Paper AD11

Automation of the CDISC Pilot study – a step-by-step guide.

Stuart Malcolm, Veramed, UK

ABSTRACT

This paper describes a systematic approach to metadata-driven automation of clinical trial data pipelines based on ARCTICA (Automation of Regulated Clinical Trial Component Architecture) and using the CDISC Pilot study as an example.

ARCTICA's goal is to define a standard make it easy to create metadata-driven automation of clinical data pipelines, and to work with existing CDISC standards, IT systems and operational procedures.

The purpose of this paper is to provide an overview of the workflow involved in automating a clinical trial data pipeline through a series of practical examples, with the aim of highlighting the similarities and differences between an automation project and a traditional manual procedure.

INTRODUCTION

The CDISC 360 project vision of standards-based end-to-end automation promises "substantially improved efficiency, consistency, and re-usability across the clinical research data lifecycle", however there are significant challenges to achieve this – not least, how to manage metadata in a way that both provides efficiencies of scale while at the same time maintaining the flexibility and 'low-level' control which the current manual programming approach.

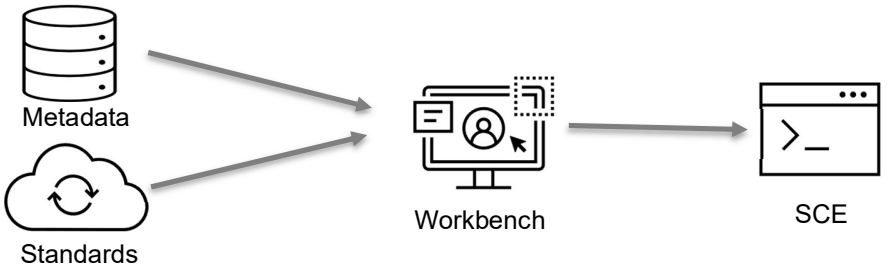
ARCTICA addresses this by treating metadata as a 'first-class citizen' – text files that can be edited and versioned, and like programming code, have syntax and grammar that can be validated in the editor up-front, and used to generate executable code.

To demonstrate this approach, this paper walks through the steps to automate the creation of the CDISC Pilot study based on the ARCTICA component architecture. The CDISC Pilot Project was a collaborative effort between industry and the FDA to create a submission package of CDISC-adherent datasets and associated metadata [1].

AUTOMATION COMPONENTS

ARCTICA defines a 'workbench' which provides an interface that enables the user to:

- Import external metadata (e.g. from CDISC Library, an MDR, or Excel spreadsheet, etc.),
- Define a clinical data pipeline consisting of inputs, programs and outputs,
- Transform metadata to build specifications that drive the code generation,
- Export the clinical data pipeline to the SCE runtime environment,
- Validate the execution of the pipeline in the runtime environment



Key ARCTICA components are shown in Figure 1 below. This paper describes the use of some of these components – a detailed description of each one is outside the scope of this paper. See the 2022 CDISC US Interchange presentation and paper for more details [3]

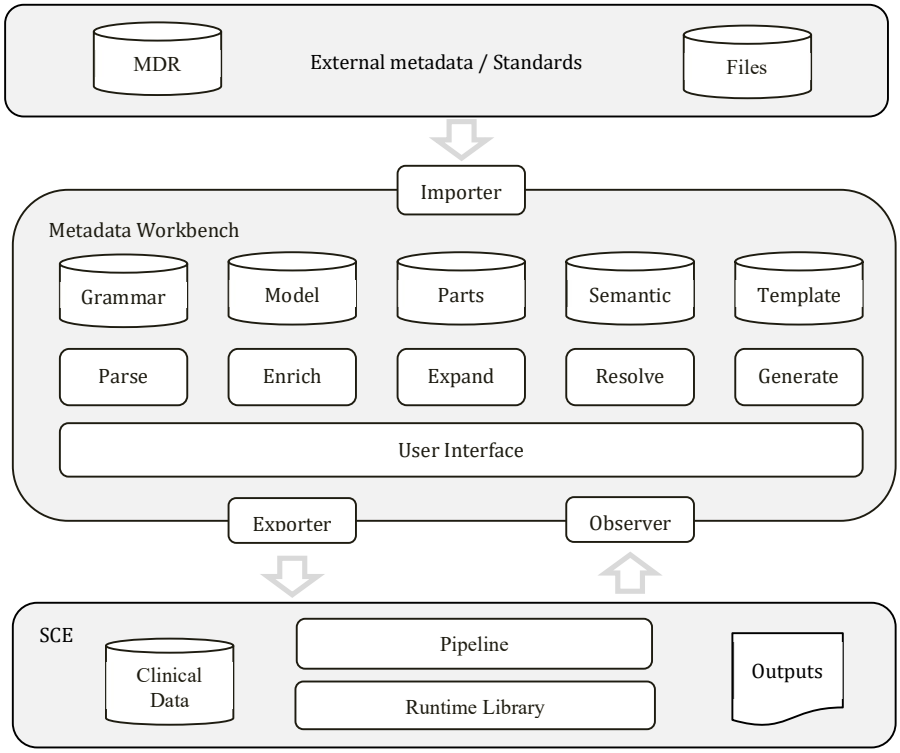


Figure 1 Overview of ARCTICA component architecture

HIGH-LEVEL WORKFLOW

At a high-level, the workflow for automating the creation of a clinical data pipeline is very similar to the manual workflow. The difference is that that for automation, specifications must be machine-readable and much of the code that runs the pipeline is generated rather than manually created.

The following diagram describes the automation workflow, covering both creation and validation phases.

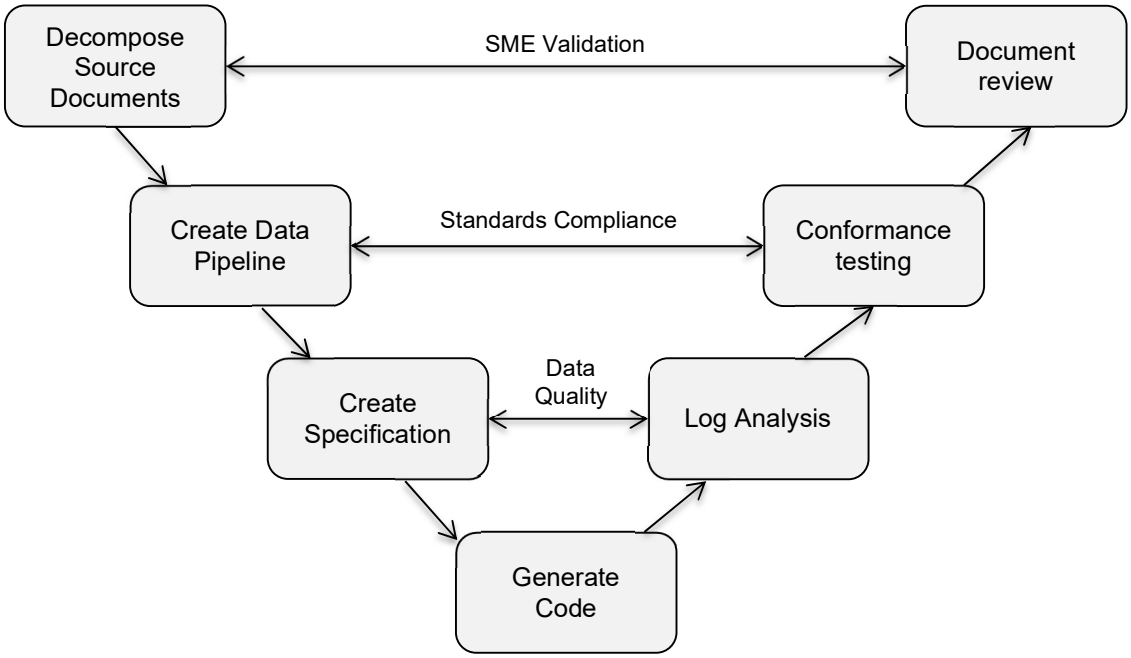


Figure 2 Methodology for creation and validation of automation projects

DECOMPOSE SOURCE DOCUMENTS INTO SPECIFICATIONS

In an ideal world, our source documentation such as Protocols, SAP, etc. would be created in a machine-readable format and easily ingested into our automation system. However, currently this is not often the case, and source documents are created manually e.g. in Microsoft Office. For this reason, the ARCTICA 'importer' component defines a common interface that allows a range of data types – PDF, Excel, XML, JSON, etc. to be imported from any location – local files, remote repositories, REST API, etc.

The 'decompose source documents' phase of an automation project serves two purposes:

1) Annotated source documents can be converted into machine-readable formats; imported into the automation system; and used for traceability
2) The annotated source documents can be used to validate that there is a common understanding between upstream subject matter experts (e.g., Protocol and/or SAP authors) and the study team implementing the automation.

In the rest of this section, we consider the following use-cases:

- Analysis populations. These are annotated in the SAP, reviewed by the document author and then the annotations are imported as useable metadata.
- Assessment Windows are defined in the SAP but connot be annotated in a form that will result in useable metadata, so the SAP is annotated with the location of an adjunct source document which is created in Excel and then imported as useable metadata.

IMPORTING SOURCE DOCUMENT METADATA USING ANNOTATIONS

A key part of decomposing source documents is to annotate PDF documents which are then reviewed with subject-matter experts and the annotations extracted into a machine-readable format.

The example below shows how the CDISC Pilot SAP (which is in the submission package CSR) is annotated with the population flags, which are then extracted to create a metadata file.

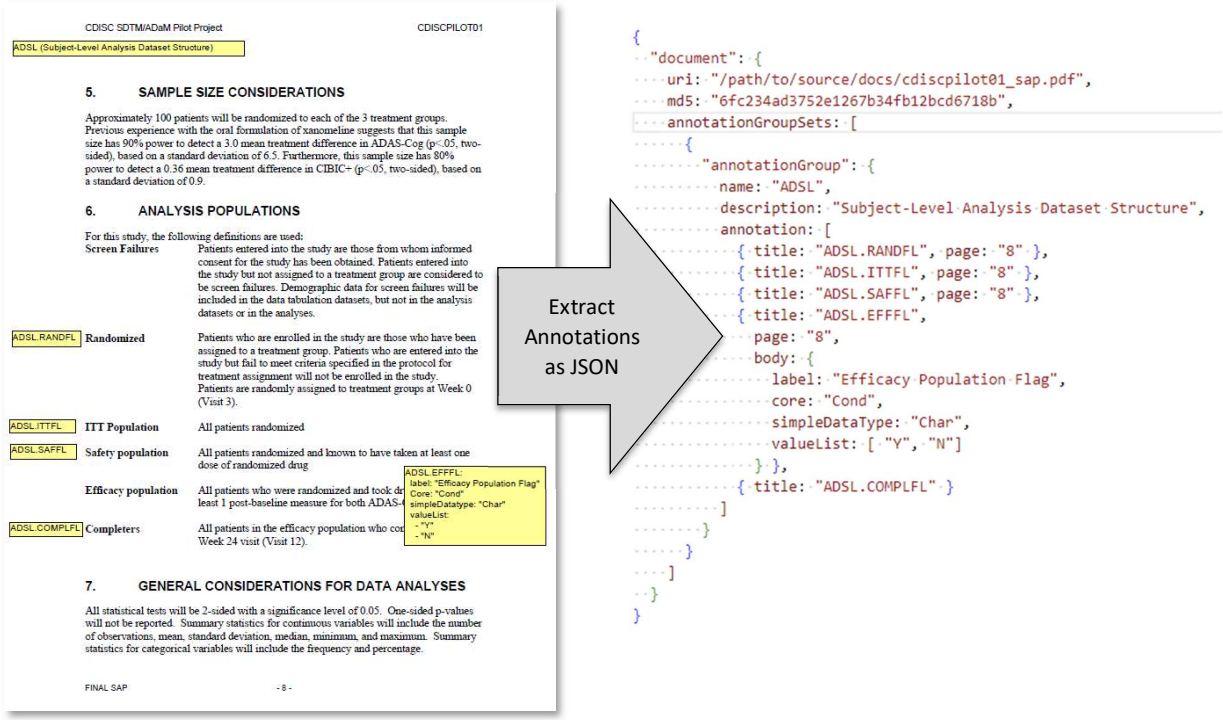


Figure 3 Structured annotations on source documents extracted as JSON

Key points to note about the PDF file above:

- The annotated SAP (aSAP) follows a similar convention as annotated CRF (aCRF) in terms of color and position of annotations
- At the top of the page is an annotation that follows a NAME (DESCRIPTION) format which describes the type of annotations (which are grouped together using the same color). This is based on the CDISC MSG v2 aCRF guidelines.
- Note that the EFFFL annotation contains additional information in YAML format which is both human and machine readable.

Key points to note about the JSON file:

- The location and md5 hash for the source document are included in the JSON. This allows full traceability back to the correct version of the source document and can be used to ensure the source document has not changed since the JSON was generated.
- The annotations are grouped using the annotation extracted from the top of the page. This is identified as the grouping annotation because it matches the NAME (DESCRIPTION) format. The name and description are parsed from the annotation and appear in the JSON as separate elements in the 'annotationGroup' object.
- Each annotation in the group contains the PDF page number that the annotation is on, which can be used for CDISC traceability requirements.
- The contents of the ADSL.EFFFL annotation contain the YAML from the PDF which has been imported as a JSON object which means the annotation contents are machine readable.

IMPORTING SOURCE DOCUMENT METADATA THAT NEEDS MORE THAN ANNOTATIONS

There will be many occasions where the source documentation contains metadata in a format that cannot be reliably imported directly from the source document using annotations – typical examples include the Study Design in a Protocol, Assessment Windows in a SAP, Adjudication data in a Data Transfer Specification, etc.

Generally, dealing with these types of sources will involve a manual step where the metadata is copied/pasted or re-entered into a sperate document – referred to as an adjunct source document - which is then imported into the automation system.

With this approach, key requirements are:

1) We must maintain end-to-end traceability from the adjunct document back to the original source,
2) We need to be able to verify consistent versioning between the original and adjunct documents – if the original source document changes, then we want to be able to at least detect that the adjunct document is out of date.

This process is shown in figure below, using the example of the CDISC Pilot assessment window definitions in the SAP. These are defined in section 8.2 on Page 10 of the SAP and are in a table. To expedite this, we start by annotating the SAP to indicate that the assessment window table is reference data, and add YAML annotation that it is reference data called 'windows' in an Excel file called Assessments.xlsx.
When we import the annotations, we get a JSON file, however the Body JSON object contains an md5 hash property which is generated for all annotations that have a 'file' property. This is the hash for the current version of the assessments.xls file, and uniquely identifies this version of the source document.

ARCTICA requires that all adjunct source documents must have a defined schema – this is to ensure they can be validated, and to enable mapping to internal metadata formats. In this instance, the annotations.xlsx file has a sheet called 'Schema' which formally defines the structure of the 'Windows' sheet, which is used to convert the sheet to JSON.

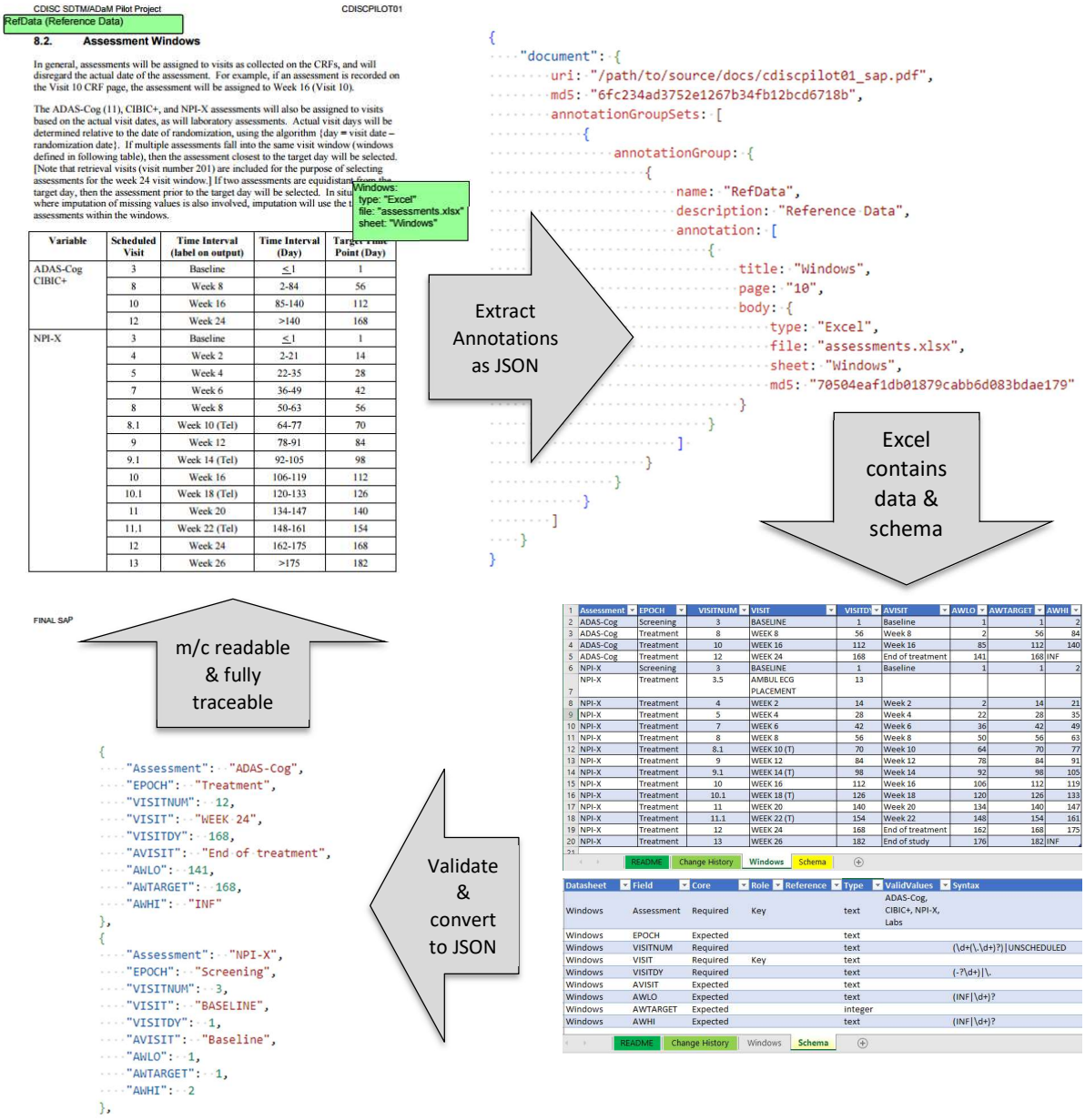


Figure 4 Using adjunct Excel workbooks and maintaining traceability back to original source document

It is also worth noting that the Assessments.xlsx file contains an enriched version of the original SAP table. In the Assessments.xlsx file, the Windows sheet uses same EPOCH, VISITNUM and VISIT values that are used in the SDTM.TV domain – this will make it easier to automate the creation of visit windows in the ADaM code, which shows the flexibility and power of adjunct source documents, but also why they need to be checked and validated.

IMPORTING MACHINE-READABLE SOURCE DOCUMENTS

Clearly, the gold-standard is that source documents are already in a machine-readable format.

The ARCTICA requirement that external metadata must have a schema applies to machine-readable sources too. If the source does not come with a schema, then one will need to be created manually and treated as an adjunct source as described above.

Typically REST API-based metadata sources, such as CDISC Library, Open Study Builder MDR, etc. provide swagger, or similar, which can be used to validate source metadata.

LOCATION OF SOURCE DOCUMENTS

Source metadata can be imported from many sources, which are independent of the format. Typical interfaces supported include:

- Individual files on disk,
- Directory tree containing multiple files,
- REST API (e.g. CDISC Library, MDR, etc.),
- Git repository (e.g. GitHub, GitLab, etc.)
-

CREATE A DATA PIPELINE

The definition of the clinical data pipeline is at the heart of how ARCTICA enables metadata-driven automation. By using a using a domain-specific language (DSL) we can describe how the source documents are imported, which data standards to use, and the programs that transform the input data to create outputs.

A DSL has the advantage of syntax checking, highlighting, code completion, etc. for the metadata text files which can be version controlled, and a DSL can be parsed and used for standard language 'backend' functions such as importing/exporting definitions from other files, and code-generation.

The pipeline description language is based on an Xtext grammar [7] with an underlying meta-model that makes the transformation and manipulation of metadata possible.

An example of ARCTICA pipeline description is shown below, where we have three pipelines – one that imports the CDISC standards from the CDISC Library using the REST API, the second which is basically a repository for the SDTM datasets and define.xml, and a third which imports from the other pipelines to create an ADSL dataset that conforms to the ADAM standard.

Some key points worth noting about the pipeline definitions:

- A pipeline definition contains definitions of resources and processes
- A resource is a (meta)data container. There are different types of resource. Currently defined: files, folders (i.e. a directory containing files), REST API, and Git repository
- Resources can only be written to by a process in the same pipeline, and a pipeline can import resources from another pipeline, in which case they will be read-only
- Typically resources are defined with their schema using the 'conforms_to' attribute. This is a reference to another resource which contains the schema.
- Resources are defined using URL attribute, although the full path to the resource is not needed and ARCTICA will locate the resource within the pipeline. Within ARCTICA, a resource can be located using the syntax: `<pipline>.<resource>[<target>]` so for example `sdtm.datasets[DM]` identifies the 'DM' target (i.e. file) in the 'datasets' resource in the 'sdtm' pipeline.

```
/*
 * Define a pipeline that is used as a local standards
 * repository, for this demo, we import CDISC Library
 *
 */
pipeline standards {
   /*
    * CDISC Library interface schema uses Swagger
    */
   resource CDISC_API_Schema {
      uri: 'CDISC1-cdisc-library-2.0-swagger.json'
      type: file
   }
   /*
    * The REST interface to CDISC Library which
    * conforms to the API interface defined above
    */
   resource CDISC_Library {
      uri: 'https://library.cdisc.org/api'
      conforms_to: CDISC_API_Schema
      type: rest
      credentials: "~/.arctica/keys"
   }
   /*
    * Import the CDISC library contents (all of it) from
    * the REST API into a local folder resource.
    */
   process import_cdisc {
      input: CDISC_Library
      output: CDISC
      template: 'import_rest_all.ps1.jinja2' // import everything!
   }
   /*
    * Local cache of everything in the CDISC Library
    */
   resource CDISC {
      extension: 'json'
      type: folder
   }
}
```

```
/*
 * SDTM pipeline (Data Store)
 */
pipeline sdtm {
    resource define {
        uri: "define.xml"
        type: file
    }
    resource datasets {
        type: folder
        extension: "xpt"
        conforms_to: define
    }
}
/*
 * ADAM pipeline that creates ADSL
 */
pipeline adam {
    /*
     * INPUTS
     */
    import sdtm.*
    import specification.adam
    /*
     * PROCESS
     */
    process adsl {
        input: sdtm.datasets
        output: adam.datasets[adsl]
        template: "adam/adsl.sas.jinja"
        conforms_to: specification.adam[adsl]
    }
    /*
     * OUTPUT
     */
    resource datasets {
        extension: "sas7bdat"
        type: folder
        conforms_to: standards.CDISC[adamig_1_3]
    }
}
```

Figure 5 Example ARCTICA pipelines definitions

Let's consider the pipeline definitions above.

The 'standards' pipeline (on the left) starts by defining a resource (CDISC_Library_Schema) which is a JSON file that is the Swagger definition of version 2.0 of the CDISC Library API. This is supplied and maintained by CDISC. The next resource (CDISC_Library) is a definition of the REST API that we are going to use to access the library, which conforms to the CDISC_Library_Schema that we previously defined.
Next, we have a process defined (import_CDISC) which generates a PowerShell script that when run will import everything from the CDISC Library and write it into the CDISC resource. The template defines the program that is generated – templates will be covered later in this paper.

The 'sdtm' pipeline (top right) does not contain any processes – it is used as a repository that contains the SDTM data and define.xml from the CDISC Pilot submission package. This is analogous to receiving SDTM transfer from a vendor.

The 'adam' pipeline starts by importing the everything (datasets and define.xml) from the SDTM pipeline, and also imports the ADaM specification (Specification pipeline not included in this example). The ADaM pipeline then defines a process which generates the ADSL SAS program using the adsl.sas.jinja template (so will generate a SAS program) which conforms to the ADaM specification.

GENERATE CODE

Metadata-driven code generation is the key to automation of clinical data pipelines, and the key to simplifying code generation is to ensure that the metadata is in a suitable format in the first place – partly this can be done during the 'specification creation' process that has been discussed above, and partly it is ensuring that the metadata is in a suitable tree structure.

The input metadata structure should implement a Document Object Model (DOM) structure [8] – note that while the original internal specification metadata should be normalized to eliminate duplication, the code-generation DOM is likely to have all the 'relations' unwound and so contain duplication – this will make the code generation easier.

Code generation is implemented by walking the DOM tree and using a code-generation templating system such as Jinja2 [6]. The minimum amount of logic possible should be included in code templates to avoid a template maintenance nightmare!

Below is an example of a TFL code generation template with input metadata DOM being transformed into SAS source code. This approach can be used to generate any target language – SAS, R, Python, makefiles, configuration files, or can even be used to transform metadata!

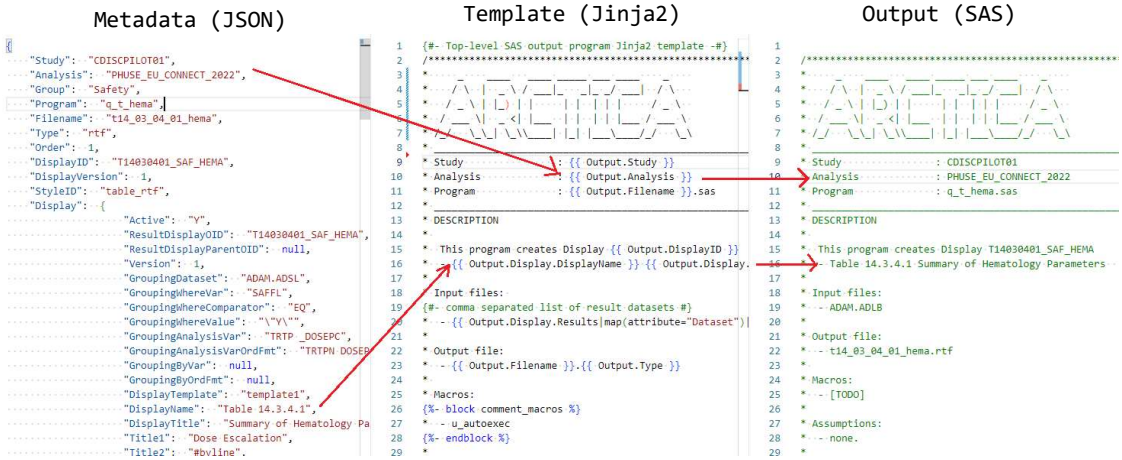


Figure 6 Code generation by substitution of JSON values into templates

Code generation templates can reformat the metadata to the requirements of the output language – for example, changing case, or splitting e.g. 'ADAM.ADSL' text in the metadata into component 'ADAM' and 'ADSL' parts, which allows the metadata to be decoupled from any specific output language.

Also, templates can be used to iterate over metadata – for example, the template below uses a {% FOR %} comprehension to loop over all the outputs in the metadata – in this case to generate a PowerShell script that runs all the TFL programs.

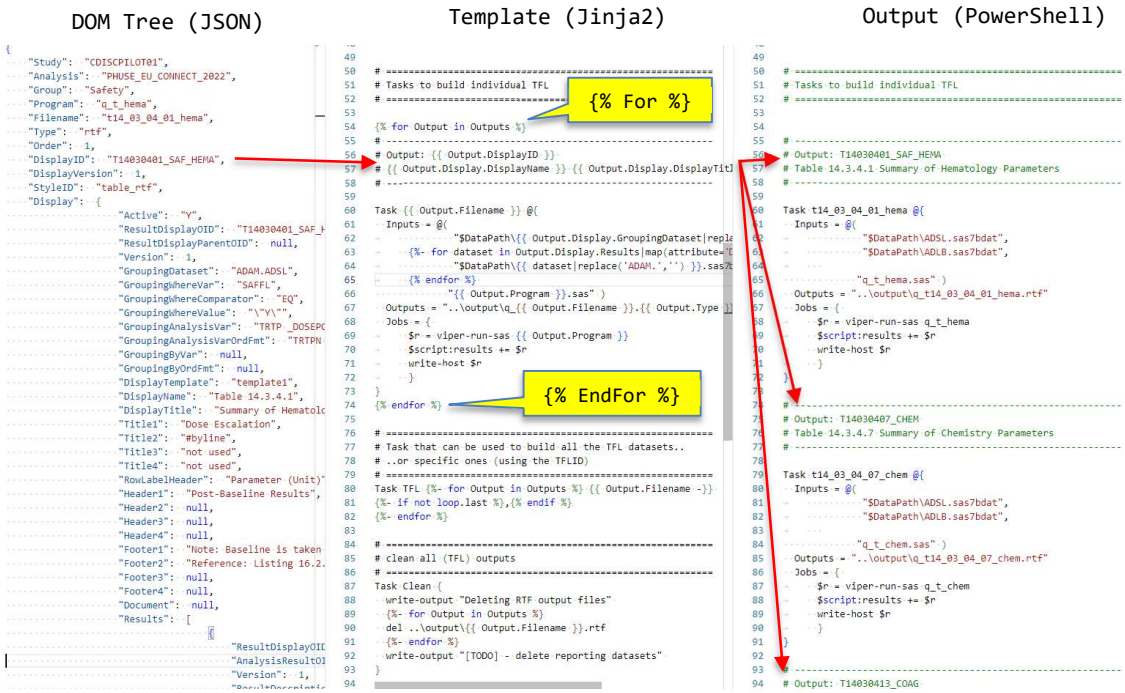


Figure 7 Code generation template looping over input metadata

CONCLUSION

One of the key challenges to achieving the benefits of end-to-end automation is the ability to manage metadata. This paper outlines a 'metadata as a first-class citizen' approach that has all the benefits of code-management (syntax checking, validation, version control, etc.) with the flexibility to import and transform metadata into a DOM-like tree structure that can drive code-generation, and maintain the traceability needed for GxP levels of validation.

REFERENCES

[1] CDISC SDTM and ADaM Pilot Study. https://github.com/cdisc-org/sdtm-adam-pilot-project
[2] ARCTICA project. https://github.com/metadatadriven/arctica
[3] CDISC EU Interchange 2022.
[4] Model-Driven Architecture. https://www.omg.org/mda/presentations.htm
[5] EMF (or link to overview of MDA technologies)
[6] Jinja2 templating engine. https://palletsprojects.com/p/jinja/
[7] Xtext DSL Framework. https://www.eclipse.org/Xtext/
[8] Document Object Model (DOM). https://www.w3.org/TR/WD-DOM/introduction.html

CONTACT INFORMATION

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

Stuart Malcolm
Head of Standards, Efficiency and Automation,
Veramed Ltd.
Email: stuart.malcolm@veramed.co.uk
Linkedin: https://www.linkedin.com/in/stuart-malcolm