Paper SM04

More Science, Less Bookkeeping: Our Path to Hyper-Efficient ADaM Program Generation via AI, Process Improvements, and Good Old-Fashioned Programming

Matthew Phelps, Novo Nordisk, Copenhagen, Denmark
Nicolai Skov Johnsen, Novo Nordisk, Copenhagen, Denmark

ABSTRACT

Novo Nordisk aims to spend less time programming and more time doing science by reducing our programmer-to-study ratio by 40% (from ~1.7 to 1). To get there we needed to re-conceptualize how we produce ADaM. We started by investigating current bottlenecks in ADaM production and discovered it was not the programming itself, but rather the surrounding processes (bookkeeping, validation, and coordination) causing blockages

Instead of using GenAI solutions to generate ADaM programs - which wouldn't address these bottlenecks - we developed a framework that breaks down ADaM code into pre-validated, reusable "atomic" components. These atomic ADaM components are combined with the study-specific specifications to deterministically auto-generate pre-validated ADaM programs. Additionally, by narrowing the work scope to atomic components, we've created ideal entry points for AI assistants to further amplify programmer efficiency.

Our presentation details the technical and process architecture that allowed us to implement this in one of our flagship projects.

INTRODUCTION

In the current landscape of clinical data standards implementation, ADaM programming follows a predominantly manual and individualized approach. Trial Clinical Data Standards (CDS) programmers leverage knowledge from previous studies to assemble programs by collecting and adapting code fragments to meet protocol requirements. This involves extensive manual effort without a systematic framework to guide standardization.

VARIABILITY IN IMPLEMENTATION

Coding style in ADaM programming is heavily influenced by individual programmer preferences, trial-specific conventions, and varying experience levels. Studies traditionally have had considerable agency in deciding TFLs and coding styles, limiting reusability and scalability. Consequently, ADaM programs differ substantially in implementation, and similar derivations may be coded inconsistently across studies. This lack of standardized practices reduces code readability and maintainability, creating learning curves that consume valuable time and resources.

VALIDATION BURDEN

Each new ADaM program requires comprehensive validation both of the specification and the code itself, even when substantial code is reused from other trials with minimal modifications. This process demands significant resources, including programmer time, quality control reviews, and documentation, creating a bottleneck that impedes efficiency and scalability.

RISK OF MISALIGNMENT

Maintaining alignment between ADaM programs and define.xml metadata presents an ongoing challenge. The manual process of ensuring consistency is tedious and error-prone, particularly when derivations are added or modified due to protocol amendments. Managing multiple sources of truth increases workload and introduces substantial risk of misalignment, potentially compromising data integrity and regulatory compliance.

SCALABILITY AND SUSTAINABILITY CONCERNS

The current approach does not scale effectively, as the relationship between studies and required FTEs remains essentially linear. With increasing clinical trial volumes and pressure to reduce cycle times, this model is unsustainable. Organizations must accelerate delivery while reducing operational risks.

THE BUSINESS PROBLEM

In summary, the traditional approach to ADaM programming creates four critical bottlenecks:

1. **Inefficient validation processes** – Redundant validation of reused code and specifications consume disproportionate resources
2. **Multiple sources of truth** – Disconnected programs and metadata increase complexity and risk
3. **Burdensome bookkeeping** – Manual tracking and alignment efforts divert focus from value-added activities

4. **Lack of standards** – traditionally studies have had much agency in deciding TFLs and coding styles limiting reusability and scalability.

These bottlenecks pose a significant business problem that threatens future operational sustainability. However, they also represent a compelling opportunity for automation. The inherent potential for code reuse and the possibility of automatically aligning define.xml metadata with ADaM programs make this domain an ideal candidate for standardization and automation initiatives.

A HOLISTIC SOLUTION

Addressing these challenges requires more than technical improvements alone. An effective solution must encompass both technical innovations and process-level transformations, fundamentally reimagining how ADaM programming is approached, executed, and validated. This paper presents a comprehensive framework designed to standardize and automate ADaM programming.

THE MIGHTY FRAMEWORK: AN OVERVIEW

To address the challenges outlined above, we have developed mighty, a comprehensive framework designed to standardize and automate ADaM programming. Mighty operates as a key component within a larger-scale automation initiative that aims to standardize and integrate the generation of Tables, Figures, and Listings (TFLs), SDTM (Study Data Tabulation Model), and ADaM datasets, creating a seamless and interconnected clinical data pipeline. The framework is currently under active development, with core technical capabilities operational and soon to be piloted in real-world studies.

STRATEGIC OBJECTIVES

The mighty framework is designed to achieve four primary objectives:

1. **Eliminate redundant validation** by creating a library of pre-validated, reusable components that can be applied across studies without re-validation
2. **Establish a single source of truth** by centralizing ADaM specifications and automatically generating aligned code, metadata, and define.xml from the same source
3. **Reduce bookkeeping** by automating the alignment between programs and metadata, eliminating manual tracking and synchronization efforts
4. **Enforce standardization** by providing a structured framework with standardized components that reduces variability, automates routine tasks, and enables scalability across studies

CORE PRINCIPLES AND ARCHITECTURE

MODULAR OPERATIONS ARCHITECTURE

At its core, mighty decomposes ADaM programming into basic, isolated operations expressed *as row actions* and *column actions*. This modular approach allows complex data transformations to be constructed from well-defined, reusable building blocks, promoting consistency and reducing redundancy across programs.
Row actions operate on observations (e.g., filtering, sub-setting, merging datasets), while column actions perform variable-level operations (e.g., derivations, transformations, labeling). By standardizing these fundamental operations, mighty creates a common vocabulary for ADaM programming that transcends individual coding styles.

SPECIFICATION-DRIVEN DEVELOPMENT

The ADaM specification serves as the single point of reference for all programming activities. Rather than maintaining separate documents for specifications, programs, and metadata, mighty uses the specification as the authoritative source from which all other artifacts are derived.
The ADaM specification defines what actions (both row and column operations) to apply to each ADaM domain in the trial, creating a complete blueprint for dataset construction. This declarative approach separates the "what" (which transformations to perform) from the "how" (the underlying code implementation).

ELIMINATING MULTIPLE SOURCES OF TRUTH

In traditional workflows, ADaM programs and define.xml are often maintained separately, creating opportunities for misalignment. Mighty fundamentally solves this problem by generating both artifacts from the same specification. When a derivation is added, modified, or removed, both the program code and the corresponding define.xml metadata are updated simultaneously and automatically. This ensures that what is documented in define.xml precisely reflects what is implemented in the programs.

COMPONENT LIBRARY AND GOVERNANCE

Mighty leverages a library of pre-validated standard components that encapsulate common ADaM operations—the building blocks from which ADaM programs are constructed. Actions in the ADaM specification invoke these components to perform specific transformations. Trivial actions (such as simple variable assignments) are handled directly by the code generation engine without requiring formal components.

The standard component library is currently under development, with priority given to the most used ADaM operations.

Standard Components undergo rigorous validation through comprehensive unit testing and are managed through a governance process overseeing:

- Component development and validation
- Version control and change management
- Documentation and usage guidelines
- Retirement and deprecation

The formal validation process and governance framework are still under development to establish strategies that satisfy regulatory requirements and organizational quality standards. Once validated and approved, components can be reused across multiple studies without re-validation, reducing validation burden and accelerating timelines.

Custom Components can be developed for trial-specific needs not covered by standard operations. While requiring trial-specific validation, they follow the same architectural patterns as standard components, ensuring consistency and facilitating potential future standardization.

AUTOMATED CODE GENERATION

ADaM programs are rendered automatically based on the specifications and selected components. This automation eliminates manual coding errors, ensures consistency in implementation across all programs, and accelerates program development timelines.

Since each component is validated through comprehensive unit testing, and the mighty framework will be validated when in production, the rendered ADaM programs will contain only validated code. This means that once the framework reaches full maturity, the generated programs will be composed entirely of pre-validated building blocks—both standard and custom—assembled by a validated code generation engine, significantly reducing the validation burden for individual studies.

GENERATIVE AI INTEGRATION

Mighty's vision includes incorporating generative AI as a productivity tool for generating custom and standard components within a controlled framework. The modular architecture provides ideal entry points for AI applications because components have well-defined inputs, outputs, and effects, making the generation task bounded and verifiable.

All AI-generated components undergo human review by standards maintainers and comprehensive unit testing before approval. AI serves as a productivity multiplier, not an autonomous decision-maker, operating within strict boundaries to maintain quality and governance standards.

GOVERNANCE AND OVERSIGHT

Our proposed vision for mighty includes the following roles and responsibilities, which have not yet been formally established

- Standards maintainers are responsible for creating, managing, versioning, and validating standard components
- Trial programmers are the consumers of standard components, applying them to their studies
- Trial programmers can create custom components for trial-specific needs
- Custom components that prove broadly applicable may be promoted to standard components
- Validation requirements remain unchanged—AI simply accelerates component development
- Human expertise remains central to quality assurance

This approach harnesses the efficiency gains of AI while maintaining the rigorous quality standards required in regulated clinical research. The framework's architecture makes AI application practical and safe by constraining it to well-defined, testable units rather than attempting to automate entire programs.

INTEGRATED WORKFLOW: FROM SPECIFICATION TO VALIDATED PROGRAMS

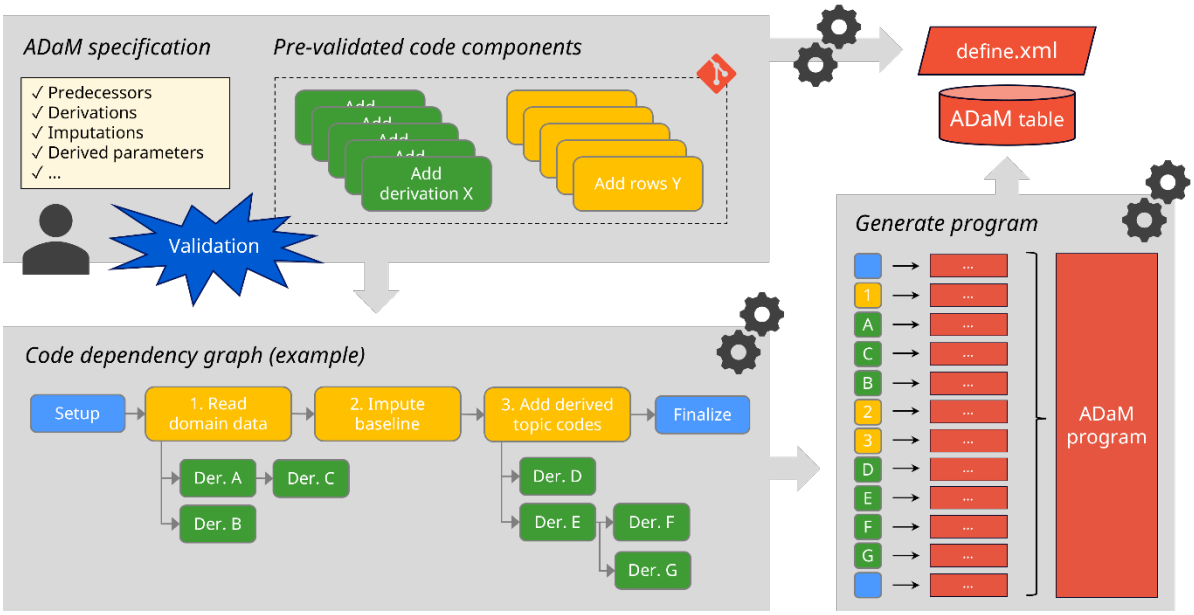

Figure 1: Mighty Framework Architecture and Workflow

The process begins with the ADaM specification, which declaratively defines what needs to be accomplished—predecessors, derivations, imputations, and derived parameters. These specifications are combined with pre-validated code components from the component library, representing both standard components (validated by standards maintainers) and custom components (validated by trial CDS programmers for trial-specific needs). The framework maps specification actions to appropriate components and organizes them into a code dependency graph, showing how individual operations are sequenced to build the complete ADaM dataset.
Through automated code generation, mighty combines the specification with selected pre-validated components to render the ADaM program. Since each component is validated and the framework itself will be validated at maturity, the rendered programs contain only validated code. In parallel, the same specification generates the define.xml, ensuring consistency between the dataset and its metadata.

IMPLEMENTING MIGHTY

The mighty framework is implemented as a suite of R packages, each addressing specific aspects of the ADaM automation workflow. This modular structure enables incremental deployment, clear separation of concerns, and flexible adoption. All packages are currently under active development and are subject to change.
The framework consists of these complementary packages:

- mighty - Core orchestration engine
- mighty.metadata – Provides the YAML data model for ADaM specifications
- mighty.component – Provides functionality for mighty to use standard and custom code components
- mighty.standards - Standard code component library with pre-built, validated derivations
- mighty.editor – A utility to provide an easier interface for trial programmers to write ADaM specifications in
- mighty.ai – an AI helper to assist writing ADaM specifications by proposing content (datasets, variables, and parameters for BDS) and ID specific code components to use based on the content
- mighty.toolbox – A utility to translate ADaM specifications into the define.xml regulatory deliverable

TOPOLOGY ANALYSIS AND EXECUTION ORCHESTRATION

A key technical capability of the mighty framework is its topology analysis engine, which automatically determines the optimal execution order of actions specified in ADaM specifications. This analysis operates at two distinct levels, completely removing the burden of program organization and execution sequencing from trial CDS programmers.

ACTION-LEVEL ANALYSIS

Each action in an ADaM specification describes both what it produces (output columns or rows) and what it depends on (required input columns and preceding row operations). Using this dependency information, mighty's

topology analysis determines the correct execution order of actions within each ADaM domain, ensuring that all prerequisites are satisfied before an action is executed.

PROGRAM-LEVEL ANALYSIS

Beyond organizing actions within a domain, the topology analysis also identifies cross-domain dependencies and organizes actions into ADaM programs. The framework groups actions into programs based on their dependencies, determines the appropriate execution order across programs, identifies when update programs are needed (e.g., if some ADSL actions depend on outputs from another domain, mighty creates ADSL update programs as needed), and minimizes the total number of programs while strictly adhering to all dependency constraints.

INTELLIGENT MISSING DATA HANDLING

The mighty framework includes missing data detection that enables partial program execution during trial conduct when not all source data may be available yet.

When an ADaM specification references columns from SDTM or other source tables that are not yet available, mighty's dependency tracking system identifies all actions that cannot execute due to missing input data and traces downstream dependencies across ADaM domains. Rather than failing, the framework automatically comments out sections of the rendered program that cannot execute, following the dependency chain to ensure consistency. The resulting program remains fully executable, with only unavailable actions temporarily disabled.

This capability provides significant flexibility:

- ADaM programs can be executed even when source data collection is incomplete
- Partial datasets can be generated for available data without manual code modification
- As new source data becomes available, the framework automatically activates previously commented sections
- Trial teams can iteratively build and test ADaM datasets as the trial progresses

This intelligent handling transforms ADaM programming from an all-or-nothing process into an incremental workflow that aligns with the realities of clinical trial data collection.

EXAMPLES

We present two examples illustrating mighty's topology analysis: a simple single-domain specification and a complex multi-domain scenario with cross-domain dependencies. The rendered program formatting and inline comments are still being refined.

EXAMPLE 1: SIMPLE ADSL PROGRAM WITH DEPENDENCY CHAIN

The ADSL specification (Appendix A) includes:

- Predecessors: USUBJID, STUDYID, ARM, AGE from SDTM domains DM and DM_VACCINE
- Derivations:
  - PLANNED_ARM (from ARM)
  - AGE_GRP1 (depends on AGE)
  - AGE_CAT1 (depends on AGE_GRP1)

Mighty automatically:

1. Detects dependencies from standard component metadata
2. Constructs the topological dependency graph
3. Renders a single program with actions in correct execution order

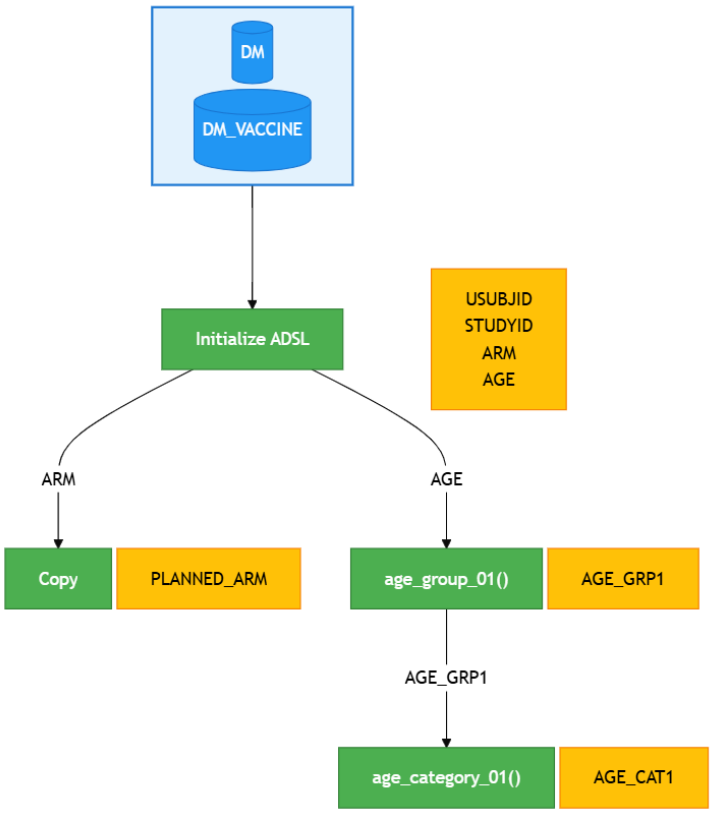

Figure 1: Internal dependency graph constructed by mighty for Example 1

The rendered program executes initialization, then derives columns in dependency order. Mighty constructs this dependency graph from the specification:

EXAMPLE 2: CROSS-DOMAIN DEPENDENCIES WITH UPDATE PROGRAMS

The specification (Appendix B) includes:

- ADSL: Extends Example 1 by adding MIN_AVAL, which requires minimum AVAL from ADLB
- ADLB: Derives AVAL from LBSTRESN in the LB domain

This creates a circular dependency:

- ADLB typically requires ADSL to exist first
- MIN_AVAL in ADSL requires ADLB to exist

Mighty resolves this by generating three programs:

1. 1_ADSL: Creates initial ADSL without MIN_AVAL
2. 2_ADLB: Creates ADLB (can reference ADSL if needed)
3. 3_ADSL: Updates ADSL to add MIN_AVAL by reading from ADLB

Mighty constructs this dependency graph from the specification:

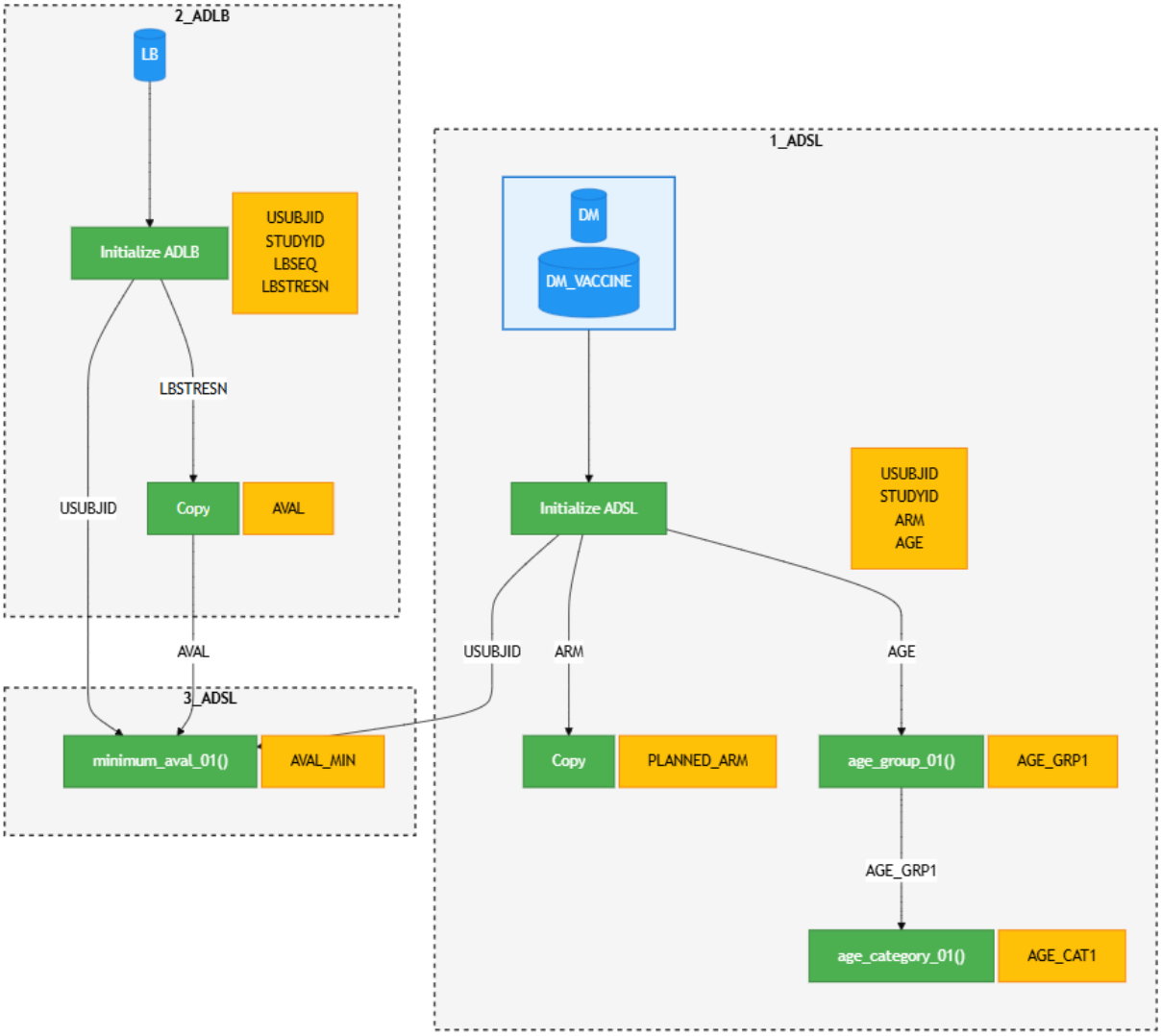

Figure 2: Internal dependency graph constructed by mighty for Example 2

CONCLUSION

Mighty represents a paradigm shift in clinical trial data programming by automating dependency resolution and program generation. Through topology analysis, it eliminates the manual effort traditionally required to determine execution order, manage cross-domain dependencies, and organize multi-program workflows.

The framework's key contributions include:

1. Automatic dependency resolution through component metadata and topological sorting
2. Intelligent program organization that generates update programs when cross-domain dependencies exist
3. Separation of concerns between specifications ("what to compute") and standard components ("how to compute")
4. Elimination of manual orchestration for complex dependency chains across ADaM domains
5. Single source of truth through specifications that serve as the authoritative reference for generating both ADaM programs and Define-XML metadata

By leveraging standard components with declared dependencies, trial CDS programmers focus on defining analysis requirements rather than managing execution logistics. The system automatically handles scenarios requiring careful manual planning in traditional approaches, from simple dependency chains to complex circular dependencies requiring multi-stage execution.

Mighty's specification-driven approach creates a unified reference point from which both executable programs and regulatory metadata are derived, ensuring consistency between implementation and documentation while reducing redundancy and potential discrepancies.

As clinical trials grow in complexity, automation frameworks like mighty become increasingly essential. The approach demonstrated here—combining declarative specifications with automatic topology analysis—provides a scalable foundation for efficient, reproducible, and maintainable clinical data programming workflows.

CONTACT INFORMATION

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

Author Name: Matthew Phelps
Company: Novo Nordisk
Address: Vandtårnsvej 114 Søborg, Denmark
Work Phone: +45 4444 8888
Email: mewp@novonordisk.com

Author Name: Nicolai Skov Johnsen
Company: Novo Nordisk
Address: Vandtårnsvej 114 Søborg, Denmark
Work Phone: +45 4444 8888
Email: nosj@novonordisk.com

APPENDIX A: SIMPLE ADSL EXAMPLE

ADSL SPECIFICATION

```
table:
  name: ADSL

init:
  base_domains:
    - dm
    - dm_vaccine
  filter_domain:
    - dm: '!is.na(AGE)'
    - dm_vaccine: NA
  filter_depend_cols:
    - AGE

column_action:
  USUBJID:
  STUDYID:
  ARM:
  PLANNED_ARM:
    source: ARM
  AGE:
  AGE_GRP1:
    code_id: age_group_01
  AGE_CAT1:
    code_id: age_category_01
```

APPENDIX B: COMPLEX MULTI-DOMAIN EXAMPLE

ADSL SPECIFICATION

```
table:
  name: ADSL

init:
  base_domains:
    - dm
    - dm_vaccine
  filter_domain:
    - dm: '!is.na(AGE)'
    - dm_vaccine: NA
  filter_depend_cols:
    - AGE

column_action:
  USUBJID:
  STUDYID:
  ARM:
  PLANNED_ARM:
    source: ARM
  AGE:
  AGE_GRP1:
    code_id: "components/age_group_01.R"
  AGE_CAT1:
    code_id: "components/age_category_01.R"
  MIN_AVAL:
    code_id: "components/minimum_aval_01.R"
```

ADLB SPECIFICATION

```
table:
  name: ADLB

init:
  base_domains:
  - lb
  filter_global:
  - '!is.na(LBSTRESN)'
  filter_depend_cols:
  - LBSTRESN

column_action:
  USUBJID:
  STUDYID:
  LBSEQ:
  LBSTRESN:
  AVAL:
    source: LBSTRESN
```

RENDERED PROGRAM: 1_ADSL

```
# ADSL-1-read_data ------------------------------------------------------------
cnt <- connector::connect(config = './_connector.yml')
  dm <-   cnt$sdtm$read_cnt('dm') |>
  dplyr::select(AGE, ARM, STUDYID, USUBJID)
  dm_vaccine <-   cnt$sdtm$read_cnt('dm_vaccine') |>
  dplyr::select(AGE, ARM, STUDYID, USUBJID)

# ADSL-init_domain ------------------------------------------------------------
dm <-  dm |> dplyr::mutate(SRC_ = "dm")
dm_vaccine <-  dm_vaccine |> dplyr::mutate(SRC_ = "dm_vaccine")

ADSL <-   rbind(dm, dm_vaccine) |>
    dplyr::select(AGE,ARM,STUDYID,USUBJID,SRC_) |>
    admiral::convert_blanks_to_na()

# ADSL-filter_domain ----------------------------------------------------------
ADSL <-   ADSL |>
  dplyr::filter((SRC_ == 'dm' & !is.na(AGE)) | (SRC_ == 'dm_vaccine')) |>
  dplyr::select(-SRC_) |>
  dplyr::select(AGE, ARM, STUDYID, USUBJID)

# ADSL-AGE_GRP1 ---------------------------------------------------------------
  ADSL <-   ADSL |>
    dplyr::mutate(
      AGE_GRP1 = cut(
        AGE,
        breaks = c(-Inf, 18, 65, 75, Inf),
        labels = c("< 18 years", "18 - 64 years", "65 - 74 years", ">= 75 years"),
        right = FALSE
      )
    )

# ADSL-AGE_CAT1 ---------------------------------------------------------------
  ADSL <-   ADSL |> dplyr::mutate(
    AGE_CAT1 = dplyr::case_when(
      AGE_GRP1 %in% c("< 18 years", ">= 75 years") ~ "Outside expected range",
      TRUE ~ "Inside expected range"
    )
  )

# ADSL-PLANNED_ARM ------------------------------------------------------------
ADSL <-  ADSL |> dplyr::mutate(PLANNED_ARM = ARM)
```

```
# ADSL-1-write_data ------------------------------------------------------------
ADSL <-  ADSL |> dplyr::arrange(USUBJID,STUDYID)
cnt$adam$write_cnt(ADSL, "ADSL.parquet", overwrite = TRUE)
```

RENDERED PROGRAM: 2_ADLB

```
# ADLB-2-read_data -------------------------------------------------------------
cnt <- connector::connect(config = './_connector.yml')
  lb <-   cnt$sdtm$read_cnt('lb') |>
  dplyr::select(LBSEQ, LBSTRESN, STUDYID, USUBJID)

# ADLB-init_domain -------------------------------------------------------------
ADLB <-   lb |>
    dplyr::select(LBSEQ,LBSTRESN,STUDYID,USUBJID) |>
    admiral::convert_blanks_to_na()

# ADLB-filter_domain -----------------------------------------------------------
ADLB <-   ADLB |> dplyr::filter(!is.na(LBSTRESN)) |>
  dplyr::select(LBSEQ, LBSTRESN, STUDYID, USUBJID)

# ADLB-AVAL --------------------------------------------------------------------
ADLB <-  ADLB |> dplyr::mutate(AVAL = LBSTRESN)

# ADLB-2-write_data ------------------------------------------------------------
ADLB <-  ADLB |> dplyr::arrange(USUBJID,STUDYID,LBSEQ)
ADLB <-  ADLB |> dplyr::select(USUBJID, STUDYID, LBSEQ, LBSTRESN, AVAL)
cnt$adam$write_cnt(ADLB, "ADLB.parquet", overwrite = TRUE)
```

RENDERED PROGRAM: 3_ADSL

```
# ADSL-3-read_data -------------------------------------------------------------
cnt <- connector::connect(config = './_connector.yml')
  ADSL <-   cnt$adam$read_cnt('ADSL')
  ADLB <-  cnt$adam$read_cnt('ADLB') |>
  dplyr::select(AVAL, USUBJID)

# ADSL-MIN_AVAL ----------------------------------------------------------------
  ADSL <-  ADSL |>
    dplyr::left_join(
      ADLB |> dplyr::select(USUBJID, AVAL) |>
        dplyr::group_by(USUBJID) |>
        dplyr::summarise(MIN_AVAL = min(AVAL)) |>
        dplyr::ungroup(),
      by = "USUBJID"
    )

# ADSL-3-write_data ------------------------------------------------------------
ADSL <-  ADSL |> dplyr::arrange(USUBJID,STUDYID)
ADSL <-  ADSL |> dplyr::select(USUBJID, STUDYID, ARM, PLANNED_ARM,
                               AGE, AGE_GRP1, AGE_CAT1, MIN_AVAL)
cnt$adam$write_cnt(ADSL, "ADSL.parquet", overwrite = TRUE)
```