Clinical Trial Data Validation Across Languages and Tools

From EDC Edit Checks to a Spreadsheet-Driven Lua Pipeline

Ronald 'Ryy' G. Thomas

2025-01-15

Table of contents

Figure 1: Clinical trial data validation spans multiple languages and tools, from spreadsheet rule definitions to real-time browser checks.

1 Introduction

I did not really understand how much complexity hides behind the phrase "data validation" until I tried to map out the full landscape for a clinical trial database. What began as a question about edit checks in an EDC system quickly expanded into comparisons across proprietary platforms, open-source alternatives, general-purpose languages, and a surprisingly elegant pipeline that starts with a spreadsheet and ends with real-time browser validation powered by Lua-generated JavaScript.

Clinical trial data validation is not a single technique. It is a layered system of temporal checks, cross-field logic, protocol adherence rules, dynamic thresholds, plausibility constraints, missing data pattern recognition, dictionary lookups, conditional queries, visit sequencing, and duplicate detection. Each of these layers can be implemented in different tools, and the choice of tool shapes the trade-offs between accessibility, flexibility, and performance.

This post walks through what I learned about each layer, compares the available tooling, and builds toward a novel pipeline where non-programmers define validation rules in a CSV file, Lua processes those rules into JavaScript functions, and a Shiny application loads the generated JavaScript for real-time form validation. The Lua pipeline is the part I find most interesting, and it appears to be genuinely novel in the clinical data validation context.

1.1 Motivations

- I was maintaining validation logic scattered across multiple scripts with no single source of truth for which checks applied to which fields.
- Non-programmer collaborators (clinicians, data managers) needed to review and modify validation rules, but they could not read R or Python code.
- I wanted to understand the full taxonomy of EDC validation checks so I could map them against what open-source tools actually support.
- The idea of generating validation code from a spreadsheet felt powerful, and I wanted to see whether Lua was a practical choice for that code generation step.
- I needed a concrete reference for how Shiny handles form validation, from basic `validate()`/`need()` to the `shinyvalidate` package to client-side JavaScript.

1.2 Objectives

1. Catalogue the ten major categories of EDC validation checks used in clinical trials, with concrete examples for each.
2. Compare proprietary and open-source EDC platforms (Medidata Rave, OpenClinica, REDCap, ClinCapture) in terms of validation capabilities.
3. Demonstrate validation implementations in R (`validate`), Python (`pandera`), and SQL.
4. Build a spreadsheet-to-Lua-to-JavaScript-to-Shiny pipeline for dynamic, non-programmer-accessible validation rule management.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know.

Figure 2: Clinical precision begins with validated data.

2 Prerequisites and Setup

This post is primarily conceptual and comparative, with code examples in R, Python, SQL, Lua, JavaScript, and HTML. No single runtime is required to follow along, but the following tools appear in the code blocks:

```
library(validate)
library(shiny)
library(shinyvalidate)
library(shinyjs)
```

For the Lua pipeline sections, a Lua interpreter with a CSV parsing library (such as `luacsv`) is assumed. For the JavaScript database examples, a modern browser with IndexedDB support is sufficient.

Background. Familiarity with clinical trial terminology (CRF, EDC, MedDRA, CDISC) is helpful but not strictly required. Each term is explained in context.

3 What is Clinical Trial Data Validation?

Data validation in a clinical trial is the systematic process of verifying that collected data is accurate, complete, and internally consistent before it enters any analysis. Think of it as a series of gatekeepers: each gatekeeper checks one aspect of the data (is the date plausible? does the dosage match the protocol? has this patient been entered twice?) and either passes the record through or flags it for human review.

The process has three main components. First, **edit checks** are automated rules applied during data entry that catch discrepancies such as out-of-range values or logical inconsistencies. Second, **source data verification** (SDV) cross-checks entered data against original source documents. Third, **batch validation** reviews accumulated data periodically to detect trends or patterns that suggest systematic errors. Together, these components are governed by regulatory frameworks from the FDA and EMA that mandate data integrity throughout the trial lifecycle.

4 Getting Started: The Ten Categories of

EDC Validation Checks

Electronic Data Capture systems support a taxonomy of validation checks that goes well beyond simple range constraints. The following ten categories represent the full spectrum of what modern EDC systems can enforce.

1. Temporal consistency checks ensure that date and time fields follow logical sequences. A visit date cannot precede the screening date. An adverse event resolution date must follow the event start date.

2. Cross-field logical consistency checks validate relationships between multiple data fields. If a participant is marked as pregnant, gender must be female. If a patient received a vaccine, the vaccination status field cannot read "unvaccinated."

3. Protocol adherence checks verify that data aligns with the study protocol. Weight-based drug dosages must fall within protocol-defined limits. Inclusion and exclusion criteria must be satisfied (for example, age between 18 and 65 for eligibility).

4. Range checks with dynamic thresholds go beyond static limits by adjusting thresholds based on patient demographics or prior values. A hemoglobin level drop greater than 2 g/dL from baseline triggers a flag, rather than relying on a fixed normal range.

5. Plausibility checks using historical data detect outliers based on patient history or known disease patterns. A blood pressure reading that drops from 140/90 to 80/40 should trigger a flag unless a serious adverse event is also recorded.

6. Missing data pattern recognition flags missing entries based on expected patterns. If "Yes" is recorded for treatment administration, treatment details must be present.

7. MedDRA and WHO Drug Dictionary validation checks that reported adverse events and medications map to standardised medical dictionaries. An adverse event entered as "Headache" must resolve to the correct MedDRA preferred term.

8. Conditional queries based on risk-based monitoring use statistical rules or machine learning to prioritise data review. Sites with higher rates of protocol deviations receive additional validation scrutiny.

9. Visit and event sequencing checks ensure that study visits and procedures occur in the correct order. Randomisation must precede the first dose of study drug.

10. Duplicate record detection identifies duplicate patients or entries using probabilistic matching on name, date of birth, and enrolment date.

Figure 3: Validation tools range from proprietary EDC platforms to open-source systems and general-purpose programming languages.

4.1 Deeper Analysis: Tools for Implementing

Validation Checks

The ten categories above can be implemented across a range of platforms. The choice depends on organisational context, budget, and the degree of customisation required.

4.1.1 Proprietary EDC Systems

Several commercial platforms provide built-in validation engines:

- **Medidata Rave** uses Medidata Rave Edit Check Scripts for defining validation rules.
- **Oracle Clinical / InForm** relies on PL/SQL-based validation.
- **IBM Clinical Development**, **Veeva Vault EDC**, and **Castor EDC** provide graphical interfaces or scripting languages for rule definition.

These platforms offer convenience and regulatory compliance tooling, but they are expensive and their validation logic is typically locked within the platform.

4.1.2 Open-Source EDC Systems

OpenClinica supports real-time edit checks using XPath expressions. The following example rejects age values above 100:

```
<rule>
    <when>
        /StudyEventData/FormData/ItemGroupData
        /ItemData[@ItemOID='AGE'] > 100
    </when>
    <then>
        <message>
            Age cannot be greater than 100 years.
        </message>
    </then>
</rule>
```

REDCap provides branching logic, calculated fields, and custom data quality rules. A simple eligibility check looks like:

```
[age] > 18 AND [age] < 65
```

ClinCapture offers JavaScript-based validation for logic and range checks, along with custom queries for detecting missing or inconsistent data.

4.1.3 General-Purpose Languages

When EDC platforms lack the flexibility needed for complex validation logic, R, Python, and SQL offer programmable alternatives.

R with the `validate` package. The `validate` package defines rules declaratively and confronts a data frame against them:

```
library(validate)

rules <- validator(
  age >= 18,
  bmi >= 15 & bmi <= 50,
  start_date < end_date
```

```
)
check_results <- confront(data, rules)
summary(check_results)
```

The `pointblank` package provides a complementary approach with pipeline-style validation and HTML reporting.

Python with `pandera`. The `pandera` library defines schema-level constraints for pandas DataFrames:

```
from pandera import DataFrameSchema, Column, Check

schema = DataFrameSchema({
    "age": Column(
        int,
        Check(
            lambda x: 18 <= x <= 65,
            error="Age must be 18-65"
        )
    ),
    "bmi": Column(
        float,
        Check(
            lambda x: 15 <= x <= 50,
            error="BMI must be realistic"
        )
    ),
    "start_date": Column(str),
    "end_date": Column(str),
})

validated_data = schema.validate(df)
```

SQL for integrity checks. Direct queries against the clinical database catch constraint violations at the storage layer:

```
SELECT patient_id, age
FROM clinical_data
WHERE age < 18 OR age > 100;
```

4.1.4 Platform Comparison

Approach	Strengths	Limitations
Proprietary EDC	Regulatory tooling, support	Expensive, locked-in
OpenClinica	XPath rules, open-source	Learning curve
REDCap	Accessible, widely adopted	Limited complex logic

Approach	Strengths	Limitations
R (`validate`)	Flexible, reproducible	Requires R expertise
Python (`pandera`)	Schema validation	Requires Python
SQL	Direct database checks	No UI integration

4.2 Shiny Form Validation: Three Approaches

Shiny can create forms with real-time validation using three progressively more sophisticated methods.

4.2.1 Method 1: `validate()` and `need()`

The built-in `validate()` and `need()` functions provide simple reactive validation with immediate error messages:

```
library(shiny)

ui <- fluidPage(
  titlePanel("Real-Time Validation Example"),
  sidebarLayout(
    sidebarPanel(
      numericInput(
        "age", "Enter Age:",
        value = NULL, min = 0, max = 100
      ),
      verbatimTextOutput("validation_message"),
      actionButton("submit", "Submit")
    ),
    mainPanel()
  )
)

server <- function(input, output, session) {
  output$validation_message <- renderText({
    validate(
      need(
        input$age >= 18,
        "Age must be at least 18"
      ),
      need(
        input$age <= 65,
        "Age must be 65 or below"
      )
    )
    "Valid input!"
```

```
  })
}

shinyApp(ui, server)
```

4.2.2 Method 2: The `shinyvalidate` Package

The `shinyvalidate` package supports dependent multi-field validation before submission:

```
library(shiny)
library(shinyvalidate)

ui <- fluidPage(
  titlePanel("Shinyvalidate Example"),
  textInput("email", "Enter Email:"),
  numericInput(
    "age", "Enter Age:",
    value = NULL, min = 0, max = 100
  ),
  actionButton("submit", "Submit"),
  verbatimTextOutput("validation_message")
)

server <- function(input, output, session) {
  iv <- InputValidator$new()
  iv$add_rule("email", sv_email())
  iv$add_rule("age", sv_between(18, 65))
  iv$enable()

  observeEvent(input$submit, {
    if (iv$is_valid()) {
      showModal(
        modalDialog("Form submitted successfully!")
      )
    } else {
      showModal(
        modalDialog(
          "Please fix errors before submitting."
        )
      )
    }
  })
}

shinyApp(ui, server)
```

4.2.3 Method 3: Client-Side JavaScript

For instant feedback without a server round-trip, JavaScript validation runs directly in the browser:

```
library(shiny)

ui <- fluidPage(
  tags$script(HTML("
    function validateNumericInput() {
      var input =
        document.getElementById('numInput').value;
      if (isNaN(input) ||
          input < 1 || input > 100) {
        document.getElementById('error')
          .innerHTML =
          'Enter a valid number (1-100)';
      } else {
        document.getElementById('error')
          .innerHTML = '';
      }
    }
  ")),
  textInput(
    "numInput", "Enter a number:", "",
    oninput = "validateNumericInput()"
  ),
  span(id = "error", style = "color: red;")
)

server <- function(input, output, session) {}

shinyApp(ui, server)
```

4.3 JavaScript and Relational Data

JavaScript can access relational datasets for real-time validation through several mechanisms.

4.3.1 IndexedDB for Local Validation

IndexedDB provides a client-side relational database in the browser, suitable for validating against small reference datasets without server calls:

```
<script>
  let db;
```

```
  let request = indexedDB.open("ClinicalDB", 1);
  request.onsuccess = function(event) {
    db = event.target.result;
  };

  function validateUserID() {
    let inputID =
      document.getElementById("userID").value;
    let transaction =
      db.transaction(["patients"]);
    let objectStore =
      transaction.objectStore("patients");
    let request = objectStore.get(inputID);

    request.onsuccess = function() {
      if (!request.result) {
        document.getElementById("error")
          .innerHTML = "Invalid Patient ID!";
      } else {
        document.getElementById("error")
          .innerHTML = "";
      }
    };
  }
</script>
<input id="userID" type="text"
  oninput="validateUserID()">
<span id="error" style="color: red;"></span>
```

4.3.2 AJAX for Remote Database Queries

For larger datasets or centralised validation, AJAX requests query a server-side SQL database:

```
<script>
  function checkPatientID() {
    let userID =
      document.getElementById("userID").value;
    fetch(`/validate_id?userID=${userID}`)
      .then(response => response.json())
      .then(data => {
        if (data.valid) {
          document.getElementById("error")
            .innerHTML = "";
        } else {
          document.getElementById("error")
```

```
            .innerHTML = "Invalid Patient ID!";
        }
      });
  }
</script>
<input id="userID" type="text"
  oninput="checkPatientID()">
<span id="error" style="color: red;"></span>
```

4.3.3 Machine Learning API Calls

For validation that requires pattern recognition (such as classifying adverse event descriptions), JavaScript can call a server-side ML model:

```
<script>
  function validateAdverseEvent() {
    let eventText =
      document.getElementById("eventText").value;
    fetch(`/predict_adverse_event`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json"
      },
      body: JSON.stringify({ "text": eventText })
    })
    .then(response => response.json())
    .then(data => {
      document.getElementById("error").innerHTML =
        data.valid ? "" :
        "Potentially invalid event description.";
    });
  }
</script>
<input id="eventText" type="text"
  oninput="validateAdverseEvent()">
<span id="error" style="color: red;"></span>
```

4.3.4 Choosing a JavaScript Validation Strategy

Approach	Best For	Server Required
IndexedDB	Local relational checks	No
AJAX + SQL	Large relational datasets	Yes
ML API (Flask)	AI-powered validation	Yes
TensorFlow.js	Local ML without server	No

4.4 The Spreadsheet-to-Lua-to-JavaScript Pipeline

This section describes what I consider the most novel and promising approach explored in this post: a pipeline where clinical team members define validation rules in a familiar spreadsheet format, a Lua script processes those rules into JavaScript validation functions, and a Shiny application loads the generated JavaScript for real-time client-side validation.

The key insight is that this pipeline separates the **rule definition** (accessible to non-programmers) from the **rule execution** (handled by generated code), while keeping the generation step lightweight and auditable through Lua.

Similar approaches exist in fragments. Medidata Rave uses a spreadsheet-like rule editor. OpenClinica supports ODK XLSForm rule definitions. Google Sheets combined with Apps Script can generate JavaScript validation. However, the specific combination of CSV rule storage, Lua code generation, and Shiny integration appears to be novel in the clinical data validation context.

4.4.1 Why This Pipeline

Component	Role	Benefit
Spreadsheet (CSV)	Rule storage	Non-programmers can edit
Lua	Code generator	Fast, lightweight, auditable
JavaScript	Rule execution	Real-time browser validation
Shiny	Application host	Dynamic reloading, reactive UI

4.4.2 Step 1: Define Rules in a Spreadsheet

Each field in the Shiny form receives a validation rule expressed as a JavaScript boolean expression in a CSV cell:

Field	Validation Rule
age	`age >= 18 && age <= 65`
email	`/^[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$/i.test(email)`
height	`height > 50 && height < 250`
bmi	`weight / (height / 100) ** 2 > 15 && weight / (height / 100) ** 2 < 50`

This format is readable by clinicians and data managers who may never write code. The CSV file (`validation_rules.csv`) serves as the single source of truth for all validation logic.

```
field,rule
age,age >= 18 && age <= 65
email,/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/.test(email)
height,height > 50 && height < 250
bmi,weight / (height / 100) ** 2 > 15 && weight / (height / 100) ** 2 < 50
```

4.4.3 Step 2: Lua Processes the Spreadsheet

A Lua script reads the CSV and generates a JavaScript file containing validation functions for each field:

```lua
local csv = require("luacsv")
local file = io.open("validation_rules.csv", "r")
local parsed_data = csv.parse(file:read("*all"))

local js_code = "const validationRules = {\n"

for i, row in ipairs(parsed_data) do
    local field, rule = row[1], row[2]
    js_code = js_code .. string.format(
        '    "%s": function(value) '
        .. '{ return %s; },\n',
        field, rule
    )
end

js_code = js_code .. "};\n"
file:close()

local js_file = io.open("validation.js", "w")
js_file:write(js_code)
js_file:close()

print("Validation JS generated successfully.")
```

Lua is well suited for this task. It is fast, lightweight, and excels at text processing. The LPeg library (https://www.inf.puc-rio.br/~roberto/lpeg/) provides additional parsing power if the rule syntax needs to become more sophisticated.

4.4.4 Step 3: JavaScript Integrates with Shiny

The generated `validation.js` file is included in the Shiny application, where it provides client-side validation without server round-trips:

```r
library(shiny)
library(shinyjs)

ui <- fluidPage(
  useShinyjs(),
  tags$head(
    tags$script(src = "validation.js")
  ),
```

```
  titlePanel(
    "Dynamic Validation Rules in Shiny"
  ),
  sidebarLayout(
    sidebarPanel(
      numericInput(
        "age", "Enter Age:", value = NULL
      ),
      textInput("email", "Enter Email:"),
      numericInput(
        "height", "Enter Height (cm):",
        value = NULL
      ),
      numericInput(
        "weight", "Enter Weight (kg):",
        value = NULL
      ),
      actionButton("submit", "Submit"),
      verbatimTextOutput("validation_message")
    ),
    mainPanel()
  )
)

server <- function(input, output, session) {
  observe({
    invalidateLater(5000, session)
    runjs(
      "delete window.validationRules; "
      %+% "$.getScript('validation.js');"
    )
  })

  observeEvent(input$submit, {
    runjs("
      let ageValid =
        validationRules['age'](
          parseFloat($('#age').val())
        );
      let emailValid =
        validationRules['email'](
          $('#email').val()
        );
      let heightValid =
        validationRules['height'](
          parseFloat($('#height').val())
```

```
        );
      let bmi =
        parseFloat($('#weight').val()) /
        ((parseFloat($('#height').val())
          / 100) ** 2);
      let bmiValid =
        validationRules['bmi'](bmi);

      let messages = [];
      if (!ageValid)
        messages.push('Invalid Age');
      if (!emailValid)
        messages.push('Invalid Email');
      if (!heightValid)
        messages.push('Invalid Height');
      if (!bmiValid)
        messages.push('Invalid BMI');

      if (messages.length > 0) {
        alert(messages.join('\\n'));
      } else {
        alert('All inputs are valid.');
      }
    ");
  })
}

shinyApp(ui, server)
```

The `invalidateLater(5000, session)` call reloads the validation JavaScript every five seconds, meaning that if someone updates the CSV and re-runs the Lua script, the new rules apply immediately without restarting the Shiny application.

4.4.5 Step 4: Database-Backed Rules (Extension)

Instead of reading from a CSV file, the Lua script can query a database for validation rules, enabling centralised rule management across multiple applications:

```
CREATE TABLE validation_rules (
    field TEXT PRIMARY KEY,
    rule TEXT
);

INSERT INTO validation_rules (field, rule) VALUES
('age', 'age >= 18 && age <= 65'),
('email',
```

```
 '/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+'
 || '+\\.[a-zA-Z]{2,}$/.test(email)'),
('height', 'height > 50 && height < 250'),
('bmi',
 'weight / (height / 100) ** 2 > 15 '
 || '&& weight / (height / 100) ** 2 < 50');
```

The corresponding Lua script uses `lsqlite3`:

```
local sqlite3 = require("lsqlite3")
local db = sqlite3.open("validation.db")

local js_code = "const validationRules = {\n"

for row in db:nrows(
    "SELECT * FROM validation_rules"
) do
    js_code = js_code .. string.format(
        '    "%s": function(value) '
        .. '{ return %s; },\n',
        row.field, row.rule
    )
end

js_code = js_code .. "};\n"
local js_file = io.open("validation.js", "w")
js_file:write(js_code)
js_file:close()

db:close()
print("Validation JS generated from database.")
```

And the Shiny server triggers regeneration:

```
server <- function(input, output, session) {
  observe({
    invalidateLater(5000, session)
    system("lua process_validation_db.lua")
    runjs(
      "delete window.validationRules; "
      %+% "$.getScript('validation.js');"
    )
  })
}
```

4.4.6 Pipeline Comparison: CSV vs Database

Feature	CSV-Based Rules	Database-Based Rules
Storage	Local file	Centralised database
Scalability	Small teams	Large-scale use
User interface	Spreadsheet editor	Web admin panel
Performance	Fast file read	Requires DB query
Versioning	Git-trackable	Requires DB history
Audit trail	File diffs	Query logs

4.5 Things to Watch Out For

1. **Security of generated JavaScript.** The Lua pipeline trusts that the CSV content is safe to embed as JavaScript. Malicious rule expressions could execute arbitrary code in the browser. Sanitise inputs before code generation.
2. **Polling interval trade-offs.** The five-second `invalidateLater()` reload is a balance between responsiveness and performance. For high-traffic applications, consider event-driven reloading instead of polling.
3. **Validation rule testing.** Generated JavaScript functions lack automated tests. Consider adding a test harness that validates each generated function against known inputs before deploying to the Shiny application.
4. **Cross-field dependencies.** The current pipeline generates independent per-field validation functions. Rules that depend on multiple fields (such as BMI requiring both weight and height) need careful handling in the JavaScript caller.
5. **Regulatory documentation.** In a GxP environment, the generated JavaScript constitutes part of the validated system. Ensure that the Lua generation step, the CSV rules, and the output JavaScript are all version-controlled and documented in the validation plan.

Figure 4: The pipeline from spreadsheet to browser validation bridges the gap between clinical domain expertise and software implementation.

5 What Did We Learn?

5.1 Lessons Learnt

Conceptual Understanding:

- Clinical trial validation is not a single check but a taxonomy of at least ten distinct categories, each addressing a different failure mode in data collection.
- The distinction between server-side validation (R, Python, SQL) and client-side validation (JavaScript) has direct implications for user experience and system architecture.
- Open-source EDC systems (OpenClinica, REDCap, ClinCapture) cover a surprisingly large fraction of the validation categories, though complex logic often requires custom code.
- The spreadsheet-as-rule-definition pattern separates domain knowledge from implementation, which is valuable in regulated environments where non-programmers must review validation logic.

Technical Skills:

- The R `validate` package provides a declarative, confrontation-based approach to data validation that integrates well with tidyverse workflows.

- The `shinyvalidate` package extends basic `validate()`/`need()` patterns with multi-field dependent validation and pre-submission checking.
- Lua's text processing capabilities make it a practical choice for lightweight code generation, particularly when the output language (JavaScript) differs from the processing language.
- IndexedDB provides a capable client-side relational database that can support local validation against reference datasets without network latency.

Gotchas and Pitfalls:

- Embedding user-defined expressions directly into generated JavaScript creates a code injection surface that must be mitigated through input sanitisation.
- The `invalidateLater()` polling pattern in Shiny is convenient but inefficient at scale; event-driven approaches (file watchers, WebSocket notifications) are preferable for production systems.
- REDCap's branching logic is powerful for simple rules but reaches its limits quickly with multi-field conditional validation.
- Generated validation code is only as correct as the rules in the source spreadsheet; there is no type-checking or static analysis step in the current pipeline.

5.2 Limitations

- This post surveys validation approaches without benchmarking their performance on realistic clinical datasets. Execution time comparisons would require a dedicated study.
- The Lua pipeline is presented as a proof of concept. Production deployment would require input sanitisation, error handling, logging, and integration testing.
- The comparison of EDC platforms is based on publicly available documentation rather than hands-on evaluation of current versions. Feature sets may have changed.
- The JavaScript examples use simplified error handling. Production clinical systems require more robust error reporting and audit logging.
- Regulatory compliance considerations (21 CFR Part 11, EU Annex 11) are mentioned but not addressed in detail. A compliance-focused review would be a separate undertaking.

5.3 Opportunities for Improvement

1. Build a complete working prototype of the Lua pipeline with input sanitisation, error handling, and a test suite that validates generated JavaScript against known inputs.
2. Add a Shiny-based admin panel where data managers can edit validation rules in a web interface that writes back to the CSV or database.
3. Extend the Lua code generator to support multi-field validation rules with explicit dependency declarations.
4. Integrate the pipeline with CDISC standards (SDTM/ADaM) so that validation rules can reference standard variable names and controlled terminology.
5. Evaluate WebAssembly as an alternative to JavaScript for running complex validation logic (including ML models) client-side without server round-trips.
6. Compare the Lua pipeline against existing spreadsheet-to-code tools (such as Google Apps Script) in terms of auditability, performance, and ease of adoption.

6 Wrapping Up

Clinical trial data validation spans a remarkable range of tools and techniques, from the ten categories of EDC edit checks through proprietary and open-source platforms to general-purpose languages and custom pipelines. The breadth of the landscape is itself a finding: there is no single tool that covers every validation need, and the practical choice depends on team composition, regulatory context, and the complexity of the validation logic.

The approach I find most promising is the spreadsheet-to-Lua-to-JavaScript pipeline. It addresses a genuine pain point in clinical data management: the gap between the people who understand the validation rules (clinicians, data managers) and the people who implement them (programmers). By putting the rule definitions in a format that non-programmers can read and edit, and using Lua to generate the execution code, the pipeline makes validation logic both accessible and auditable.

Main takeaways:

- EDC validation encompasses at least ten distinct check categories, each targeting a different data integrity failure mode.
- R (`validate`, `pointblank`), Python (`pandera`), and SQL provide flexible validation for custom analysis pipelines.
- Shiny supports three tiers of form validation: `validate()`/`need()`, `shinyvalidate`, and client-side JavaScript.
- A Lua-generated JavaScript pipeline can bridge the gap between non-programmer rule definition and real-time browser validation.

If you are working with clinical trial data validation, I would be interested to hear which combination of tools you have found effective in practice.

7 See Also

7.0.1 Related Posts

- Configure the Command Line for Data Science Development
- Building a Minimalist EDC Application

7.0.2 Key Resources

- R `validate` package documentation
- Python `pandera` documentation
- OpenClinica documentation
- REDCap project site
- Lua LPeg pattern matching library
- CDISC standards (SDTM/ADaM)
- `shinyvalidate` package

8 Reproducibility

This post is primarily a conceptual survey with embedded code examples. The code blocks are self-contained and can be executed independently in their respective language environments.

For the R examples:

```
sessionInfo()
```

For the Lua pipeline:

```
lua process_validation.lua
```

For the Shiny applications, save the code blocks as `app.R` and run with:

```
Rscript -e "shiny::runApp('app.R')"
```

Key files referenced in this post:

File	Purpose
`validation_rules.csv`	Spreadsheet rule definitions
`process_validation.lua`	CSV-to-JavaScript generator
`process_validation_db.lua`	DB-to-JavaScript generator
`validation.js`	Generated JavaScript rules
`app.R`	Shiny application with dynamic reload

9 Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.