Paper PD05

Leadership Strategies and Practical Frameworks for Statistical Programming in a Multilingual Era

Shaveta Bansal, Merck & Co., Inc., Rahway, NJ, USA

INTRODUCTION

Statistical programming in clinical trials has evolved into a strategic discipline impacting trial design, data integrity, compliance, and patient outcomes

Combines languages like SAS (regulatory/legacy), R (statistics/visualization), Python (automation/ML), and SQL (scalable data)

Guidance focuses on four key areas:

Code Standardization and Governance

Documentation and Communication (programming language interoperability)

Data Integrity and Regulatory Compliance

Collaboration and Knowledge Sharing

CODE STANDARDIZATION AND GOVERNANCE

Objective

Build consistent, high-quality, and auditable code across multiple programming languages used in clinical trials.

Why It Matters

Inconsistent Practices

Rework & Submission Risk

Mismatched Outputs & Errors

Regulatory Traceability

Reviewable Pipelines

Deterministic & Reproducible

Gen AI & Automation

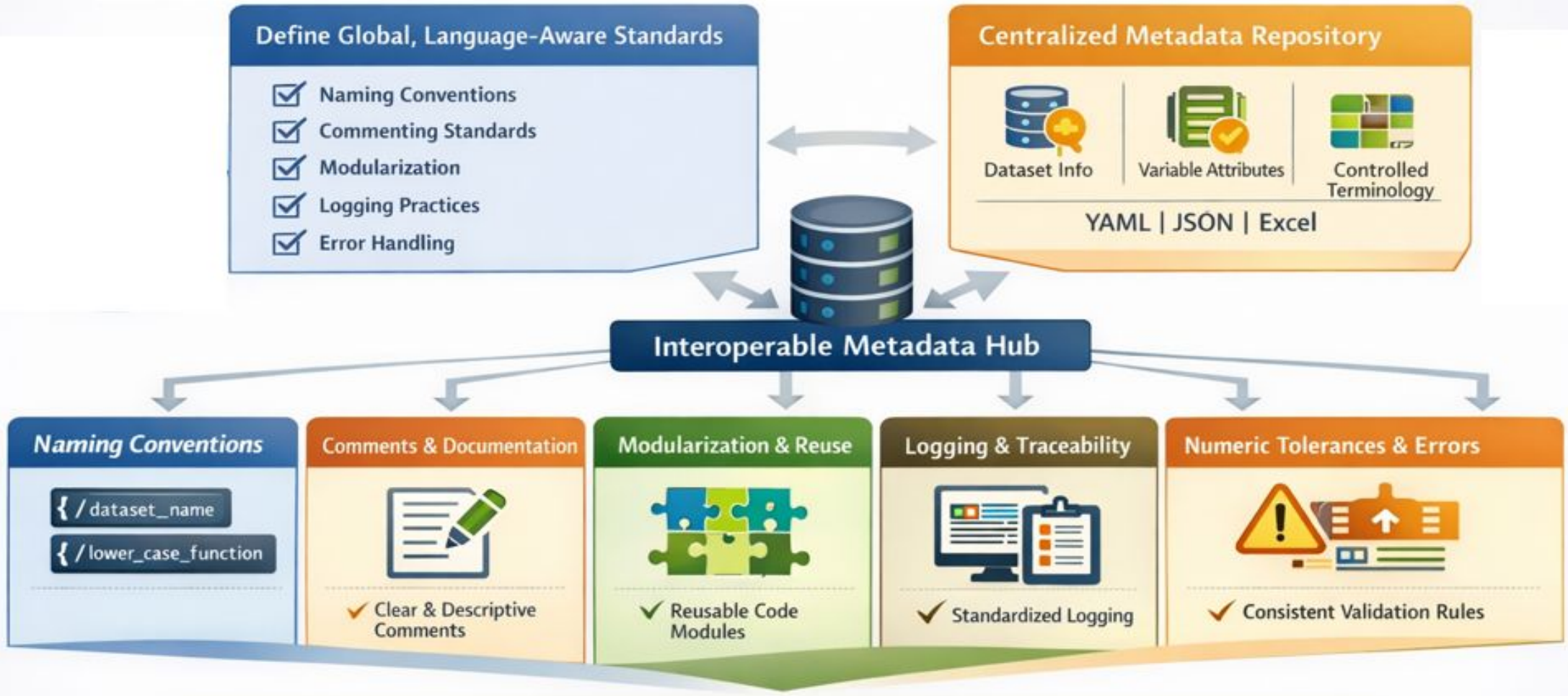
Speed & Risk Management

Needs Oversight & Controls

Outcomes

- ✓ Consistent Process
- ✓ Reduced Review Effort
- ✓ Scalable, Compliant Analytics

CODE STANDARDIZATION AND GOVERNANCE



Governance and Review at Scale

Purpose: Embed Quality, Consistency and Compliance into SAS, R and Python Workflows
Prevent Issues Early, Not Fix later

Multilingual Governance & Review

- Cross-language Code Review Board
- Common Coding Standards
- Review Critical Code

Architecture Decision Records (ADRs)

- Document key technical choices
- Explain Decisions and alternatives
- Sotre in Central ADR repository

Gen AI Governance

- Approved AI tools
- Validate AI code
- Human Review Required

Automated Enforcement

- Quality checks into CI/CD Pipelines
- Automated Checks
- Catch Issues Early

Reusable Metadata

- Reusable templates
- Metadata and CI models
- Expert Champions
- Phased Rollout

Documentation & Communication

Unified, Language-Neutral Specs

Define	Define programming logic once in language-neutral formats (e.g., YAML, JSON)
Establish	Establish a single source of truth for derivations and specifications
Use	Use modern tools and automation to keep documentation, code, and outputs synchronized as changes occur.
Move away	Move away from siloed, static processes by modernizing incrementally, starting with high-value use cases.
Enable	Enable adoption through leadership support, training, and protected time.

Data Integrity & Regulatory Compliance

Ensure Data quality and Reduce Submission risk

Unified Metadata & Version Control

- Standardized Data Rules
- Shared Metadata Repository
- Version Controlled Schemas

Access Controls & Security

- User Permissions
- Electronic Signatures
- Audit Logs

Compliance & Audit Readiness

Automated Validate Checks

- CDISC & Validation Checks
- Blocking non-compliant code early
- Audit trails and run logs

Gold Standard Testing

- Double/Independent Programming
- Comparative results

Collaboration and Knowledge Sharing

Open & Connected Culture – Share Best Practices, Engage external experts

Job Rotations & Micro-Apprenticeships

- Cross- Training with other language teams
- Skill Building Projects

Knowledge Sharing Incentives

- Recognition & Awards
- Contribution Credits
- Mentorship Across Languages

Shared Learning

Cross Language Collaboration

Innovative Sandbox

- Safe Experimentation Space
- New Tools & Methods

Meetups and External Connections

- Hackathons & Symposia
- Internal Knowledge Sharing
- Cross-pharma Collaboration (Phuse workshops)

Challenges and Solutions

Protect Capacity, Clarify Ownership, Invest in skills and Reinforce Shared Standards

Challenges in Multilingual Programming Implementation

Competing Priorities & Time Pressure
- Tight delivery timelines, limited capacity for new standards.

Fragmented Ownership
- Split responsibilities across Programming, Biostatistics, Data Management, IT.

Legacy Systems & Processes
- Old templates, Word specs, single language workflows.

Skill Gaps Across Languages
- Lack of experience with Git, YAML, and automation.

Difficulty Sustaining Momentum
- Change depends on many contributors, no single owner.

How Leaders Can Help Address These Challenges

Protect Capacity for Modernization
- Dedicate time & recognize long-term improvements.

Establish Cross-Functional Governance
- Steering groups & shared accountability.

Start Small & Scale Gradually
- Modernize high-value workflows first.

Invest in Capability Building
- Structured training & micro-apprenticeships.

Reinforce & Reward Collaboration
- Incentives for reusable assets & language champions.

Thank You!

Driving Consistent & Collaborative Programming Across SAS, R, & Python

sas | R | Python

Standard Error Detection and Management Across SAS, R, and Python

Derive SDTM measurements (height_cm, weight_kg) with derived BMI
Rule: `bmi = weight_kg / (height_cm / 100)^2`
Key: `USUBJID`

SAS

```
/* Check for dataset existence */
%macro assert_exist(ds);
%if not %sysfunc(exist(&ds)) %then
%do;
%put ERROR: Dataset &ds does not
exist.;
%abort cancel;
%end;
%mend assert_exist;

%assert_exist(adsl);

/* Validate inputs and derive BMI */
data adsl;
set adsl;
if missing(height_cm) or
missing(weight_kg) or height_cm <= 0
then do;
bmi = .;
put "WARN: Invalid height/weight for "
usubjid=;
end;
else bmi = weight_kg /
((height_cm/100)**2);
run;
```

R

```
# Validate columns
validate_columns <- function(df) {
required <- c("usubjid", "height_cm",
"weight_kg")
missing <- setdiff(required, names(df))
if (length(missing) > 0) stop(paste("Missing
columns:", paste(missing, collapse=", ")))
}

# Safe BMI derivation
safe_derive_bmi <- function(weight_kg,
height_cm) {
if (is.na(height_cm) || is.na(weight_kg) ||
height_cm <= 0) {
warning("Invalid height/weight; returning
NA.")
return(NA_real_)
}
weight_kg / (height_cm / 100)^2
}

validate_columns(adsl)
adsl$bmi <- mapply(safe_derive_bmi,
adsl$weight_kg, adsl$height_cm)
```

Python

```
import logging
import pandas as pd

def validate_columns(df):
required = {"usubjid", "height_cm", "weight_kg"}
missing = required - set(df.columns)
if missing:
raise ValueError(f"Required columns missing:
{missing}")

def safe_derive_bmi(weight_kg, height_cm):
try:
if pd.isna(height_cm) or pd.isna(weight_kg) or
height_cm <= 0:
logging.warning("Invalid height/weight;
returning NaN.")
return float("nan")
return weight_kg / (height_cm / 100) ** 2
except Exception as exc:
logging.error("BMI derivation failed: %s", exc)
return float("nan")

validate_columns(adsl)
adsl["bmi"] = adsl.apply(lambda r:
safe_derive_bmi(r["weight_kg"], r["height_cm"]),
axis=1)
```