specific EDC vendors, allowing the system to focus purely on CDISC mapping business logic. After processing, the key design principles include:

Abstraction and Preservation of Control Types

All data collection items in a CRF can be abstracted and mapped into four primary control types: Free Text, Single Selection (Radio / Drop down), Multi-Selection (Checkbox), and Date/Time. Retaining this control-type metadata within the structured CRF data is essential for two main reasons:

- Accurate UI Rendering: It ensures the faithful visual reconstruction of the CRF layout and input behaviors within the Annotation UI;
- Downstream Programming Automation: It enables intelligent filtering and processing in subsequent stages. For instance, explicitly identifying Date/Time control types allows the system to automatically isolate these variables and apply specific SDTM programming rules, such as ISO 8601 formatting for --DTC variables

Annotation Slots Across CRF Hierarchies

To support comprehensive and precise annotations, the framework defines dedicated annotation slots across four distinct hierarchical levels within the CRF data model:

- Form Level: Captures form-wide annotations. This includes mapping an entire form to a specific SDTM domain qualifier or topic. For example, mapping the "Hematology" form to "LBCAT = HEMATOLOGY", or mapping the "Informed Consent" form to "DSTERM = INFORMED CONSENT OBTAINED when DSCAT = PROTOCOL MILESTONE";
- Item Level: The most fundamental annotation slot, targeting individual data collection fields. It maps specific CRF items to their corresponding SDTM variables (e.g., mapping the "Sample Collection Date" item in the "Blood Chemistry" form to "LBDTC");.

- Option/Codelist Level: Targets specific response options within selection controls. This slot maps individual CRF item options to conditional SDTM variable assignments. A typical use case is mapping reasons for non-randomization in the Disposition domain:
    - Eligibility Criteria not Met → "DSTERM when DSDECOD = SCREEN FAILURE"
    - Consent withdrawn → "DSTERM when DSDECOD = WITHDRAWAL BY SUBJECT"
    - Investigator Decision → "DSTERM when DSDECOD = PHYSICIAN DECISION"
    - Lost to Follow-up → "DSTERM when DSDECOD = LOST TO FOLLOW-UP"
    - Adverse Event → "DSTERM when DSDECOD = ADVERSE EVENT"
    - Death → "DSTERM when DSDECOD = DEATH"
    - Other → "DSDECOD = OTHER"
- Fixed Unit Level: Applies to static, non-editable unit labels associated with measurement items. It ensures accurate mapping to the Result Unit variable (e.g., mapping the fixed text label "mmHg" directly to item `Systolic Blood Pressure`).

Handling Log Forms

If a CRF contains log forms (e.g., Adverse Events, Concomitant Medications and Lab Tests), it indicates a one-to-many relationship in the underlying data. For instance, a leading question asking if a lab test was performed may trigger multiple repeating records for specific lab parameters. While most EDC systems merge these into a single logical view, traditional programmers must manually "unroll" or flatten this data during SDTM programming. We recommend flattening the log line/repeating structure from a one-to-many hierarchy into a normalized one-to-one structure within the reconstructed CRF data. This prevents misleading contextual information when the LLM drafts annotations and reduces the complexity of generating downstream mapping specifications.

CRF CONTROLLER

The CRF Controller serves as the central data access layer within the framework. It is responsible for the comprehensive create, read, update, and delete operations of the unified structured CRF data, as well as the management of all associated annotation metadata. Acting as the critical bridge between the underlying storage and the upper-layer consumers (AI Agents and UI), it fulfills the following core functions:

Historical Data Retrieval for AI Annotation

To ensure the AI generates consistent and study-specific annotations, the Annotation Drafter Agent relies on past knowledge. The CRF Controller queries the repository of historical, approved aCRFs to retrieve structurally or semantically similar forms for improving the accuracy and standardization of the initial annotation drafts;

Data Provisioning for CRF UI Rendering and Annotation Refinement

The CRF Controller acts as the data supplier for the frontend CRF UI. It dynamically fetches and delivers the reconstructed form layouts, abstracted control types (e.g., Free Text, Date/Time), and existing AI-generated annotation tags. This ensures the UI can render a high-fidelity, interactive, and visually accurate representation of the traditional aCRF for the end-users. When human reviewers interact with the CRF UI to validate and refine the AI-drafted annotations, the CRF Controller handles all state changes and user interactions.

ANNOTATION AGENT

The Annotation Agent leverages LLM to automate the drafting and quality control of CRF annotations. It operates via two specialized sub-agents:

- Annotation Drafter: This agent queries the vector index of historical aCRFs through the CRF

Controller to retrieve semantically similar forms as references. It then generates initial SDTM annotation drafts for the target form based on this retrieved contextual knowledge;

- Annotation Reviewer: Acting as an automated quality assurance mechanism, this agent retrieves SDTM metadata and Controlled Terminology (CT) from the Standard Data Controller. It evaluates the Drafter's output against predefined annotation guidelines. If discrepancies or rule violations are detected, the Reviewer logs the issues and flags them within structured comment data for human intervention.

The Annotation Agent is orchestrated by the framework and manages multi-turn conversations to accommodate subsequent user modifications:

- Drafter Invocation: The Stage Controller invokes the Drafter. Upon completing the initial annotation draft, the Drafter calls the persistence function within the CRF Controller to save the results. The conversation context is then maintained in a waiting state, ready for Human-in-the-Loop (HITL) modifications;
- Reviewer Invocation: The Reviewer is triggered automatically via review hooks activated by the CRF Controller whenever annotations are created or updated. Crucially, the structured comments generated by the Reviewer are routed directly to the human reviewer rather than back to the Drafter. After evaluating these comments, the human user retains full agency to decide whether to feed them back into the Drafter as prompts for refinement.

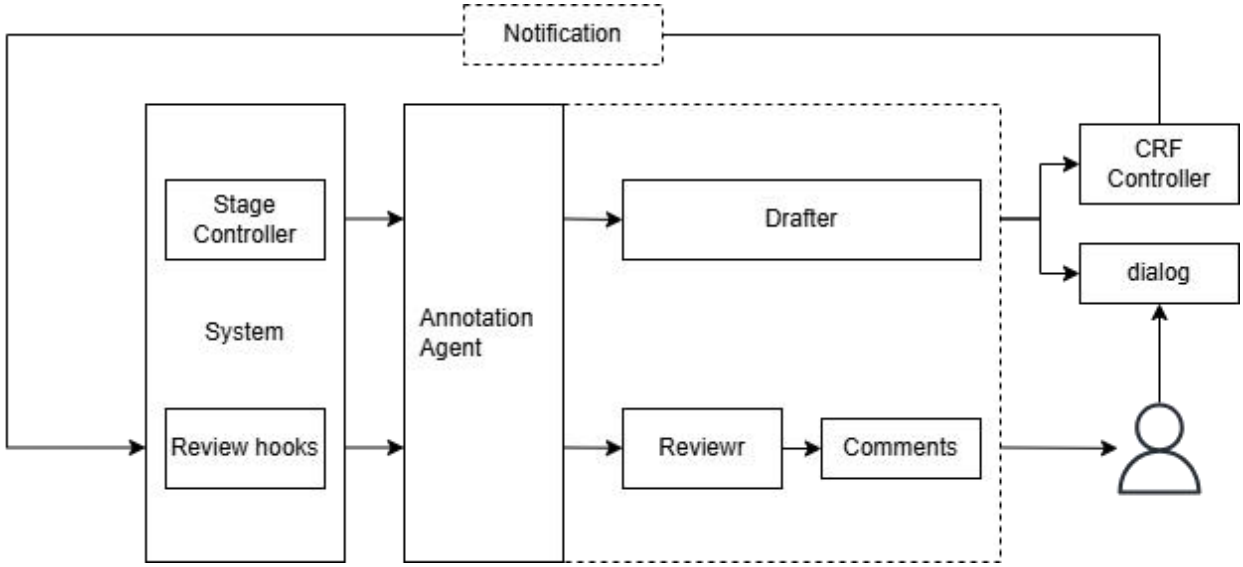

Figure 4. Implement of Annotation Agent

CRF UI

To facilitate human review and refinement, a dedicated UI is provided. The UI is designed with the following core capabilities:

- Visual Fidelity to Traditional aCRF:
  1. The layout and items closely mimic the visual representation of the unique PDF CRF.
  2. Annotation tags are positioned contextually, mirroring traditional aCRF styles (e.g., placed adjacent to the specific item, option, or fixed unit).
- Intuitive Editing: Users can seamlessly add, modify, or remove annotation text and mapping variables;
- Advanced Search: Robust search functionalities for both CRF items and existing annotation texts;
- Role-Based Access Control: The system enforces granular permissions. Users assigned as "Form Owners" or "Project Leader" are granted write/edit privileges for their specific forms, while other

stakeholders are restricted to read-only access to maintain data integrity.

SPECIFICATION GENERATION STAGE

The Specification Generation Stage utilizes the structured aCRF data as input and produces structured SDTM spec data for the subsequent stages of the workflow.

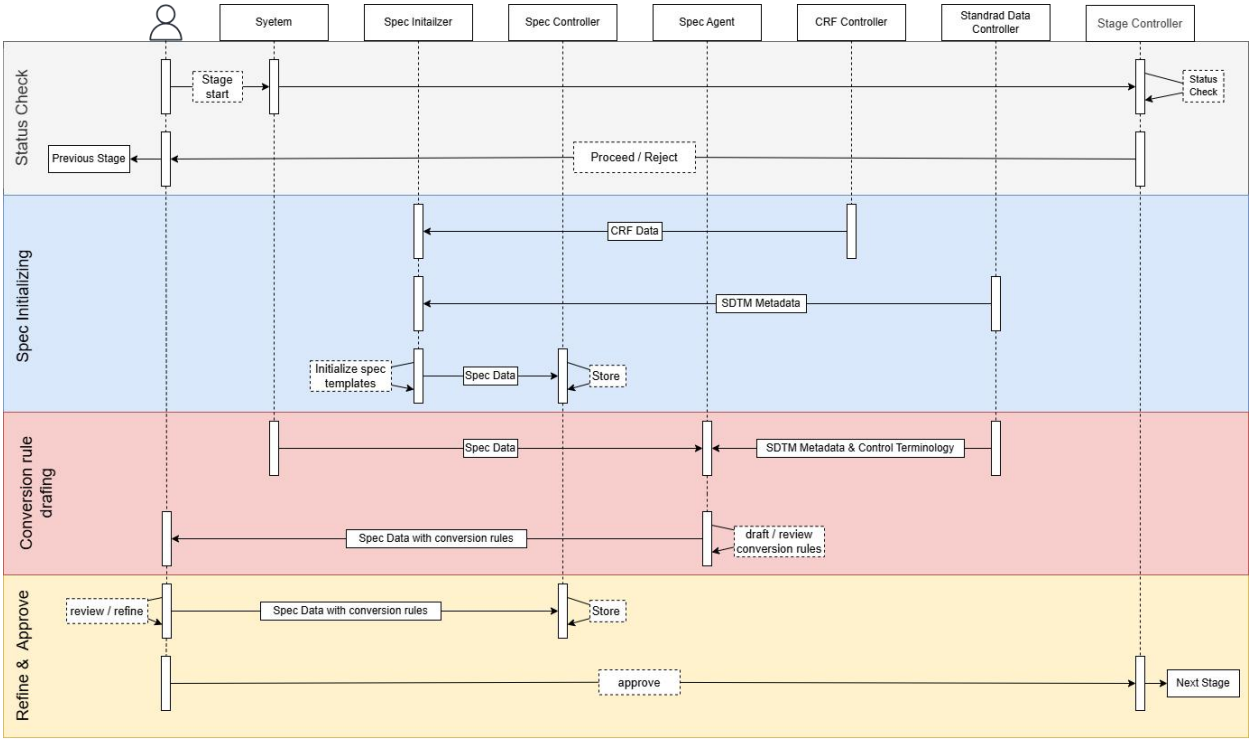

Figure 5. Workflow Design of Specification Generation Stage

This stage comprises four primary steps:

- Status Check: The Stage Controller verifies whether all forms from the preceding aCRF Annotation Stage have been approved. The subsequent steps cannot be initiated until the approval status of all forms is confirmed;
- Spec Initialization: Based on the annotation text within the structured CRF data model, the system extracts all valid mapped variables. It then merges these with variables marked as "Req" (Required) or "Exp" (Expected) Core in the SDTM metadata—retrieved via the Standard Data Controller—to construct initial domain templates without conversion rules;
- Conversion Rules Drafting: An AI-driven Spec Agent drafts and reviews mapping and derivation rules by referencing:
    1. **SDTM Metadata**: To validate that variables declared in the annotation text (excluding Supplemental Qualifiers) exist within the specified target SDTM domains;
    2. **Controlled Terminology**: To ensure consistency in value-level mapping and derivation methodologies across studies.
- Refinement and Approval: Human reviewers evaluate, refine, and approve the generated specifications via a dedicated UI.

SPEC INITIALIZER

The Spec initializer is responsible for populating the specification template using structured CRF data. It constructs the foundational datasets without conversion rules while preserving the raw source information required for subsequent rule generation.

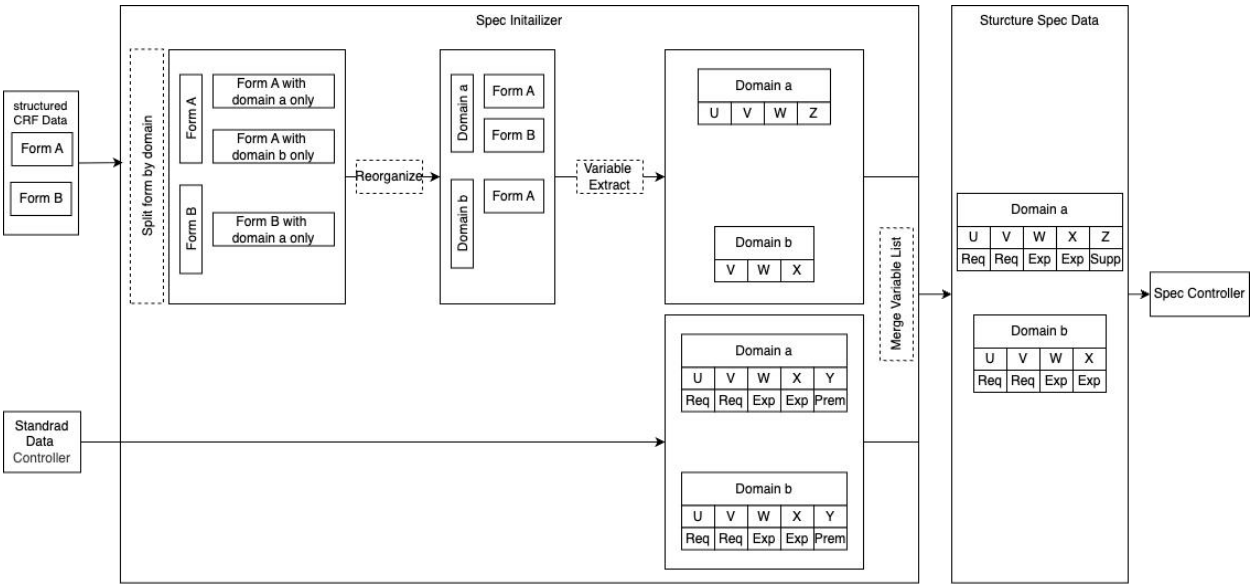

Figure 6. Implement of Spec Initializer

There are 5 core steps to initialize the spec template:

- Reorganize: Group the CRF data by their mapped target SDTM domains and mask annotations unrelated to the current domain being processed;
- Extract: Parse and extract the target SDTM variables from the raw annotation text;
- Merge and Enrich: Based on the official domain variable list from SDTM metadata, update the source information for existing variables and append required/expected variables that lack direct CRF mappings. The following source information is preserved for the Spec Agent to generate accurate conversion rules:
  - **Form OID**: Required
  - **Item OID**: Optional
  - **Item Type**: Optional
  - **Item Option**: Optional
  - **Item Unit**: Optional
  - **Annotation Text**: Required
- Filter: Exclude variables that have no source mapping and are not classified as "Req" or "Exp" Core (e.g., filtering out "Perm" Core variables with no CRF origin);
- Append Paired Variables (Optional): Automatically append logically paired variables based on FDA Validation Rules (e.g., ensuring --DTC and --DY are generated as a pair).

SPEC CONTROLLER

The Spec Controller serves as the specification data access layer. It is responsible for the comprehensive create, read, update, and delete operations of the domain spec data. It acts as the critical bridge between the underlying storage and the upper-layer consumers (AI Agents and the UI).

SPEC AGENT

Similar to the Annotation Agent, the Spec Agent consists of two specialized sub-agents operating in a collaborative loop:

- Spec Drafter: Invoked by the Stage Controller, this agent generates mapping and conversion rules for the variables in the structured spec data. It bases its generation on both general and domain-specific implementation rules. The conversation context is then maintained in a waiting state, ready for HITL modifications just like the Annotation Agent;
- Spec Reviewer: Invoked automatically via review hooks in the framework whenever spec data is created or updated. It queries the SDTM domain metadata and Controlled Terminology from the Standard Data Controller to audit the conversion rules generated by the Drafter. Any discrepancies, logic flaws, or rule violations are logged into structured comment data for human review.

Handling of Generic Conversion Rules

In SDTM mapping, several variables share identical conversion logic across all domains (e.g., STUDYID, USUBJID, --SEQ, --DY). In clinical programming practice, it is strictly required to process these variables consistently to ensure data integrity and cross-study standardization.

To address this, the framework introduces a **Function Marker** mechanism. Rather than relying on the AI to generate these repetitive rules, it is highly recommended to pre-populate these predefined function markers (e.g., @assign_usubjid, @assign_seq) directly into the conversion rule fields during the Spec Initialization stage.

This design enforces strict behavioral constraints and provides critical benefits across the automated workflow:

- Spec Agent Constraints:
    1. **Drafter**: During the Conversion Rules Drafting step, the Spec Drafter is explicitly constrained not to modify or overwrite any conversion rules that already contain a Function Marker. It simply bypasses these fields and focuses its generation capacity on complex, domain-specific derivations;
    2. **Reviewer**: The Spec Reviewer is configured to automatically skip auditing rules containing these markers. This optimizes the QA process, reduces unnecessary LLM token consumption, and prevents false-positive flags on universally accepted standard logic.
- Coding Agent Optimization: Downstream, the Function Marker signals the Coding Agent to invoke a standard function or macro rather than attempting to implement the logic from scratch based on text descriptions. This eliminates coding hallucinations and guarantees that standard variables are processed identically across all domains.

SPEC UI

The Specification UI serves as the dedicated workspace for human to review, refine, and finalize the generated specifications. The key capabilities and constraints include:

Domain-Level Management

- Standard Domains: For standard SDTM domains (e.g., DM, AE, CM), the UI strictly locks core metadata to ensure compliance. Users cannot modify the Domain Name, Label, Class, or Structure Descriptions.
- Custom Domains: For non-standard, custom domains, users are granted full permissions to modify these domain-level attributes.
- Key Variables: Users are permitted to update the designated "Key Variables" (used for identifying unique records) across all domains, regardless of whether they are standard or custom.

Variable-Level Management

- Standard Domain Variables: For variables belonging to the standard domain classes, the UI enforces strict immutability on core metadata. Users cannot modify the Variable Name, Label, Data Type, or assigned Controlled Terminology, nor can they alter the predefined variable order (ordinal sequence) dictated by the SDTM Implementation Guide (SDTMIG);
- Supplemental Qualifiers: For variables mapped to the Supplemental domain (SUPP--), users are granted the flexibility to add/remove variables and modify their metadata and ordering as needed;
- Conversion Rules: Users retain full editorial control to update, refine, or override the AI-generated conversion/mapping rules for all variables.

Context-Aware Reference Navigation

To significantly improve review efficiency and reduce context-switching, the UI integrates direct, contextual hyperlinks to external reference materials. Reviewers can instantly access the original Variable Source (e.g., deep-linking back to the specific aCRF page) and the controlled terminology definitions without leaving the specification workspace.

CODING STAGE

The Coding Stage represents the final phase of the AI-augmented SDTM workflow, responsible for translating the approved SDTM specifications into executable programming code. Unlike traditional manual programming or naive LLM code generation, this stage employs a highly structured, dependency-aware architecture governed by a Code Controller and a specialized Coding Agent. This design ensures that the generated code is not only syntactically correct but also logically sequenced according to complex cross-domain dependencies. The workflow of the Coding Stage is driven by the collaboration between the Coding Agent and the Code Controller. The architecture is designed as follows:This stage operates through three primary phases:

- **Status Verification**: The Stage Controller verifies the approval status of all domain specifications from the preceding stage. The coding workflow is strictly gated and cannot be initiated until all target domain specs are explicitly marked as "Approved";
- **Autonomous Generate-Execute-Refine Loop**: This phase is orchestrated by the Coding Agent and Code Controller with 4 sub-modules. First, the Coding Agent generates variable-level code plans based on the domain specifications, which the Code Manager then merges with validated boilerplate templates to assemble the actual executable code files. Next, the Execute Planner resolves cross-domain dependencies and sequentially triggers the Code Executor to run the code in an isolated process, strictly managing domain states (Pending, Executing, and Completed). Finally, the Log Checker evaluates the execution output; if errors or warnings are detected, the system enters an automated refinement loop, iteratively correcting the code plan and re-executing until the log passes predefined quality thresholds;
- **Human-In-The-Loop Review and Refine**: Clinical programmers review the generated code, resulting datasets, and execution logs. They possess the authority to manually edit the code, approve the final output, or provide targeted natural language prompts to instruct the AI to perform further refinements.

CODING AGENT

Launched by the Stage Controller, the Coding Agent is the core reasoning engine responsible for translating mapping logic into executable syntax. Its workflow is governed by the following principles:

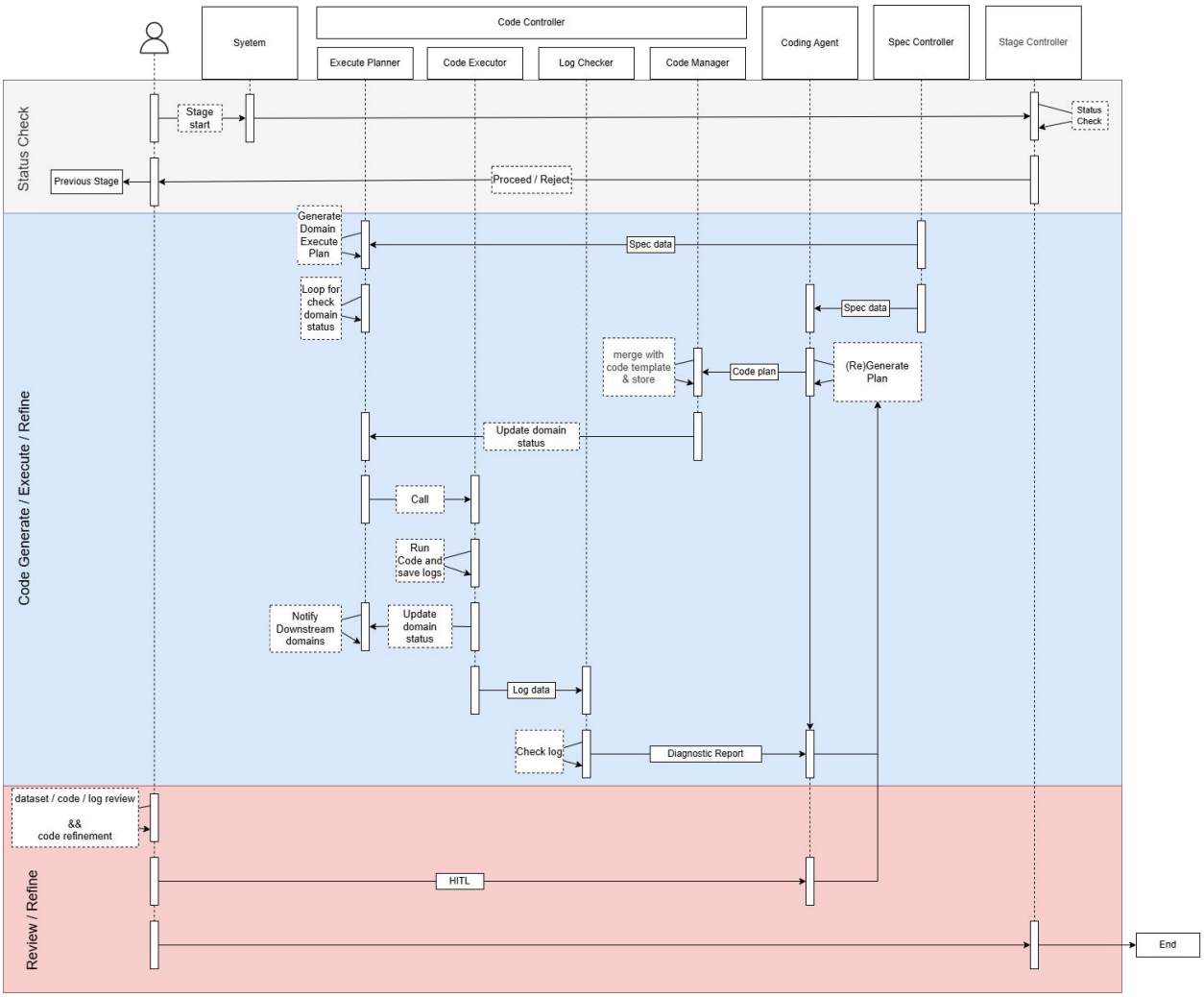

Figure 7. Workflow Design of Coding Stage

- **Dependency-Aware Planning**: Before writing any code, the Agent analyzes all conversion rules and maps the dependencies between variables (e.g., calculating --DY requires --DTC). It generates a step-by-step code plan to ensure variables are derived in the correct topological order.
- **Rule-Specific Translation**:
    1. **Natural Language Rules**: For complex derivations described in natural language, the Agent interprets the logic and generates the corresponding programming syntax;
    2. **Function Markers**: For standardized rules containing Function Markers (e.g., @assign_usubjid, @assign_seq), the Agent retrieves the predefined signature and usage instructions for that function, automatically populating the required parameters based on the domain context.
- **Contextual Memory Management**: All interactions, including initial generation prompts, intermediate code drafts, and execution feedback, are persisted as structured conversation records (in a database or file system). This ensures that when the Agent receives a Log Checker report or a human refinement prompt, it can seamlessly resume the context and apply targeted fixes without losing the broader scope of the domain mapping.

CODE CONTROLLER

The Code Controller acts as the central orchestration engine of the Coding Stage. It manages the life cycle of the code from assembly to execution and debugging. The Code Controller comprises four critical sub-modules:

Code Manager

The Code Manager serves as the assembly engine and repository for the programming artifacts.

- **Code Assembly**: It merges the variable-level transformation logic from the Code Plans with predefined boilerplate templates. These templates are primarily utilized to initialize the runtime environment—such as defining library reference, loading standard utility macros/functions, and configuring global system environment—before the domain-specific logic runs. By prepending this initialization sequence to the AI-generated logic, the Code Manager ensures a standardized and fully prepared execution context, ultimately outputting the complete, executable code files for each domain.
- **Storage**: It securely stores the domain-specific Code Plans and the executable code files for each domain.

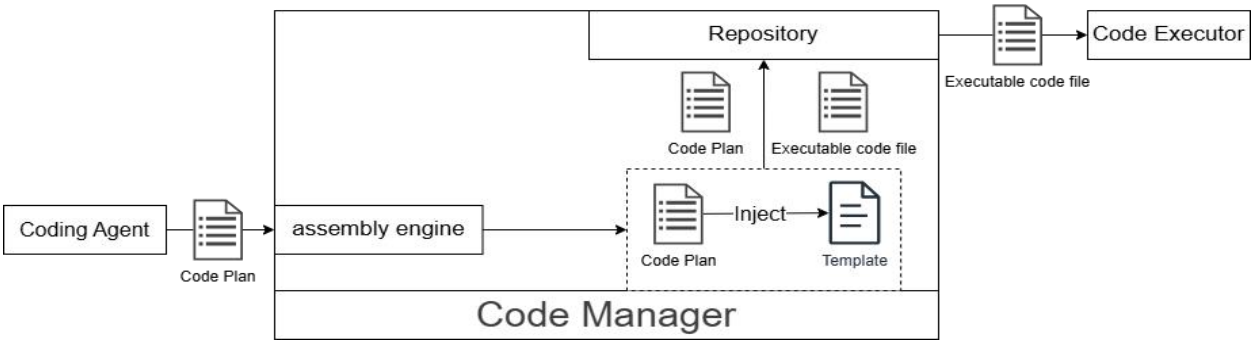

Figure 8. Implement of Code Manager

Execute Planner

While the Code Manager handles single-domain code assembly, the Execute Planner manages the cross-domain execution strategy since SDTM datasets are highly interdependent.

- **Dependency Resolver**: Ingesting specification data from the Spec Controller, this component maps out the dependency graph across all target domains. Based on these inter-domain dependencies, it groups domains into sequential execution ranks and formulates a comprehensive execution plan, which is then instantiated and saved as the Plan Context;
- **Plan Context**: Designed as a hierarchical two-layer queue structure to manage the execution flow. The outer queue represents execution ranks (i.e., sequential batches of domains grouped by their dependency levels), maintaining rank-level states (Pending, Completed). The inner queue contains the individual domains assigned to each specific rank, maintaining domain-level states (Pending, Executing, Completed);
- **State Machine Manager**: This component continuously monitors the status of both ranks and domains through a strict state machine. State transitions are hierarchical: updates to individual domain states cascade upward to the rank level. A rank automatically transitions to the Completed state only when all constituent domains within its inner queue have reached the Completed state;
- **Turn Scheduler**: Acting as the dispatch engine, it periodically polls the Plan Context to identify the highest-priority Pending rank (the most upstream dependency level). It then invokes the Code Executor to run the code for the domains within that rank and simultaneously notifies the State Machine Manager to transition those specific domains into the Executing state.

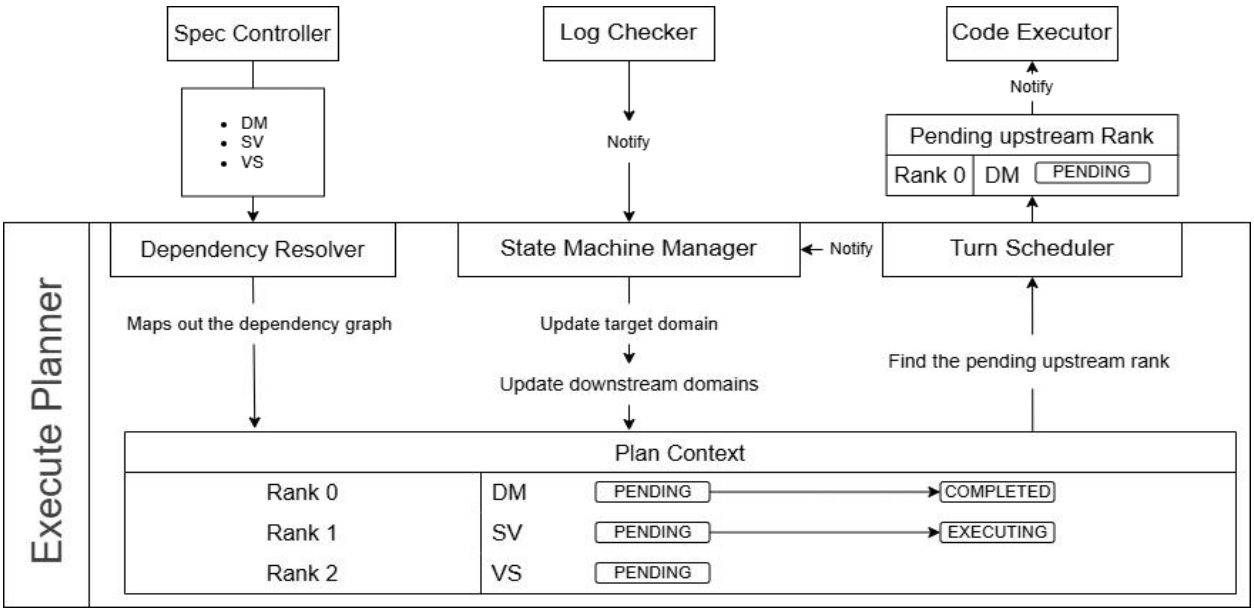

Figure 9. Implement of Execute Planner

Code Executor & Log Checker

These two components function together as a unified abstraction layer that decouples the core orchestration logic from the underlying programming environments. While they define standard interfaces for code execution and log validation within the system, their concrete implementations are language-specific. This design ensures the framework's extensibility across different technology stacks.

- Code Executor: Acting as the runtime engine, it receives execution instructions from the Execute Planner and runs the assembled code files generated by the Code Manager within the designated, language-specific environment. It captures the standard output and raw execution logs, passing them immediately to the Log Checker for evaluation.
- Log Checker: Functioning as the automated quality gate, it parses the execution logs provided by the Code Executor, scanning for syntax errors, missing variable warnings, or data type mismatches based on language-specific diagnostic rules. If the log is clean, the Log Checker signals the Execute Planner to transition the domain's state to Completed. If errors or critical warnings are detected, the Log Checker intercepts the failure, extracts the relevant diagnostic context, and triggers the auto-debugging loop for iterative refinement.

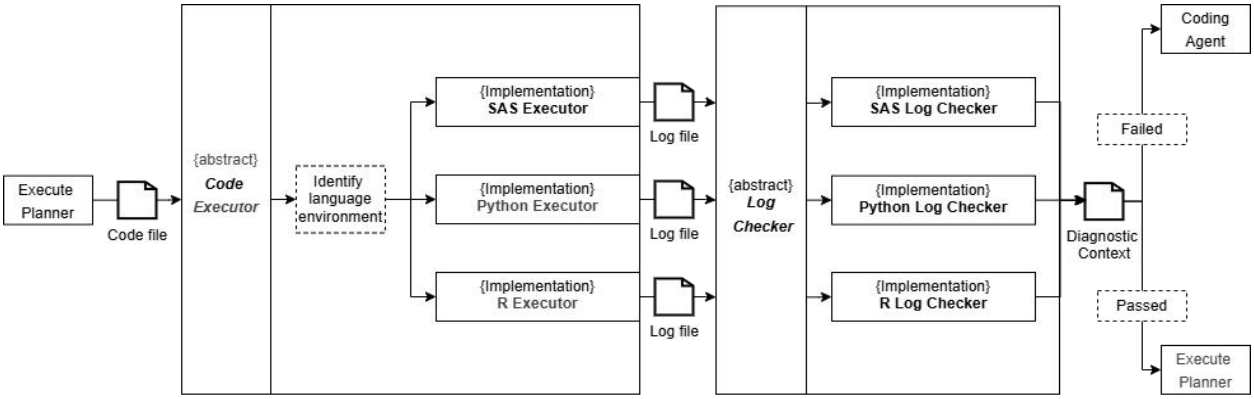

Figure 10. Implement of Code Executor and Log Checker

CODING UI

The Coding UI serves as the interactive workspace for clinical programmers to review, debug, and finalize the AI-generated assets. The key capabilities include:

- **Code Plan Visualization**: A dedicated panel displaying the AI's logical execution steps, allowing reviewers to quickly grasp the structural intent and variable dependency graph of the code;
- **Interactive Code Editor**: A fully featured IDE-like editing area where users can manually modify the generated code, complete with syntax highlighting and line numbering;
- **Execution & Validation Controls**: Action buttons to manually trigger code execution, halt running processes, and invoke the Log Checker on demand;
- **Smart Log Viewer**: A log visualization panel that automatically highlights errors and warnings identified by the Log Checker, providing direct jump-links to the corresponding lines in the code editor for rapid debugging;
- **Integrated File Explorer & Native App Integration**: Recognizing that clinical programmers frequently rely on specialized local tools for deep-dive validation, the UI features a built-in file navigator for all generated artifacts. This component provides one-click access to physical outputs (e.g., .sas7bdat/.xpt datasets, .sas scripts, and .log files), seamlessly invoking the user's OS-default applications. This ensures a frictionless handoff between the AI-driven web workspace and the programmer's traditional native debugging environment;
- **Conversational Refinement Dialog**: An integrated chat interface allowing users to provide natural language feedback. The chat is directly linked to the Coding Agent's context, enabling seamless AI-driven code modifications.

DISCUSSION

ADVANTAGES OF THE PROPOSED FRAMEWORK

- **Reduction in Manual Effort**: The framework dramatically boosts SDTM project efficiency from start to finish by automating critical tasks across annotation, specification, and coding. Historical references power rapid and consistent CRF annotation, AI-driven tools generate precise conversion rules for specifications, and automated code synthesis streamlines execution. By integrating these capabilities, the framework accelerates each stage while ensuring seamless transitions—significantly reducing manual effort, minimizing errors, and enabling timely delivery of compliant SDTM outputs;
- **Programmatic Orchestration with Deterministic Guardrails**: Unlike naive LLM applications that rely entirely on generative AI, this framework synergizes LLM semantic reasoning with deterministic programmatic execution. Foundational modules, such as the ALS Resolver for CRF reconstruction and the Spec Initializer for template population, operate on strict, rule-based logic. By structuring the raw inputs and initializing standardized templates before invoking LLMs, the framework significantly constrains the AI's generative space. This hybrid approach minimizes hallucination risks and ensures that downstream AI agents operate within a highly structured, CDISC-compliant context;
- **Improved Consistency**: By anchoring annotation to historical references, the framework ensures tha t similar CRF structures receive similar annotations across studies. This consistency facilitates cross study analysis and regulatory review;
- **Structured Review Process**: The multi stage review architecture ensures quality at each transition p oint. AI reviewers catch systematic issues (control terminology violations, missing required variables) while human reviewers validate domain specific decisions;
- **Quality Assurance**: The framework replaces single-pass AI generation with a rigorous, multi-layered quality assurance architecture. At the micro-level, the dual-agent design (Drafter and Reviewer) creates an adversarial refinement loop where the Reviewer continuously audits the Drafter’s output against SDTM metadata and Controlled Terminology. At the macro-level, the Stage Controller enforces strict stage-gating, ensuring that no downstream process can commence until the upstream

artifacts have passed both AI review and mandatory HITL approval. This "defense-in-depth" strategy is critical for meeting the stringent regulatory and auditability requirements of clinical data submissions.;

- **Context-Aware Human-AI Collaboration**: Rather than treating AI as a "black box", the framework elevates the clinical programmer to a strategic validator through purpose-built, context-aware UIs. Ensure that human reviewers can seamlessly interact with the AI's context. This design drastically reduces context-switching fatigue and accelerates the review-refine-approve cycle.

LIMITATIONS AND CHALLENGES

- **Protocol Derived Domains**: Domains like Subject Visits (SV) and Inclusion/Exclusion Criteria Not Met (IE) depend on protocol defined that are not fully derivable from CRF annotations. The current fra mework requires external definition of these domain frameworks;
- **External Data Handling**: Protocol deviations, pharmacokinetic data, and other external data sources may not map cleanly to standard SDTM domains. The framework does not currently address these sp ecialized mapping scenarios;
- **PDF CRF Reconstruction**: While the framework uses ALS files for accurate reconstruction, producti on environments may only have PDF exports from legacy EDC systems. Automated reconstruction fr om PDF remains challenging and may require human correction;
- **Hallucination Risks**: Despite the historical reference approach, AI may still generate plausible but in correct annotations in edge cases. Robust human review remains essential for quality assurance;

CONCLUSION

This paper has presented a comprehensive, multi-stage AI-augmented framework designed to fundamentally transform the generation of SDTM artifacts. By structuring the workflow into Annotation, Specification, and Coding stages, the proposed architecture transitions clinical data programming from a manual, labor-intensive endeavor into an automated, AI-augmented pipeline.

The core innovation of this framework lies in its hybrid design, which synergizes the semantic reasoning capabilities of LLM with strict, deterministic guardrails. By anchoring AI generation to historical references and enforcing dependency-aware execution planning to mitigates the inherent hallucination risks of generative AI. Furthermore, the Stage Controller and Standard Data Controller ensure rigorous adherence to CDISC standards, maintaining full traceability and state management across the entire data lifecycle. Crucially, this framework is built on the philosophy of human-AI collaboration rather than full autonomy. Through Explainable AI interfaces, transparent code plans, and a "defense-in-depth" review architecture, it elevates the role of the clinical programmer from a manual coder to a strategic validator.

While challenges remain—particularly in handling protocol-derived Trial Design domains, non-CRF external data, and the validation of non-deterministic outputs—the foundation established by this framework provides a clear roadmap for future innovation. As multimodal models and advanced natural language processing continue to mature, the integration of protocol extraction and automated regulatory documentation will further close the gap between clinical study design and data delivery. Ultimately, this framework represents a significant step forward in accelerating clinical trial timelines, enhancing data quality, and empowering clinical data teams to focus on high-value scientific insights rather than repetitive programming tasks.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Yuqi Chen

Akeso, Inc.

E-mail: yuqi01.chen@akesobio.com