Paper AD09

No Patients? No Problem!
Define your Study Data with Python at the CORE

John McDade, PHASTAR, Glasgow, United Kingdom

ABSTRACT

In clinical trials, access to study data is often delayed due to challenges around patient recruitment. Slow patient recruitment can challenge downstream activities which are often more time consuming such as complex ADaM and TFL creation. Test data from an EDC is often counterproductive for downstream programming as makes little clinical sense. This paper presents a python application that can generate CDISC-compliant SDTM data to allow downstream work to progress at the convenience of the study team. The application leverages a study define.xml to use study specific metadata to drive SDTM generation – or simply configure the number of patients if only generic data is required. The data generated can be in many different formats including dataset JSON and has real time validation using CDISC CORE [1], taking advantage of recent industry initiatives.

Keywords: CDISC, Dataset-JSON, SDTM, Synthetic Data, CDISC CORE, Define-XML, Data Standards, CDISC 360i

INTRODUCTION

The clinical research industry continues to evolve toward more open, interoperable data standards. The Clinical Data Interchange Standards Consortium (CDISC) has been at the forefront of this transformation, developing standards that facilitate data exchange and improve regulatory submissions. The introduction of CDISC 360i [2] represents a significant step toward comprehensive data standardization across the clinical research lifecycle. However, the adoption and validation of these standards face several challenges.

- Limited availability of real clinical data for training and development purposes
- Slow adoption of new standards due to lack of practical implementation examples
- Difficulty in testing validation rules without comprehensive test datasets
- Patient recruitment delays that impact study timelines and data availability

To address these challenges, we have developed a comprehensive synthetic data generation tool that creates CDISC-compliant datasets in multiple formats. This tool specifically targets four key areas of CDISC standards advancement:

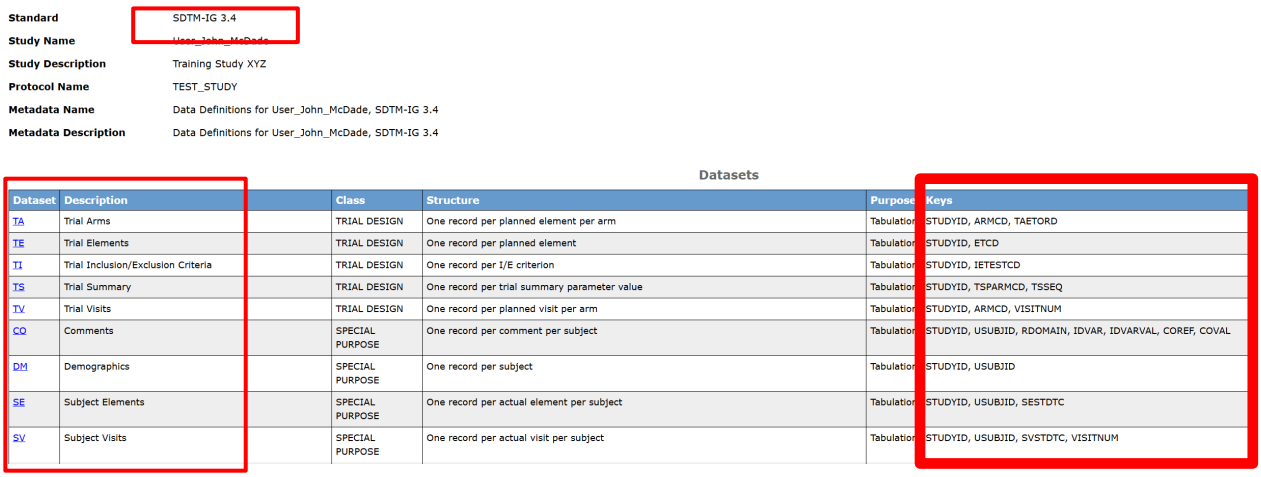
- Synthetic Data Generation: Creating realistic, compliant SDTM datasets
- Define-XML Integration: Parsing and utilizing Define-XML specifications as metadata foundation
- Dataset-JSON Implementation: Supporting the new CDISC Dataset-JSON v1.1 standard [3]
- CDISC CORE Validation: Generating test cases for rules engine development

DEFINE.XML

Define.xml is the entry point for the synthetic data generation, this is for two main reasons, first is that we can generate study specific metadata making it directly relevant to a given study. The second is that the define itself can be used in the cross validation against any generated synthetic data, giving instant feedback.

A study define.xml describes the metadata of all submission datasets, that includes trial information, domain and variable names included and any relevant codelists used. This is the core information that is being parsed for use in synthetic data generation.

Figure 1: Study and dataset level define.xml metadata.

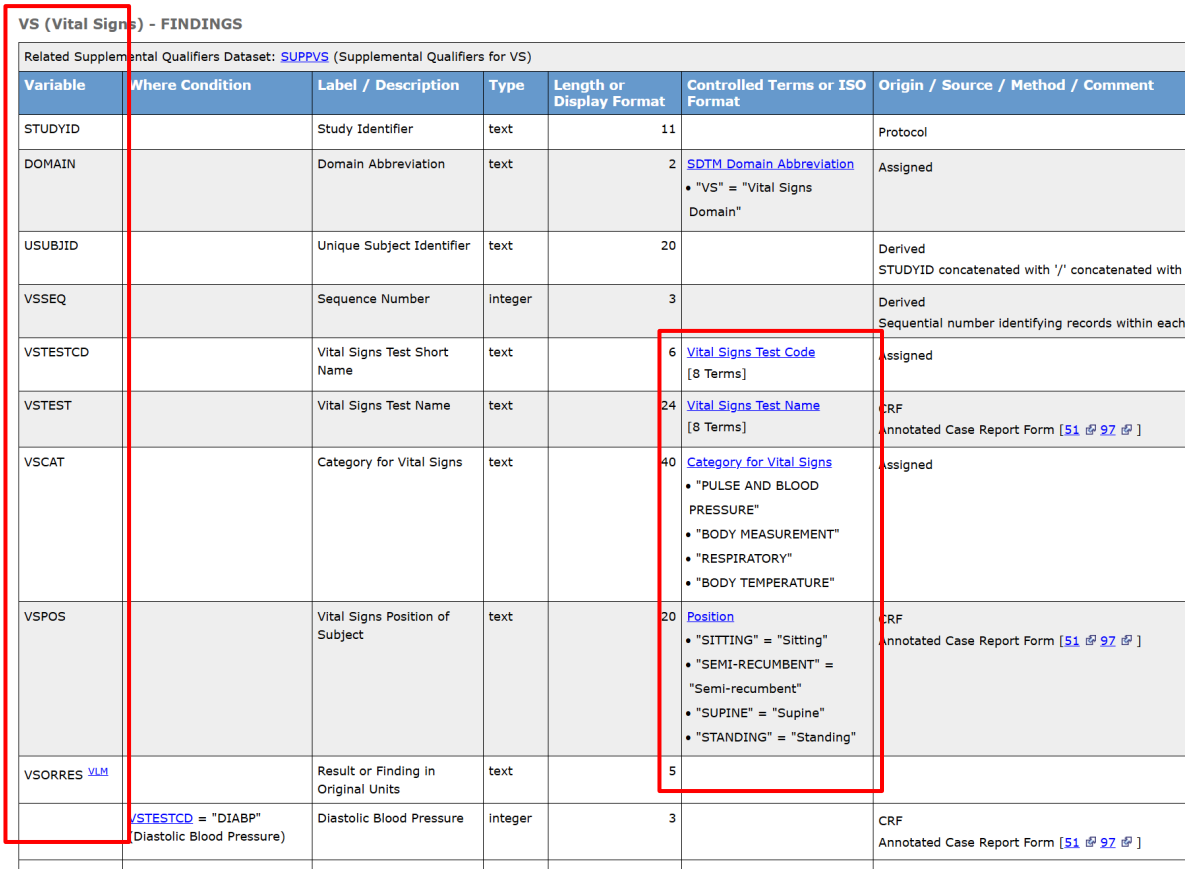


Standard	SDTM-IG 3.4
Study Name	[redacted]
Study Description	Training Study XYZ
Protocol Name	TEST_STUDY
Metadata Name	Data Definitions for User_John_McDade, SDTM-IG 3.4
Metadata Description	Data Definitions for User_John_McDade, SDTM-IG 3.4

Datasets

Dataset	Description	Class	Structure	Purpose	Keys
TA	Trial Arms	TRIAL DESIGN	One record per planned element per arm	Tabulation	STUDYID, ARMCD, TAETORD
TE	Trial Elements	TRIAL DESIGN	One record per planned element	Tabulation	STUDYID, ETCD
TI	Trial Inclusion/Exclusion Criteria	TRIAL DESIGN	One record per I/E criterion	Tabulation	STUDYID, IETESTCD
TS	Trial Summary	TRIAL DESIGN	One record per trial summary parameter value	Tabulation	STUDYID, TSPARMCD, TSSEQ
TV	Trial Visits	TRIAL DESIGN	One record per planned visit per arm	Tabulation	STUDYID, ARMCD, VISITNUM
CO	Comments	SPECIAL PURPOSE	One record per comment per subject	Tabulation	STUDYID, USUBJID, RDOMAIN, IDVAR, IDVARVAL, COREF, COVAL
DM	Demographics	SPECIAL PURPOSE	One record per subject	Tabulation	STUDYID, USUBJID
SE	Subject Elements	SPECIAL PURPOSE	One record per actual element per subject	Tabulation	STUDYID, USUBJID, SESTDTC
SV	Subject Visits	SPECIAL PURPOSE	One record per actual visit per subject	Tabulation	STUDYID, USUBJID, SVSTDTC, VISITNUM

Figure 2: Variable and codelist level define.xml metadata.



VS (Vital Signs) - FINDINGS

Related Supplemental Qualifiers Dataset: SUPPVS (Supplemental Qualifiers for VS)

Variable	Where Condition	Label / Description	Type	Length or Display Format	Controlled Terms or ISO Format	Origin / Source / Method / Comment
STUDYID		Study Identifier	text	11		Protocol
DOMAIN		Domain Abbreviation	text	2	SDTM Domain Abbreviation • "VS" = "Vital Signs Domain"	Assigned
USUBJID		Unique Subject Identifier	text	20		Derived STUDYID concatenated with '/' concatenated with
VSSEQ		Sequence Number	integer	3		Derived Sequential number identifying records within each
VSTESTCD		Vital Signs Test Short Name	text	6	Vital Signs Test Code [8 Terms]	Assigned
VSTEST		Vital Signs Test Name	text	24	Vital Signs Test Name [8 Terms]	CRF Annotated Case Report Form [51 97]
VSCAT		Category for Vital Signs	text	40	Category for Vital Signs • "PULSE AND BLOOD PRESSURE" • "BODY MEASUREMENT" • "RESPIRATORY" • "BODY TEMPERATURE"	Assigned
VSPOS		Vital Signs Position of Subject	text	20	Position • "SITTING" = "Sitting" • "SEMI-RECUMBENT" = "Semi-recumbent" • "SUPINE" = "Supine" • "STANDING" = "Standing"	CRF Annotated Case Report Form [51 97]
VSORRES VLM		Result or Finding in Original Units	text	5		
	VSTESTCD = "DIABP" (Diastolic Blood Pressure)	Diastolic Blood Pressure	integer	3		CRF Annotated Case Report Form [51 97]

One key consideration in how useful this approach relates to the timing of define.xml generation, often in the past companies might leave define generation until later stages of a study purely for the purpose of a submission. Fortunately the define is treated with more respect as more people understand the benefits of early generation, the truth is the define should be available before any programming begins. Using the define as the dataset specification means it can be used in validation against the generated data avoiding costly rework. Building many TFLs on non-compliant ADaMs which in turn have been built on non-compliant SDTMs creates a ‘house of cards’ effect that can really burn study budgets with issues filtering right across all deliverables.

Define-XML Parsing and Metadata Extraction

The Define-XML parsing implementation leverages xml.etree.ElementTree (ET) [4] Python's built-in XML parsing library provides the foundation for parsing Define-XML documents. This lightweight, efficient parser handles XML navigation using XPath expressions and namespace-aware element selection.

The parser implementation is designed to handle both Define-XML v2.0 and v2.1 specifications [5]. Both versions utilize the same ODM v1.3 [5] base structure, differing primarily in minor namespace attributes and extended features. The core metadata elements required for synthetic data generation—study information, domain definitions, variable specifications, and codelists—are structurally identical across both versions.

Figure 3: Example python parsing logic

```python
datasets[dataset_name] = []
key_sequences[dataset_name] = []
attributes[dataset_name] = {
    'domain_name': dataset_name,
    'domain_label': dataset_label,
    'variables': {}
}

domain_metadata[dataset_name] = {
    'name': dataset_name,
    'label': dataset_label,
    'oid': domain_oid,
    'structure': item_group.get('Structure', ''),
    'repeating': item_group.get('Repeating', 'No'),
    'isReferenceData': item_group.get('IsReferenceData', 'No'),
    'purpose': item_group.get('Purpose', ''),
    'variables': []
}
```

Figure 4: Sample parsed XML metadata in JSON format

Output Structure

The parser returns a comprehensive dictionary:

```json
{
  "metadata": {
    "study": {...},
    "standard": {"name": "SDTM-IG", "version": "3.4"},
    "domains": {
      "VS": {
        "name": "VS",
        "label": "Vital Signs",
        "variables": [...],
        "repeating": "Yes"
      }
    },
    "codelists": {...}
  },
  "datasets": {...},
  "key_sequences": {"VS": ["STUDYID", "USUBJID", ...]},
  "attributes": {...}
}
```

The JSON schema is optimized for synthetic data generation use cases, providing efficient access to frequently queried metadata elements. The structure separates concerns:

- metadata: Comprehensive study and domain information for documentation
- datasets: Simplified variable-codelist mappings for quick lookups

- key_sequences: Pre-sorted key variable lists for data ordering
- attributes: Domain-level and variable-level attributes for validation

SYNTHETIC DATA

Generation of synthetic data within a single column is easy across most languages. Making synthetic data meaningful across rows becomes harder and making sense across multiple domains is where the real challenge lies. With this in mind, several strategies were implemented to make compliant SDTM data.

DEMOGRAPHY SEED

One obvious way to ensure subject-level consistency across domains was to generate the DM domain first and use that as a seed for all others. DM also sets the framework for the duration of the trial: informed consent, treatment start / end, trial completion. Once we have an arbitrary date we can randomly assign each subject an Informed Consent date which will further trigger random windows for the rest of the key dates in the trial.

Figure 5: Key DM date generation

Detailed Date Calculation

Variable	Formula	Example Result	Purpose
RFICDTC	`2023-08-01 + random(-30, 30)`	`2023-07-15`	Informed Consent
RFSTDTC	`RFICDTC + 30 + random(-5, 5)`	`2023-08-14`	Study Start
RFXSTDTC	`= RFSTDTC`	`2023-08-14`	**Treatment Start (ANCHOR)**
RFXENDTC	`= RFENDTC`	`2024-06-28`	Treatment End
RFENDTC	`RFICDTC + 350 + random(-10, 10)`	`2024-06-28`	Study End
RFPENDTC	`RFICDTC + 370 + random(-10, 10)`	`2024-07-18`	End of Participation

At this point we can combine the complete the picture for DM by assigning variables like ARM / ARMCD, RACE, ETHNIC, CONTRY, SEX from the define.xml codelists. AGE can be changed on a slider to set minimum and maximum values, makes the data more realistic if trying to mimic e.g. a pediatric study. Finally, we are setting DTHFL=Y in < 10% of subjects but this can be altered.

ADVERSE EVENTS

With each subject now having a timeframe for the entire trial, we can assign them adverse events, these can obviously happen during screening / after treatment end but for simplicity AE start and end dates (AESTDTC and AEENDTC) have been assigned to during treatment so are treatment emergent.

The main configuration for adverse events is how many, this is defaulted to a random value of 0 – 10 per subject but can be altered. To make the AE data useful, we must also incorporate MedDRA coding, we cannot simply assign random values from MedDRA as they would make no sense across rows so we have to assign a logic group from the MedDRA dictionary. This ensures compliance and makes the data meaningful. Finally we must ensure all flags within AE make sense within a record but also across domains:

- If AESER=Y at least one of AESDTH, AESLIFE, AESDISAB, AESHOSP, AESCONG, AESMIE = Y
- If DM.DTHFL=Y then AESDTH=Y, AEOUT=FATAL, AESER=Y

FINDINGS DOMAINS

The SDTM class FINDINGS are our clinical trial results covering all planned assessments not limited to Vital Signs, Lab and ECG data. Similar to adverse events and the use of MedDRA, we must assign assign FINDINGS data as a logical group. As an example below from Vital Signs, we need a dictionary of all tests per domain and these will form a logical group of –TEST, --TESTCD, --ORRESU, --CAT / --SCAT (as applicable) and some range values along with decimal precision desired. We can essentially have dictionaries of the most common tests used by domain as a

starting point and build these as makes practical sense. For the LB domain, many companies will already have a formal dictionary including ranges and applicable units which can be used as an input for LB.

For practical reasons we are using SI units for both original and standard units throughout the findings domains. There is little value to be gained from setting up conversions as we will only be summarizing with the SI unit. The lower and upper ranges should be a good approximation for each test and there are various sources of this information, differences in age / gender is not being accounted in the lab ranges but is a future development.

Figure 6: Snippet from Vital Signs source dictionary

VSTESTCD	VSTEST	VSORRESU	VSCAT	Lower	Upper	DCP
SYSBP	Systolic Blood Pressure	mmHg	PULSE AND BLOOD PRESSURE	90	120	0
DIABP	Diastolic Blood Pressure	mmHg	PULSE AND BLOOD PRESSURE	60	80	0
HR	Heart Rate	beats/min	PULSE AND BLOOD PRESSURE	60	100	0
RESP	Respiratory Rate	breaths/min	RESPIRATORY	12	20	0
TEMP	Temperature	C	BODY TEMPERATURE	36.5	37.5	1
HEIGHT	Height	cm	BODY MEASUREMENT	170	185	0
WEIGHT	Weight	kg	BODY MEASUREMENT	60	90	1
BMI	Body Mass Index	kg/m2	BODY MEASUREMENT	18.5	24.9	1

A default of 5% is used to generate outliers within the data, hardcoded value shown here for clarity but this can be adjusted. The code allows a 50% chance of an outlier being below the lower range or above the higher range along with it being a proportion of the range width so the values scale for different test and ranges:

Figure 7: Python outlier code

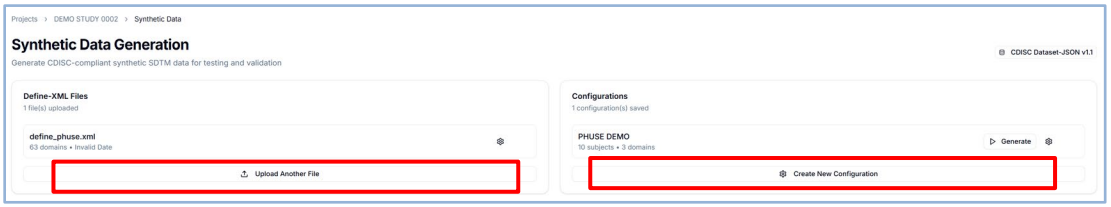
```
range_width = upper_bound - lower_bound

if random.random() < 0.05:  # 5% chance of being abnormal
    # deviation = 5-15% of range
    deviation = Decimal(random.uniform(0.05, 0.15)) * range_width

    if random.random() < 0.5:
        # High outlier
        xxorres = upper_bound + deviation
    else:
        # Low outlier
        xxorres = lower_bound - deviation
else:
    # Normal (within range)
    xxorres = Decimal(random.uniform(float(lower_bound), float(upper_bound)))
```

When the process is run, the tests (xxTESTCD values) will subset against the dictionaries described so we are generating study specific tests.

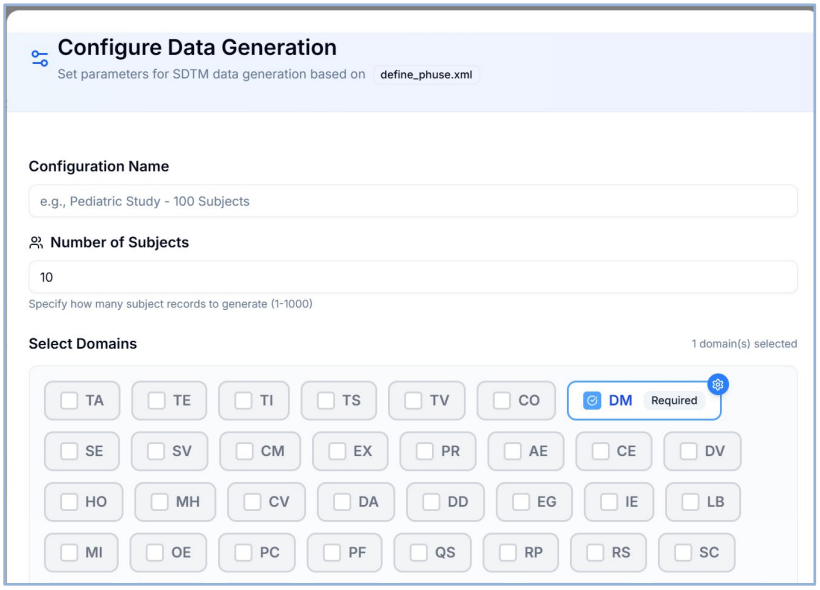
GENERATING SYNTHETIC DATA

With the framework above, we can now generate some synthetic data, the first step will be to load a define.xml file, this is uploaded via the button on the left. This will parse the metadata and be used in the configuration step which is created on the panel on the right

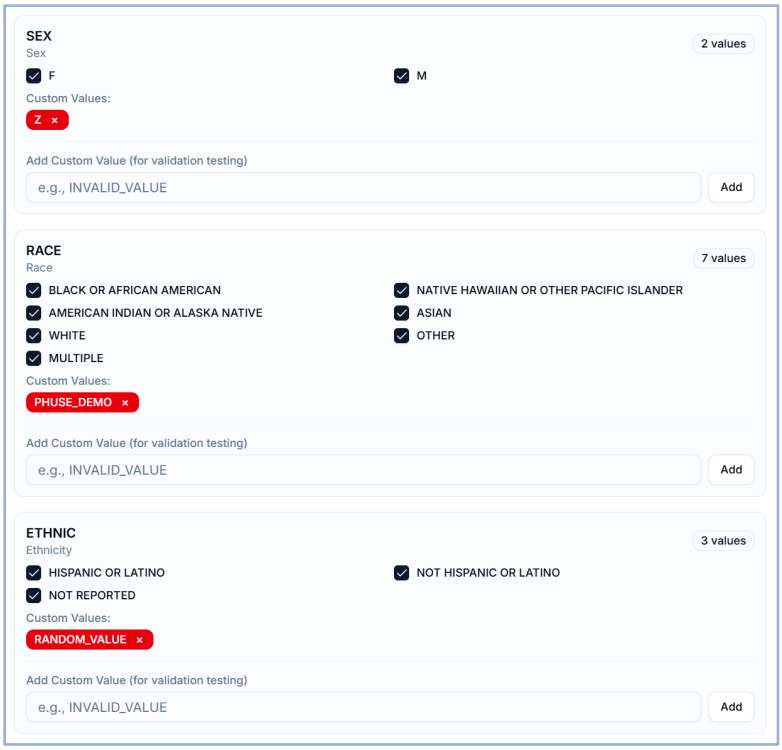
Figure 8: Synthetic Data tool frontend, loading and configuration buttons



Once a define has been loaded, the synthetic data generation can be configured, the most obvious selections will be, numbers of subjects and which domains will be available



We can also configure at a domain level to give high level of control over the generated data. We automatically have all codelists returned per variable, all values are returned by default but we can deselect if needed. We can also add any values we want, these are highlighted in red below. We will see some application of this during the CORE validation section.



Once we are happy with any customization we can save that configuration and finally generate the data. The data is returned to the UI and can be viewed directly or downloaded. We don't want to typically save the data within the tool although both JSON and NDJSON (newline delimited JSON) [6] versions of the data will persist for a study until a subsequent run is triggered. NDJSON is the preferred format to pass to CDISC CORE for speed of validation, further

details on the CDISC Dataset JSON project are provided in the references but the key consideration is that using JSON format will help enable tool connectivity and interoperability per the CDISC 360i vision.

Figure 9: Synthetic Data tool frontend and custom values injected into the data

DM Domain Preview

Showing 20 records

BRTHDTC Date/Time of Birth	AGE Age	AGEU Age Units	SEX Sex	RACE Race	ETHNIC Ethnicity
1966-08-01	57	YEARS	Z	MULTIPLE	RANDOM_VALUE
1966-08-16	57	YEARS	M	ASIAN	HISPANIC OR LATINO
1997-07-26	26	YEARS	M	OTHER	NOT REPORTED
1982-07-27	41	YEARS	Z	BLACK OR AFRICAN AMERICAN	NOT REPORTED
1970-07-20	53	YEARS	M	NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER	HISPANIC OR LATINO
1959-08-08	64	YEARS	F	WHITE	RANDOM_VALUE
1986-07-14	37	YEARS	Z	NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER	NOT REPORTED
1989-08-10	34	YEARS	F	WHITE	NOT HISPANIC OR LATINO
1961-07-19	62	YEARS	F	WHITE	NOT HISPANIC OR LATINO
1971-07-16	52	YEARS	F	ASIAN	NOT HISPANIC OR LATINO
2000-08-18	23	YEARS	M	OTHER	NOT HISPANIC OR LATINO
1978-07-20	45	YEARS	M	WHITE	NOT HISPANIC OR LATINO
1969-08-16	54	YEARS	M	AMERICAN INDIAN OR ALASKA NATIVE	HISPANIC OR LATINO
1976-07-23	47	YEARS	F	BLACK OR AFRICAN AMERICAN	HISPANIC OR LATINO
1978-08-15	45	YEARS	Z	PHUSE_DEMO	RANDOM_VALUE

CDISC CORE

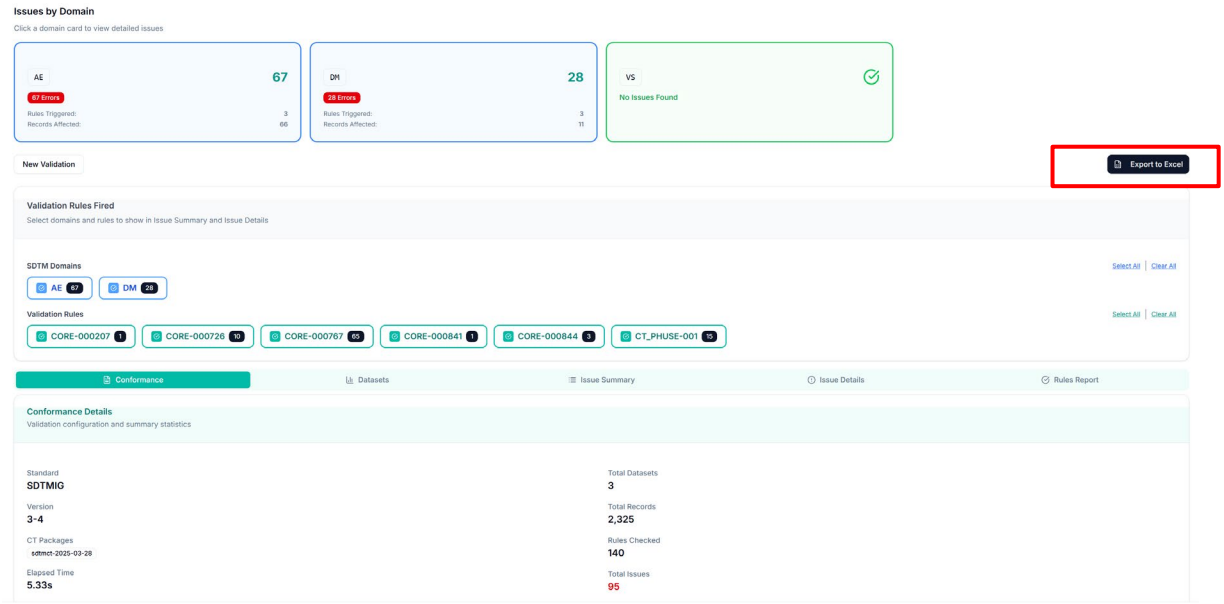
CDISC CORE (CDISC Open Rules Engine) is an open-source initiative by CDISC designed to promote transparency, consistency, and automation in the validation of clinical data against CDISC standards. Its primary goal is to provide a machine-readable, executable representation of CDISC conformance rules, enabling sponsors, regulators, and technology providers to validate study datasets in a standardized and reproducible way. By making these rules openly available and technology-neutral, CDISC CORE supports the broader objectives of improving data quality, interoperability, and regulatory readiness across the clinical research ecosystem. The platform also encourages collaboration and innovation by allowing the community to contribute, test, and refine rules as CDISC standards evolve.

You can run CDISC CORE as a compiled CLI, via Docker, or as a Python package [7], we have opted for the python package within the Sythetic Data tool. Rules and terminology are retrieved and cached from the CDISC Library API, and you can extend the cache with custom rules and custom standards. The CLI validates a folder of datasets

(commonly XPT) and can also execute a single rule against a JSON-shaped dataset for rule testing. Outputs are machine- and human-readable reports. For data interchange, Dataset-JSON v1.1 is supported by CDISC as a transport standard (separate from CORE's rule-testing JSON); teams typically convert Dataset-JSON to the shape CORE expects or to XPT as part of the pipeline. This flexibility lets you embed CORE in Python, CI/CD containers, or even drive it from SAS® code [8].

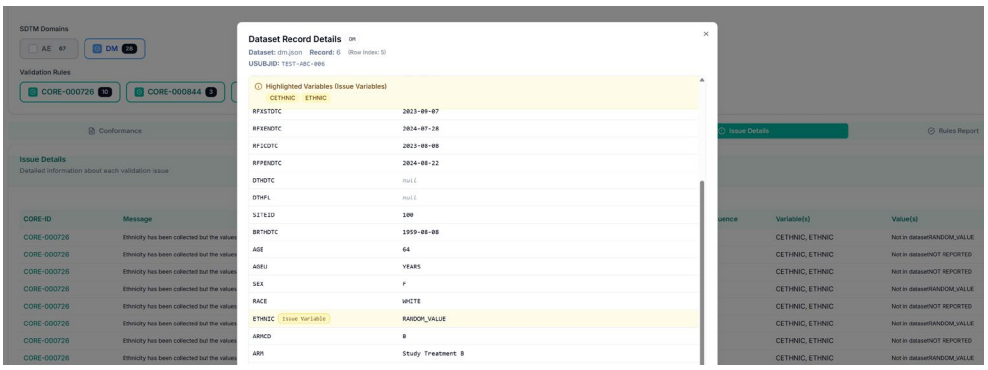
In our Synthetic Data Tool, as shown in **Figure 9**, there is a '**Validate with CDISC CORE**' button highlighted that will directly submit the NDJSON version of the data directly to our CDISC CORE. By default we get an excel report that is part of the CDISC CORE package, this can be generated using the Export to Excel button below. Th enhance the user experience, the same results are returned to the web interface but given some dynamic features to make the experience more interactive. The top cards let us know what domains have been validated and if there are any issues, the bottom section focuses on the issues with various browsing and filtering options to navigate the report.

Figure 10: CDISC CORE web interface



Generally, when working through compliance reports, users will want to work through each dataset and then through each issue within that domain. One of the limitations just now working with validation reports and excel output, there is a lot of manual work required to link the returned issue to the records within a dataset. Filtering excel files and sub setting datasets takes a lot of effort when there are many issues returned. In a web interface it is relatively easy to link issues directly to the NDJSON records:

Figure 11: Issue Details linked to JSON data with highlighting

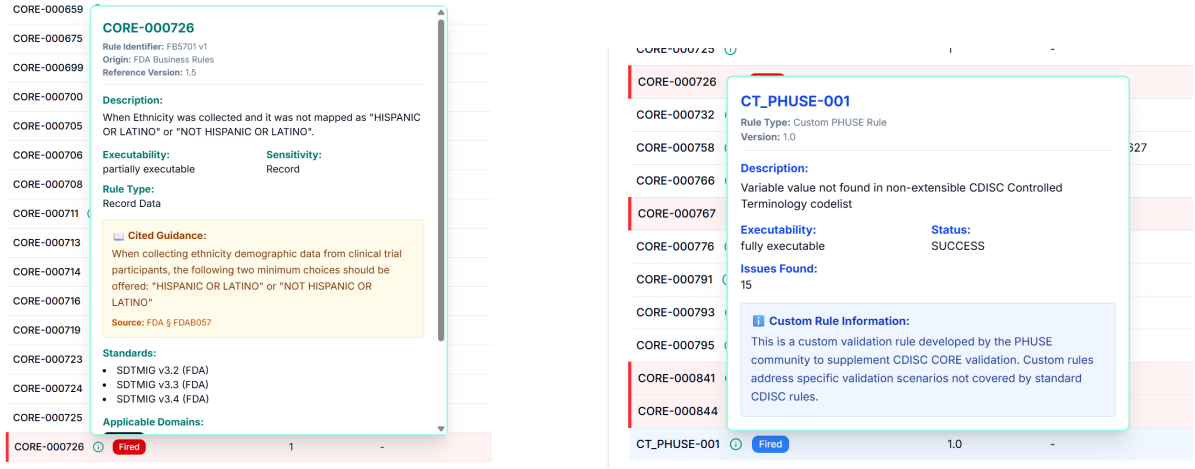


Another great feature of CDISC CORE is we have a list of all rules executed, we can dynamically link to those and have a closer look at which rules have fired and if any custom rules. From the cached rules we can also return any

information saved alongside the rule to provide broader context and any cited guidance to understand the rule origin better:

Figure 12: Rules Report tab, CORE rules fired in pink, CUSTOM rules in blue

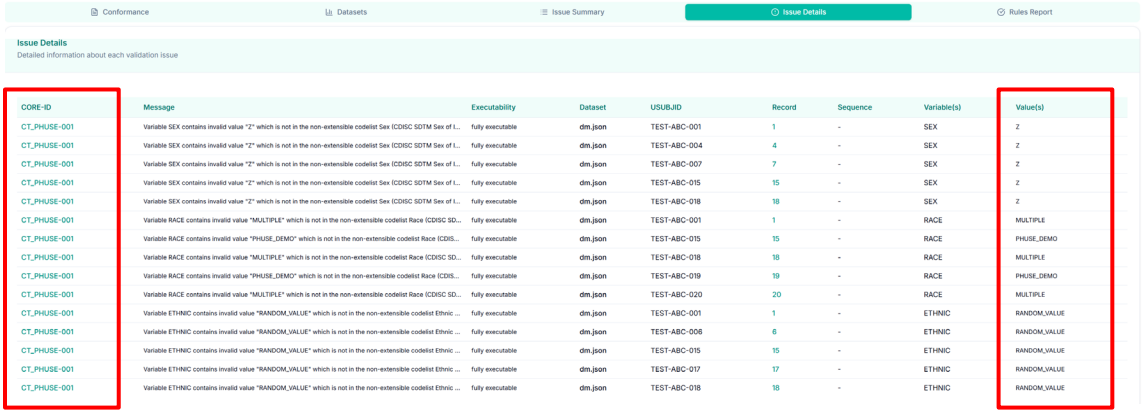
Figure 13: Full background of rules returned to web interface



CUSTOM RULES

CDISC CORE supports the creation and execution of custom conformance rules, giving users flexibility to extend or tailor validation beyond the official CDISC rule sets. This is done by creating a custom rule in a yaml file and storing locally, there is a parameter so the engine executes this rule which also must be added to the rules cache.

In the example above, the CT_PHUSE-001 custom rule has been added as an exploratory example when deploying the rules engine. In this case the rule is covering codelist values found outside the list of a non-extensible codelist. So this is directly catching some issues not already covered in the existing CORE rules, these are values we injected above when creating synthetic data and modifying codelists for SEX and RACE.



The custom rule in this case was added in python code given the complexity of the rule – this was not an attempt to change or persuade how custom rules should be added but merely an exploration of how flexible the CORE engine is and pushing the boundaries of a hybrid approach when uncertain of the implementation via yaml.

One of the nice things about working in a web interface compared to excel files is we can persist the status of fields for sharing across different tools i.e. stop working in silos. As an example below, we can save explanations against issues and this information will persist against future validation runs and more importantly, can be passed directly to Reviewer's Guide tool, which in turn picks up information from other elements and suddenly we can build the full picture required for full SDTM CRT package.

Figure 14: Saving Issue Summary Explanations

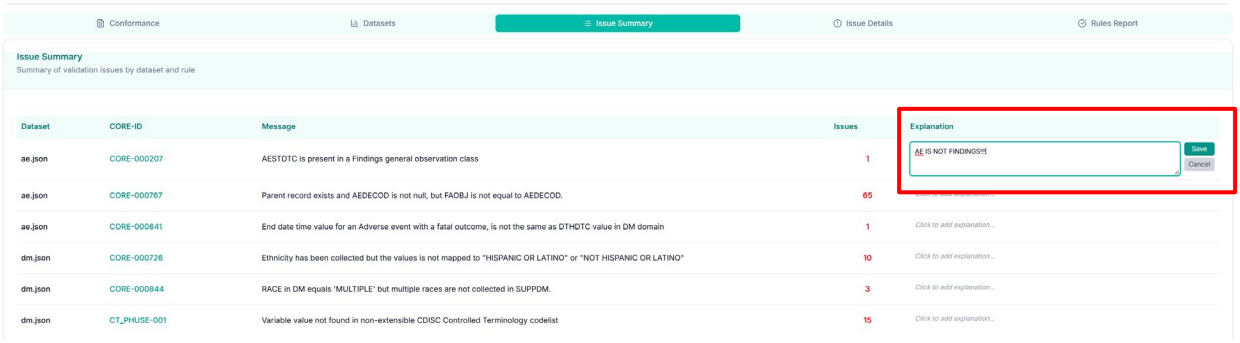
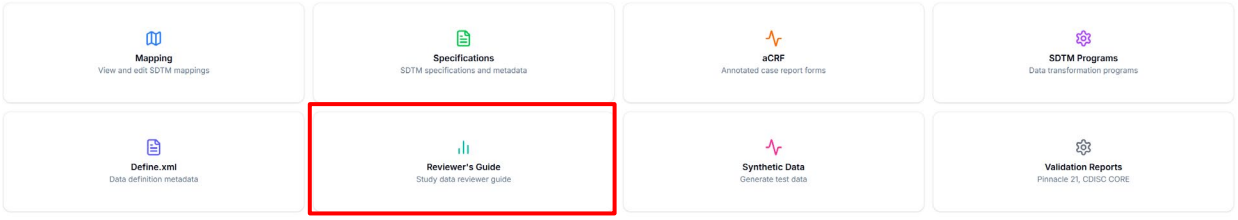


Figure 15: Reviewer's Guide module linked to CORE validation reports



CONCLUSION

The synthetic data generation tool described here enables the creation of highly customizable, study-specific datasets driven directly by define.xml metadata. This allows the generation of realistic, standards-based data tailored to specific study designs, making it highly valuable for downstream processes such as training / tool development but also downstream ADaM / TFL work when live patient data is slow to accumulate.

Built with full CDISC CORE compliance, the tool not only supports validation rule testing but allows users to target and test individual CORE rules, ensuring precision and transparency in standards implementation. By producing datasets in Dataset-JSON and NDJSON formats, it promotes interoperability and directly supports CDISC 360i enablement—linking metadata, data, and validation logic in a single workflow. The integration of real-time CORE validation further ensures that generated data remains compliant throughout the development process.

ACKNOWLEDGEMENTS

I would like to thank Rajwanur Rahman for his dedication and inspiration on all things web-development, giving me all the tools and training to start my own journey and helping me get the project presented here off the ground.

I would also like to acknowledge the contributions of the CDISC community, particularly the members of the Dataset-JSON and CDISC CORE working groups who have shown tremendous dedication to advance the initiatives to their status today. It has been a privilege to implement some of their work and get a closer look at these community driven projects.

REFERENCES

[1] CDISC CORE – https://www.cdisc.org/core

[2] Chris Decker & Peter Van Reusel, CDISC 360i: Implementing the CDISC Roadmap Connecting Standards from Design to Analysis, Proceedings of PHUSE US Connect 2025. https://www.lexjansen.com/phuse-us/2025/ds/PAP_DS01.pdf

[3] Dataset JSON v1.1 - https://www.cdisc.org/standards/data-exchange/dataset-json/dataset-json-v1-1

[4] Python Software Foundation, 2025. xml.etree.ElementTree — The ElementTree XML API. [online] Available at: https://docs.python.org/3/library/xml.etree.elementtree.html

[5] Define-xml - https://www.cdisc.org/standards/data-exchange/define-xml

[6] JSON and NDJSON - https://wiki.cdisc.org/display/PUB/Representing+Dataset-JSON+as+NDJSON

[7] CDISC CORE Github, deployment details - https://github.com/cdisc-org/cdisc-rules-engine

[8] Lex Jansen, Running the CDISC Open Rules Engine (CORE) in BASE SAS, Proceedings of PharmaSUG 2025. https://pharmasug.org/proceedings/2025/SD/PharmaSUG-2025-SD-044.pdf

CONTACT INFORMATION

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

John McDade
PHASTAR
2D Bollo Ln,
Chiswick,
London, W4 5LE,
United Kingdom

Email: john.mcdade@phastar.com
Web: phastar.com