Apache Arrow as a Data Representation Across Pharmaceutical Data Workflows

Ken Nakamatsu

Paper Number: AT-043

Contents

01 Introduction

02 Practical Examples with Apache Arrow

03 Guidance for Users

04 What to take away

01

Introduction

Why Apache Arrow for Data Architecture?

Why focus on architecture now?

AI agents are changing coding and writing in clinical development.

- Coding and writing tasks can be completed much faster
- Developing specialized libraries and tools is becoming more practical
- As implementation becomes easier, design and architecture create more of the difference

Several of my projects converged on the same foundation: **Apache Arrow**

Getting ahead of this shift - and encouraging more development from Japan.

What Apache Arrow is



https://arrow.apache.org

"language-independent standard for representing tabular data in memory.

More broadly, Arrow also refers to:

- **Libraries** implementing the standard in Python, R, Rust, JavaScript, and other languages
- **An ecosystem** of formats and tools built around Apache Arrow, such as **Apache Parquet**, **Polars**, **DuckDB**, and **Apache Spark**

In this presentation, "Arrow" refers to this broader ecosystem.

Arrow is often introduced for its performance:

- Large tables can be processed efficiently
- Tools can exchange data without repeatedly converting or copying it

Why Arrow fits pharmaceutical workflows

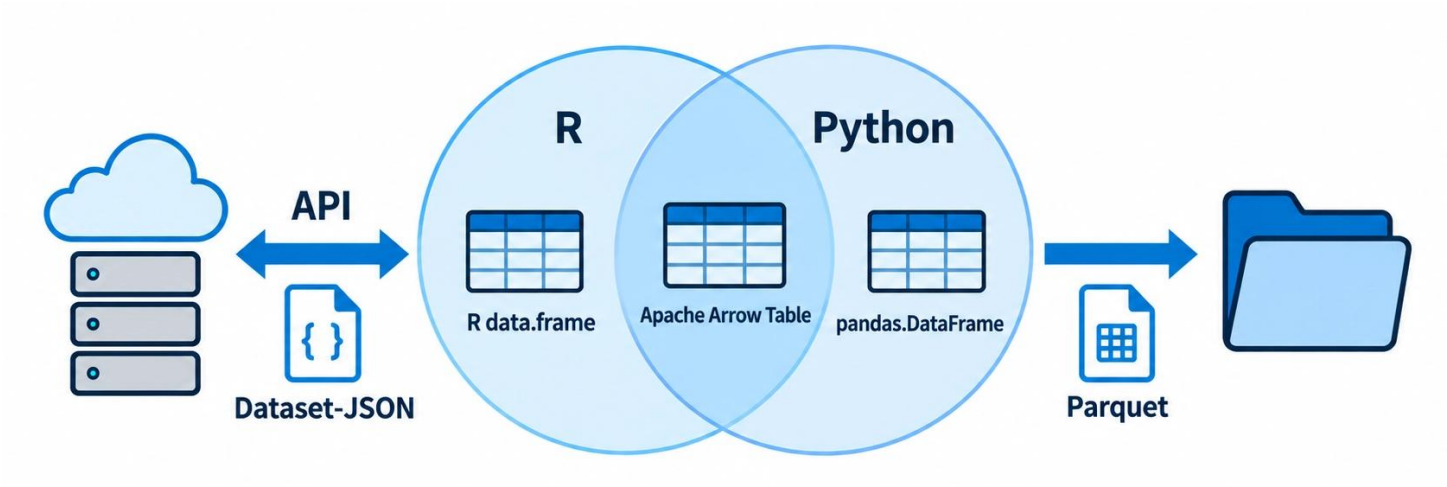
```
>>>type(table)
<class 'pyarrow.lib.Table' >

>>>table.schema
ADTM: timestamp[us]
  -- field metadata --
  itemOID: 'IT.EXAMPLE.ADTM'
  label: 'Analysis Datetime'
  dataType: 'datetime'
AVAL: decimal128(38, 18)
  -- field metadata --
  itemOID: 'IT.EXAMPLE.AVAL'
  label: 'Analysis Value'
  length: '7'
ASEQ: int64
  -- field metadata --
  itemOID: 'IT.EXAMPLE.ASEQ'
  label: 'Analysis Sequence'
  dataType: 'integer'
-- schema metadata --
datasetJSONCreationDateTime: '2026-03-25T00:00:00Z'
datasetJSONVersion: '1.1'
itemGroupOID: 'IG.EXAMPLE'
name: 'EXAMPLE'
label: 'EXAMPLE Dataset'
```

- Rich data types represent values with explicit types
- Nullability and missing values are handled explicitly
- Schema checks can detect unintended changes early
- Field and dataset metadata remain attached to the data
- The same representation works across languages and tools
- Dataset-JSON requirements fit naturally into the schema

This reduces custom code for data models, type checking, validation, and metadata handling.

The right representation for each boundary



Dataset-JSON : Standards-based exchange with external parties

Apache Arrow : In-memory data inside tools and between components

Apache Parquet : Efficient storage within a person or organization

Apache Arrow is NOT a replacement for Dataset-JSON,

but another representation for **in-memory tabular data** like pandas, R data.frame.

02

Practical Examples with Apache Arrow

Libraries and tools built around Apache Arrow

1. DATA step-style data handling

The problem

- SAS language is available only in licensed environments
- AI-assisted conversion can greatly reduce rewriting time

  Converted code still requires careful review

- Existing DATA step skills and code remain valuable

The implementation

Build a DATA step–style execution engine, instead of converting code.

Language: Python, Rust

Repository: https://github.com/k-nkmt/limulus

License: PolyForm Noncommercial License 1.0.0

→ Broader use permissions are under consideration.

- Apache Arrow is the shared internal table representation
- Apache Arrow connects the Python and Rust layers
- Reuses familiar DATA step skills and programming patterns
- Designed as a modern data-handling engine, not as a SAS® software compatible product

Demo: limulus

Documentation: https://k-nkmt.github.io/limulus/

```
import limulus
import pandas as pd
import polars as pl

iris_data = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv')

session = limulus.Session()
session.loads({"iris": iris_data})
```

```
session.submit("""
data ratios;
  set iris;
  sepal_ratio = round(sepal_length / sepal_width, 0.01);
  petal_area = round(petal_length * petal_width, 0.01);
  keep species sepal_length sepal_width sepal_ratio petal_area;
run;
""")

print(pl.from_arrow(session["ratios"]).head(5))
```

shape: (5, 5)

sepal_length	sepal_width	species	sepal_ratio	petal_area
f64	f64	str	f64	f64
5.1	3.5	setosa	1.46	0.28
4.9	3.0	setosa	1.63	0.28
4.7	3.2	setosa	1.47	0.26
4.6	3.1	setosa	1.48	0.3
5.0	3.6	setosa	1.39	0.28

```
session.submit("""
data large_petal small_petal;
  set iris;
  if petal_length > 2.0 then do ;
    size_group = "large" ;
    output large_petal;
  end ;
  else do ;
    size_group = "small" ;
    output small_petal;
  end ;
run;
""")

# 2 ways convert to polars
print(pl.from_arrow(session["large_petal"]).head(3))
print(session.dataset("small_petal").to_polars().head(3))
```

shape: (3, 6)

sepal_length	sepal_width	petal_length	petal_width	species	size_gr
f64	f64	f64	f64	str	str
7.0	3.2	4.7	1.4	versicolor	large
6.4	3.2	4.5	1.5	versicolor	large
6.9	3.1	4.9	1.5	versicolor	large

Representative features:

- Do loop
- Multiple output datasets
- BY group processing
- Joining Data
- Set Label
- Put, input, and type conversion

```
session.submit("""
data avg_petal;
  set ratios;
  by species;
  retain area_sum 0 area_count 0;

  if first.species then do;
    area_sum = 0;
    area_count = 0;
  end;

  area_sum + petal_area;
  area_count + 1;

  if last.species then do;
    avg_petal_area = round(area_sum / area_count, 0.01);
    output;
  end;

  keep species area_count avg_petal_area;
run;
""")

print(session.dataset("avg_petal").to_polars())
```

shape: (3, 3)

species	area_count	avg_petal_area
str	f64	f64
setosa	50.0	0.37
versicolor	50.0	5.72
virginica	50.0	11.3

2. Dataset-JSON and dataframe conversion

The problem

- Existing Python libraries mainly focus on file-to-file conversion
- Python dataframe support is limited to pandas

- The convenience of a text-based format comes with trade-offs

  Dataset-JSON needs **safeguards** for data structure and types; JSON Schema alone is not sufficient.

The implementation

Use Apache Arrow and additional checks to preserve the intended structure and types in Dataset-JSON.

Language: Python

Repository: https://github.com/k-nkmt/dsjframe

License: AGPL-3.0

- Converts directly between Apache Arrow and Dataset-JSON
- Apache Arrow types cover Dataset-JSON data types without fallback conversions
- Adds checks for data structure and types
- Addresses unresolved issues in the Dataset-JSON specification
- Connects to **saspy** and **pyreadstat** for SAS7BDAT

Demo: dsjframe

Example: https://github.com/k-nkmt/dsjframe/blob/main/example.ipynb

```python
import dsjframe
# Read
table = dsjframe.read_dataset("adsl.json")

# Write
metadata = {
    "datasetJSONVersion": "1.1.0",
    "label": "Subject Level Analysis Dataset",
}

dsjframe.write_dataset(table, "adsl.ndjson", metadata)

# validate
report = dsjframe.validate_dataset("adsl.dsjc")
```

```python
with open('tests/data/input/adsl.json', 'r', encoding='utf-8') as f:
    sample = json.load(f)

test_meta = sample.copy()
del test_meta["datasetJSONVersion"] # missing required field

test_rows = sample.copy()
test_rows["rows"] = [
    ['SUBJ-001', 35.1, 'https://example.com/1'], # invalid type (not int)
    ['SUBJ-002', None, 'https://example.com/2']
]

test_meta_byte = json.dumps(test_meta, ensure_ascii=False).encode('utf-8')
print(dsjframe.validate_dataset(test_meta_byte))

test_rows_byte = json.dumps(test_rows, ensure_ascii=False).encode('utf-8')
print(dsjframe.validate_dataset(test_rows_byte))
```

```
>>>
{'ok': False, 'format': 'json', 'metadata_valid': False, 'data_valid': None, 'issue_code':
'schema_error', 'message': 'missing required field: datasetJSONVersion'}

>>>
 {'ok': False, 'format': 'json', 'metadata_valid': True, 'data_valid': False, 'issue_code':
'value_conversion_error', 'message': 'failed to convert value for column AGE'}
```

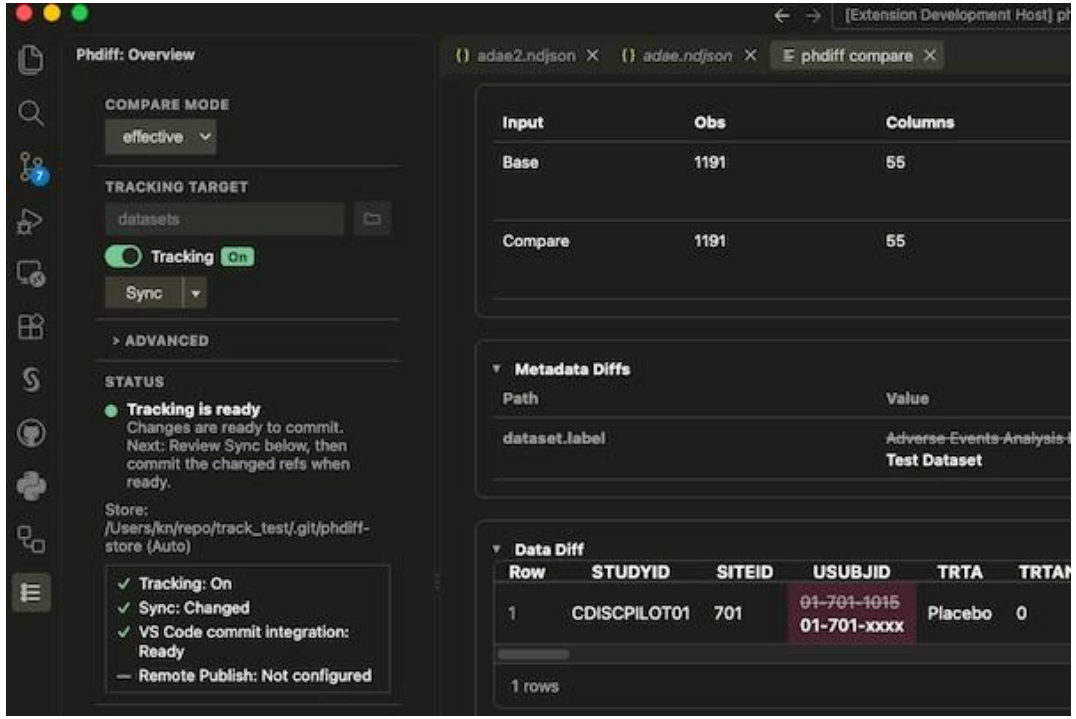
- Supports the DSJC format
- Processes NDJSON and DSJC as streams for efficient handling
- Can preserve empty strings instead of converting them to null via an option

3. Work in progress

Dataset viewer and version-control support

A VSCode extension for viewing, comparing, and managing submission datasets with Git.

- Dataset viewer and structural comparison
- Git-based version-control support: more flexible than Git LFS, simpler than DVC
- Rust for data processing, JavaScript for the VSCode interface
- Apache Arrow as the internal data representation



What's next?

03

Guidance for Users

Where Arrow Fits in Existing Workflows

Python: modern data workflows with Arrow

Apache Arrow was shaped by **limitations** encountered in pandas.

- Pandas itself has increasingly adopted Apache Arrow.

New workflows: Start with the Arrow ecosystem, such as PyArrow, Polars, and DuckDB, and use pandas only for the parts where it is useful.

Existing pandas workflows: PyArrow-backed I/O and data types can be used while keeping familiar pandas syntax.

```python
import pandas as pd


df_arrow = pd.read_csv(
    "iris.csv", engine="pyarrow", dtype_backend="pyarrow"
)

# Convert
df = pd.read_csv("iris.csv")
df_arrow = df.convert_dtypes(dtype_backend="pyarrow")
```

R: moving beyond base R data handling

Base R data handling has some of the same problems as pandas.

- The tidyverse (tibble, dplyr) already addresses most of these issues
- Prefer tidyverse-based workflows for general data processing
- Arrow for R supports many dplyr operations using the same syntax.
- Arrow can be a better fit for specific use cases

Like tibble, Arrow for R allows whole-column type changes. PyArrow Tables do not support direct assignment.

```r
library(tibble)
library(arrow)


tbl <- tibble(id = 1:3, value = c(10, 20, 30))
df  <- data.frame(id = 1:3, value = c(10, 20, 30))
atb <- arrow_table(id = 1:3, value = c(10, 20, 30))


df[1, "value"]  <- "AA"   # Allowed
tbl[1, "value"] <- "AA"   # Type error
atb[1, "value"]  <- "AA"  # Not Allowed

df$value  <- "AA"
tbl$value <- "AA" # Whole-column replacement is allowed

atb$value <- "AA" # Allowed in Arrow for R
```

R and Python interoperability: CSV vs. Arrow

From R, reticulate can be used to call Python and exchange data with it.

```
library(arrow)
library(reticulate)
use_virtualenv(".venv")
py_run_string("import pandas as pd")


rows <- 500000L
arrow_data <- arrow_table( id = 1:rows, value = (1:rows) * 0.5)
data_frame <- data.frame( id = 1:rows, value = (1:rows) * 0.5)
```

Observed: for 500,000 rows.

- CSV about 250 ms (50ms with Arrow CSV writer)
- Arrow under 1 ms

Results vary depending on data size, data types, and environment.

CSV path

R writes a temporary file. Python reads the file and rebuilds the dataframe.

```
csv_path <- tempfile( fileext = ".csv")
write.csv(data_frame, csv_path, row.names = FALSE)
py$csv_path <- csv_path
py_run_string("df_csv = pd.read_csv(csv_path); print(len(df_csv))")
unlink(csv_path)
```

Arrow path

R passes a reference to the Arrow data in memory, so Python can use the same data without file I/O or rebuilding it.

```
py$arrow_table <- r_to_py(arrow_data)
py_run_string("print(len(arrow_table))")
```

04

What to take away

For developers

- Apache Arrow is a strong architectural foundation for multi-language tools
- Its standard types and metadata can cover Dataset-JSON requirements
- Consider it early, before defining custom table and conversion layers

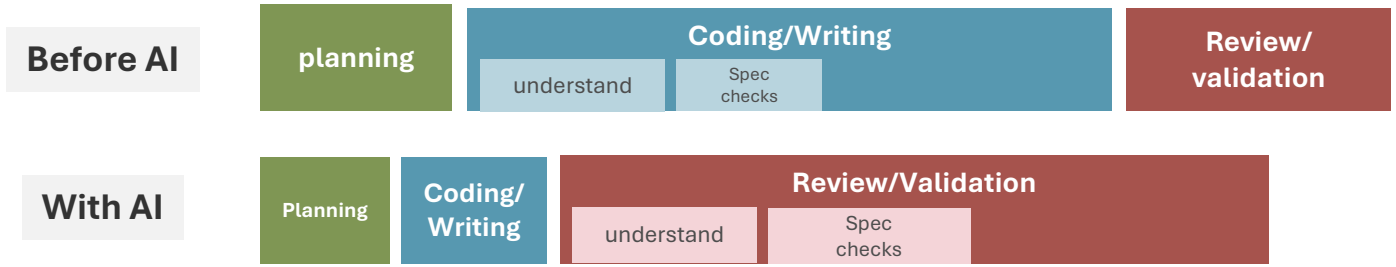
For users

- **Python**: start with Arrow for cross-tool workflows and convert to pandas when its ecosystem is needed
- **R**: prefer tidyverse conventions over base data.frame for predictable handling

  Consider Arrow for specific workflows where it offers a practical advantage.

Validate Dataset-JSON

- Validation in most Dataset-JSON libraries is limited to schema checks.(Oct 2026)
- These checks do not guarantee consistency between the actual data and its metadata.



The bottleneck is shifting to **review and validation**.

Possible approaches:

- AI is not needed for deterministic tasks.
- Build quality into design and processes.
- Share tools and libraries for continuous improvement.

Thank you.

E-mail: ken-nakamatsu@knworx.com
Github: https://github.com/k-nkmt