PharmaSUG China 2026 - Paper AA-184

A Shiny-Based Tool for SAS and R Batch Execution and Log Review

Meng Zhang, Clinical Trial Service Co., Ltd.
Nan Zhao, Aistarfish Technology Co., Ltd.

ABSTRACT

Statistical programmers in clinical studies routinely submit large batches of SAS programs and then inspect every resulting log for errors, warnings, and a broader set of quality-relevant diagnostic messages such as character truncation, uninitialized variables, unresolved macro references, and zero-observation outputs. As R has been increasingly adopted for derivation of datasets and production of tables, listings, and figures, the same review effort now extends to a second language. Most existing batch utilities, however, are single-language tools - built exclusively for either SAS or R - leaving programmers to maintain two separate toolchains and two sets of log-review rules. This paper presents a lightweight Shiny-based toolkit (BatchTool) that unifies batch execution and log scanning across both SAS (sas.exe and SAS Enterprise Guide) and R (any installed version, with automatic renv pinning detection). The tool is deployed as two folders only: a shared root that lives at a fixed, team-agreed location and holds YAML rule files and the execution engine; and a tiny per-project front-end folder that is copied into each study and double-clicked to launch. We describe the architecture, core technical modules, serial batch route, and in-browser review workflow. Using an example folder derived from public clinical-programming examples, we demonstrate the folder layout, configuration points, rule examples, and run/log-check workflow so readers can reproduce the design or adapt individual components.

INTRODUCTION

Batch work in a clinical programming group is usually more than a shortcut for running many files. It is an operational route: define the programs to be submitted, run them in a documented order, create one log per program, and review the logs for failures and quality-relevant diagnostic messages. This route is familiar for SAS programs and becomes more complex when R programs are present in the same study area.

BatchTool was designed around that route. It provides one Shiny interface for discovering SAS and R programs, dispatching them to the appropriate runtime, recording logs in predictable locations, scanning those logs with external dictionaries, and returning the results to a browser-based review table. The following sections therefore follow the same sequence a programmer sees in practice: design route, folder structure, program loading, Shiny construction, language-specific execution, log scanning, and example review.

DESIGN ROUTE

The design route is serial and contract-based. The loader first creates a normalized program table. The batch engine then consumes that table and returns status and log path information for each row. The log scanner consumes the log path and returns counts plus hit lines. The Shiny layer is responsible for presentation and user interaction; it does not duplicate the execution and scanning rules. Figure 1 summarizes this route from discovery through browser review.

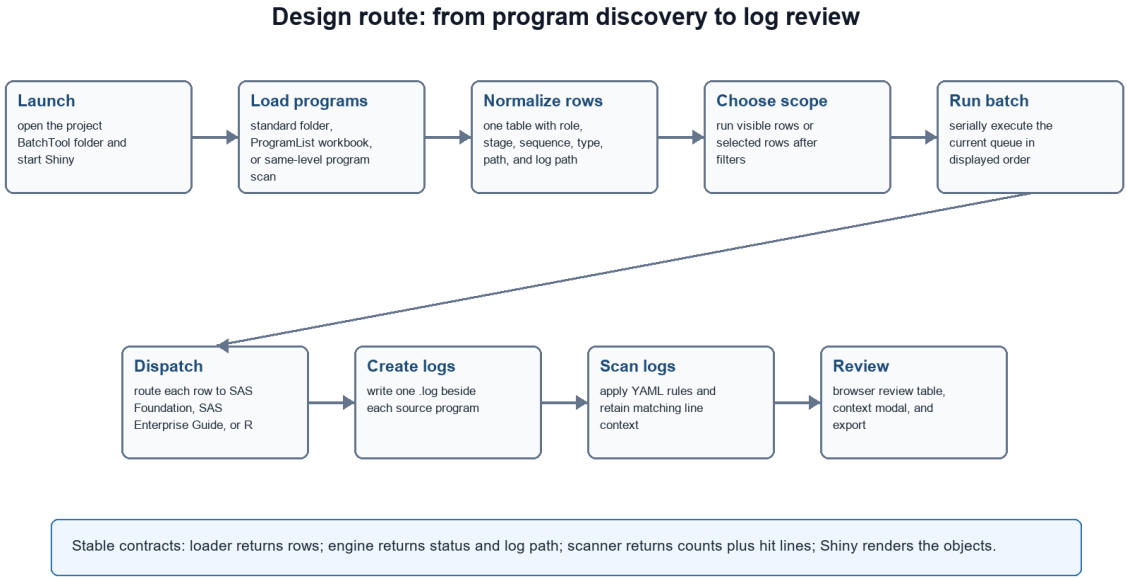

Figure 1. Serial route from program discovery to browser log review.

DEPLOYMENT ARCHITECTURE

BatchTool uses a two-folder model. The shared root is controlled once for a team or site and contains server configuration, YAML rule files, and the R modules for program loading, execution, and log scanning. The project side contains a small BatchTool launcher folder only. At launch, the project app sources the shared root and presents the local study programs through the same browser interface. Figure 2 shows the main deployment boundary.

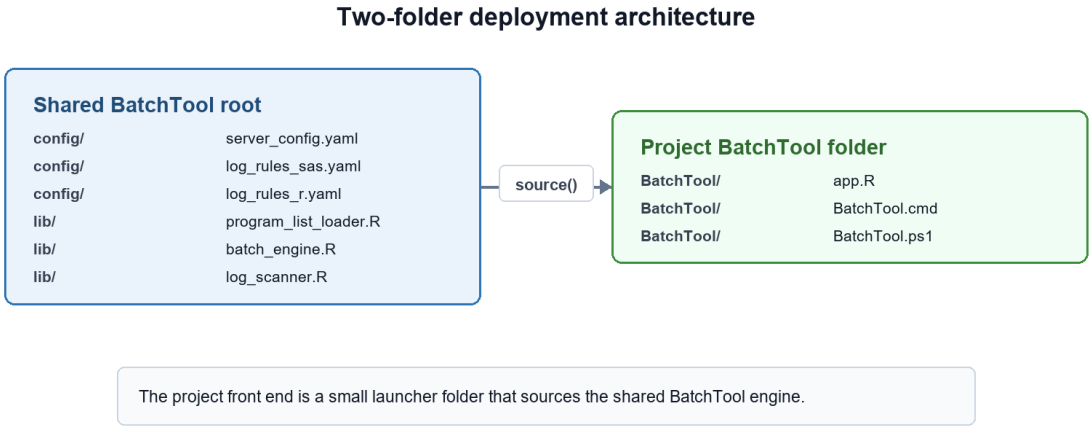

Figure 2. Two-folder deployment model with a shared engine and a per-project launcher.

PROGRAM LOADING

Program loading is intentionally separated from program execution. This separation lets users adjust the source of the program list without changing the batch engine. BatchTool supports three loading modes. Standard folder mode scans conventional stage folders. ProgramList workbook mode reads a structured spreadsheet for explicit sequencing or ownership. Auto sibling mode loads programs located at the same

folder level as the launcher. Figure 3 presents the three supported loading routes as inputs to the same normalized program table. This design keeps the loading rules visible to the user while preserving a stable technical interface for the batch engine.

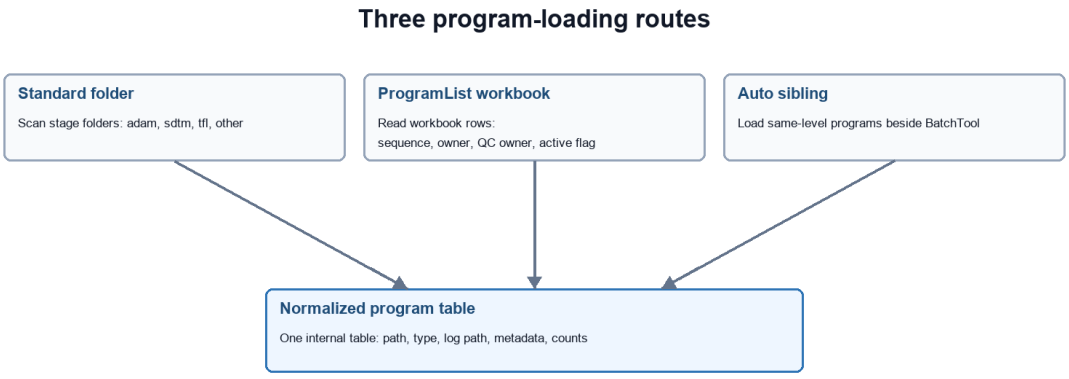

Figure 3. Three program-list inputs are converted to the same internal table.

The server-side switch in Code Display 1 chooses the loader and then standardizes the result. The excerpt is simplified to emphasize the contract; helper names stand for the corresponding path, type, and log-path derivations in the application.

Code Display 1. Load-source selection and normalization in the Shiny server:

```
observeEvent(input$btn_load, {
  src <- input$src_type
  df <- switch(src,
    standard = load_from_standard(PROGRAM_ROOT),
    xlsx     = load_from_xlsx(input$xlsx_upload$datapath),
    auto     = load_from_auto(dirname(BATCHTOOL_DIR))
  )
  if (src != "auto") {
    df <- resolve_program_paths(df, PROGRAM_ROOT)
  }
  df$type     <- vapply(df$full_path, detect_type, character(1))
  df$log_path <- vapply(df$full_path, derive_log_path, character(1))
  rv$programs <- ensure_cols(df)
})
```

After the switch, all loading routes are converted into the same program table. Each row has a program path, a detected language type, a planned log path, and the metadata columns used by the review table. Because the downstream object is the same, the Run and Log Check buttons work the same way regardless of how the list was loaded.

Figure 4 shows the standard example folder as a file tree. The loader treats stage folders as sources for executable rows. The R programs are derived from Pilot 3 of the R Consortium Submission Working Group, and the SAS programs are derived from the CDISC SDTM/ADaM Pilot Project. The root-level renv.lock supplies the preferred R version.

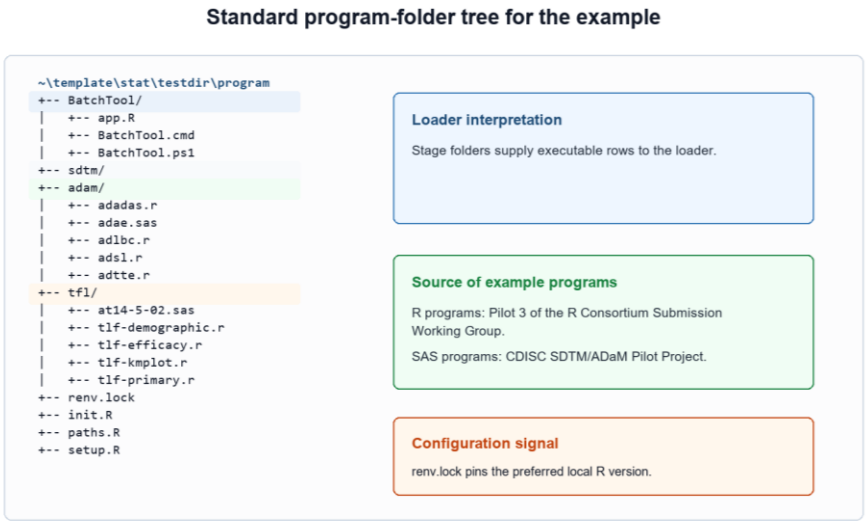

Figure 4. Standard program-folder tree populated with the example.

Figure 5 is the program-list screen after loading the example folder. It shows the three load modes, stage and type filters, the batch action buttons, the SAS engine selector, and the R version control locked to the detected renv pin. At this point logs have not been scanned, so the table is still a program queue rather than a finding summary.

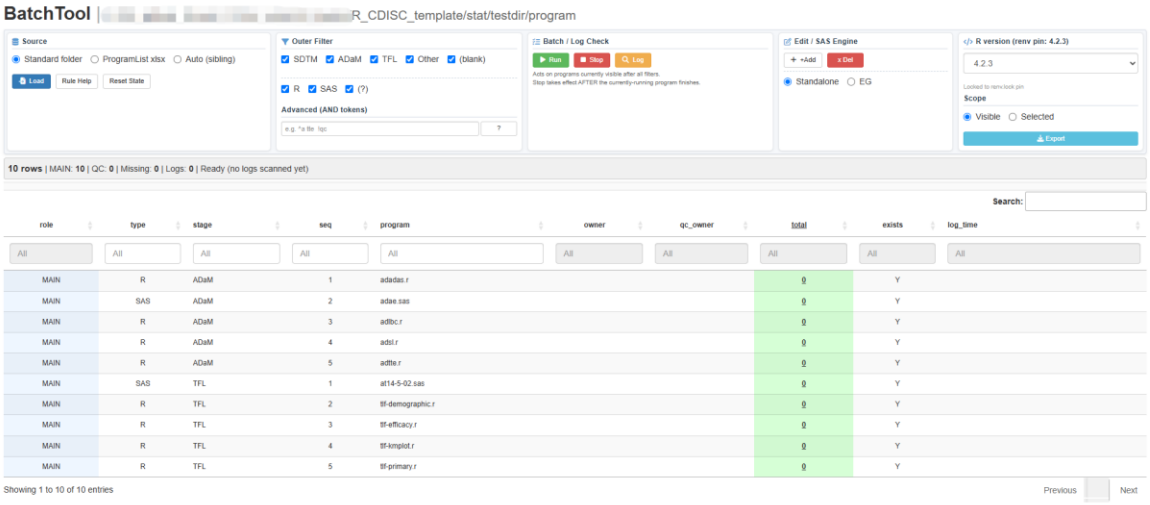

Figure 5. BatchTool after loading the example program folder.

SHINY FRONT-END CONSTRUCTION

Shiny is used as the browser layer for a local desktop workflow. At startup, app.R identifies the shared BatchTool root, sources the loader, execution, and scanner modules, and detects the active program root. The server stores runtime objects in reactiveValues: the loaded program rows, per-rule count matrix, scan objects, selected rows, and persisted UI choices. This design keeps the user interface responsive while retaining enough state for a later log review or export.

The screen is organized around the workflow rather than around implementation modules. The Source panel controls the loading mode. Outer Filter and the advanced token filter define the visible queue. Batch / Log Check contains the Run and Log actions. Edit / SAS Engine controls row editing and the SAS route. The R version panel displays the detected renv pin and the selectable Rscript executable. The review

table uses fixed identity columns, column filters, editable metadata fields, severity-colored count cells, and click-to-context behavior.

Code Display 2. Shiny state, event handlers, and review-table rendering:

```
source(file.path(ROOT, "lib", "log_scanner.R"))
source(file.path(ROOT, "lib", "batch_engine.R"))
source(file.path(ROOT, "lib", "program_list_loader.R"))

rv <- reactiveValues(
  programs = empty_df(),
  per_rule = NULL,
  log_scans = list(),
  sel_programs = character(0)
)

observeEvent(input$btn_run, do_batch_run())
observeEvent(input$btn_log, do_log_check())

output$tbl_main <- renderDT({
  df <- display_df()
  datatable(
    df,
    extensions = "FixedColumns",
    filter = "top",
    editable = list(target = "cell"),
    selection = list(target = "row", mode = "multiple"),
    options = list(scrollX = TRUE,
                   fixedColumns = list(leftColumns = fixed_left))
  )
})
```

At the Shiny layer, Code Display 2 acts as the orchestration surface. The source statements load the reusable modules from the shared root, reactiveValues keeps the current program list and scan results in memory, observeEvent connects toolbar buttons to batch and log-check actions, and renderDT builds the interactive review table with fixed left columns and top filters.

BATCH EXECUTION AND LOG CREATION

Batch execution begins from the visible or selected rows in the review table. Rows are executed serially so that the submitted order, status, and log path can be audited row by row. The engine dispatches by extension and, for SAS programs, by the selected SAS engine. BatchTool targets Windows for the SAS Foundation and SAS Enterprise Guide routes because these routes depend on sas.exe, PowerShell, and the Enterprise Guide COM automation object. Mechanistically, all three routes are started from the R/Shiny server process: SAS Foundation is launched as a direct sas.exe process through system2(); SAS Enterprise Guide is launched by system2() calling powershell.exe, and that PowerShell driver uses the EG COM automation object; R programs are launched by system2() calling the selected version with a temporary wrapper script. Figure 6 summarizes the three execution routes and their log creation mechanisms.

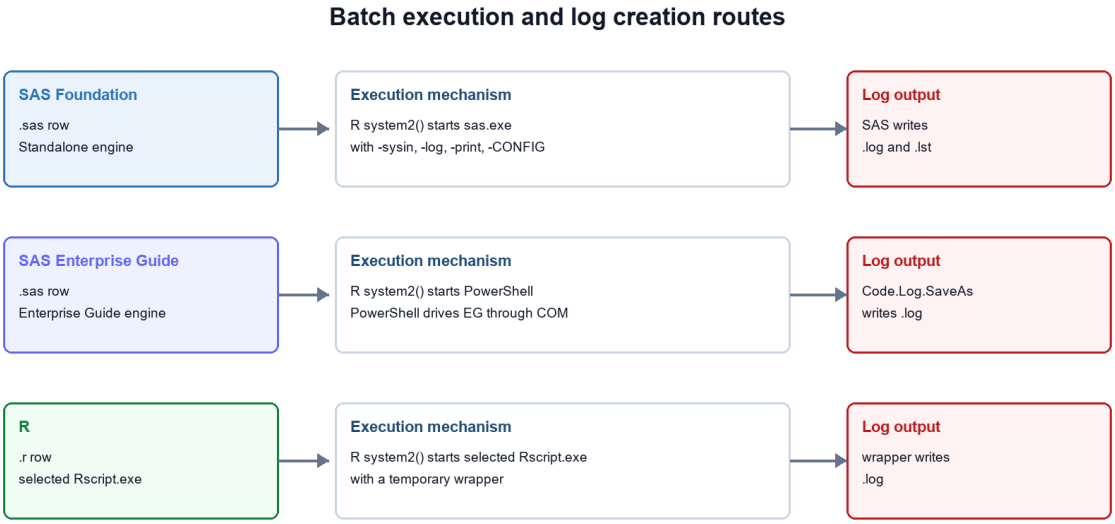

Figure 6. SAS Foundation, SAS Enterprise Guide, and R execution routes.

SAS-related execution settings are externalized in server_config.yaml. The same configuration block controls whether a SAS row is submitted through SAS Foundation or SAS Enterprise Guide, and it also holds the executable path, SAS configuration file, Enterprise Guide profile, and Enterprise Guide application identifier. For Enterprise Guide, the server connection details are kept in the named profile configured inside SAS Enterprise Guide; BatchTool records the profile name and activates it during automation.

Code Display 3. SAS and SAS Enterprise Guide configuration fields:

```
sas:
  engine: standalone
  exe_path: "C:/Program Files/SASHome/SASFoundation/9.4/sas.exe"
  config: "C:/Program Files/SASHome/SASFoundation/9.4/nls/u8/sasv9.cfg"
  eg_profile: "My Site Profile"
  eg_progid: "SASEGObjectModel.Application"
  eg_hide_window: true
```

Code Display 3 lists the configuration fields rather than executable logic. In practice, an administrator sets these values once for the local environment, and the Shiny front end only exposes the SAS engine choice to the user. The eg_profile value must match an Enterprise Guide profile that has already been configured and tested in the local SAS Enterprise Guide client.

SAS FOUNDATION ROUTE

The SAS Foundation route launches sas.exe as an operating-system child process from R. BatchTool calls R's system2() function with the configured sas.exe path and a vector of SAS command-line options; it is not a CMD script submission. The program path is supplied through -sysin, the log path through -log, the listing path through -print, and the Unicode SAS configuration through -CONFIG. SAS itself writes the log and listing files to the paths supplied by the batch engine.

Code Display 4. Core SAS Foundation execution call:

```
args <- c(
  "-RSASUSER", "-icon", "-nosplash",
  "-CONFIG", shQuote(cfg$sas$config),
  "-sysin", shQuote(program_path),
```

```
  "-log",   shQuote(log_path),
  "-print", shQuote(lst_path)
)

rc <- system2(cfg$sas$exe_path, args, stdout = TRUE, stderr = TRUE)
exit_code <- attr(rc, "status") %||% 0L
status <- exit_map[[as.character(exit_code)]] %||% paste0("EXIT=",
exit_code)
```

In the SAS Foundation route, BatchTool does not open SAS interactively and does not submit through SAS Enterprise Guide; it starts sas.exe directly, waits for the process to finish, then maps the returned exit status and the generated log path back to the program row.

SAS ENTERPRISE GUIDE ROUTE

The SAS Enterprise Guide route uses the configuration fields shown in Code Display 3, especially eg_profile and eg_progid. BatchTool writes a temporary PowerShell driver and runs it with system2("powershell.exe", ...). The driver creates a SAS Enterprise Guide COM application object, activates the configured profile, opens a new project, adds a Code object, submits the program text, and saves the log. Because the profile contains the server definition used by Enterprise Guide, BatchTool does not need to hard-code server host names in the YAML file. The production code also checks registered application identifiers and writes a fallback diagnostic log when the Enterprise Guide log cannot be saved.

Code Display 5. Core SAS Enterprise Guide automation steps:

```
$app = New-Object -ComObject $usedProgId
$app.SetActiveProfile($profile)
$proj = $app.New()
$code = $proj.CodeCollection.Add()

$programText = Get-Content -Path $srcFile -Raw -Encoding UTF8
$code.Text = $programText

$null = $code.Run()
try {
  $code.Log.SaveAs($logFile)
} catch {
  Set-Content -Path $logFile -Value $code.Log -Encoding UTF8
}
```

This PowerShell layer is intentionally small because the Enterprise Guide automation object is exposed through Windows COM. After SetActiveProfile() succeeds, the submitted program is executed on the SAS server associated with that Enterprise Guide profile, and Code.Log.SaveAs writes the review log to the path prepared by BatchTool.

R ROUTE AND R VERSION SELECTION

R execution is driven by Rscript. At startup, BatchTool detects installed R versions and their Rscript executables. It also walks upward from the launcher to find renv.lock. When the lockfile contains an R version that exists locally, that version becomes the default selection in the UI; otherwise, the newest detected local R version is used. The selected Rscript path is then copied into the runtime configuration before execution.

Code Display 6. R version pin detection and runtime selection:

```
detect_renv_r_version <- function(start_dir) {
```

```
  cur <- normalizePath(start_dir, winslash = "/", mustWork = FALSE)
  for (i in 1:8) {
    f <- file.path(cur, "renv.lock")
    if (file.exists(f)) {
      txt <- paste(readLines(f, warn = FALSE), collapse = " ")
      m <- regmatches(txt, regexpr('"Version"\\s*:\\s*"[0-9.]+"', txt))
      if (length(m) > 0) {
        return(sub('.*"Version"\\s*:\\s*"', "", sub('"$', "", m)))
      }
    }
    parent <- dirname(cur)
    if (parent == cur) break
    cur <- parent
  }
  NULL
}

cfg_run$r$rscript_path <- input$r_version %||% default_rscript()
```

The version-selection contract in Code Display 6 treats renv.lock as a pin rather than an executable script. The UI can therefore prefer the locally installed version that matches the project lockfile, while execution still happens through the selected other versions.

The R route writes a temporary wrapper script and submits that wrapper to the selected Rscript executable through system2(). The current wrapper is constructed so that the resulting log displays the R code being executed. This is useful during review because the log contains both the executed expressions and any output, messages, or errors produced by the program.

Code Display 7. R wrapper pattern for creating one log per program:

```
con <- file(log_path, open = "wt")
sink(con, split = FALSE)
sink(con, type = "message")

status_code <- 0L
tryCatch({
  if (has_init) source(init_R, echo = FALSE, local = FALSE)
  source(program,
         echo = TRUE,
         max.deparse.length = Inf,
         keep.source = TRUE,
         chdir = TRUE,
         local = FALSE)
}, error = function(e) {
  cat("\n!!ERROR!! ", conditionMessage(e), "\n", sep = "")
  status_code <<- 1L
})

sink(type = "message")
sink()
close(con)

writeLines(wrapper, wrapper_file)
system2(rscript,
        args = c("--no-save", "--no-restore", shQuote(wrapper_file)),
        stdout = TRUE, stderr = TRUE)
```

The R wrapper redirects output and messages into the target log with sink(), and source(..., echo = TRUE) mirrors the executed R expressions into the same log. The result is a single .log file per R program that can be reviewed with the same scanner contract used for SAS logs.

LOG REVIEW

Log review is YAML-driven. SAS and R use separate YAML rule files because the languages produce different diagnostic text, but the rule schema is the same. Each rule contains a key, a label, a pattern, an optional regex flag, and a severity. The scanner reads the appropriate rule file, reads the log, applies skip masks, matches each rule, and returns both summary counts and hit-line records. The scanner route is summarized in Figure 7.

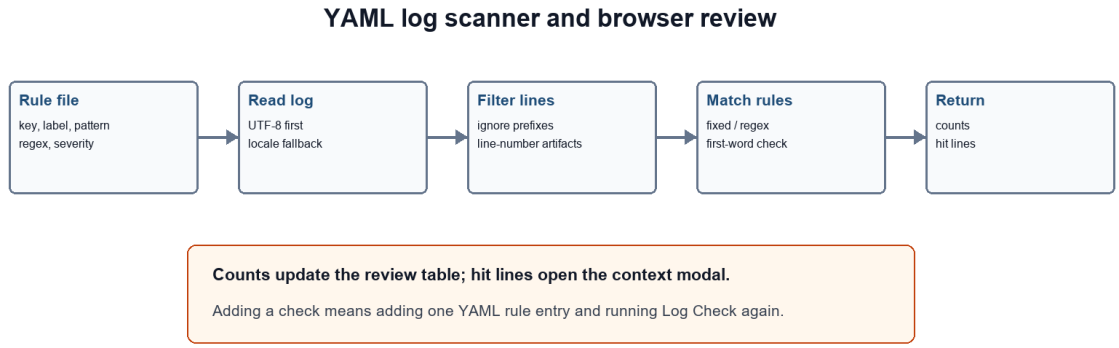

Figure 7. YAML rules are executed into count summaries and line-level context.

Code Display 8. Simplified YAML rule entries:

```
important:
  - key: ERROR
    label: "ERROR:"
    pattern: "(^|\\s)ERROR([: \\-]\\d|:)"
    regex: true
    severity: error

  - key: UNINIT
    label: "uninitialized variable"
    pattern: "is uninitialized"
    regex: false
    severity: warning
```

The YAML excerpt in Code Display 8 illustrates the rule schema rather than the full dictionary. key becomes the table column, label provides the human-readable name, pattern is either a fixed string or a regular expression, and severity determines how the count is colored in the review table.

Code Display 9. Core scanner loop:

```
rules <- yaml::read_yaml(rules_path)
all_rules <- c(rules$important, rules$self_check)

lines <- readLines(file(log_path, encoding = "UTF-8"), warn = FALSE)

for (r in all_rules) {
  mask <- if (isTRUE(r$regex)) {
    grepl(r$pattern, lines, perl = TRUE)
  } else {
```

```
      grepl(r$pattern, lines, fixed = TRUE)
    }
    hits <- which(mask & !skip_mask)
    summary_rows[[r$key]] <- data.frame(
      key = r$key,
      severity = r$severity,
      count = length(hits)
    )
    hit_rows[[r$key]] <- data.frame(
      key = r$key,
      line_no = head(hits, max_hits_per_rule),
      text = lines[head(hits, max_hits_per_rule)]
    )
  }
```

The loop in Code Display 9 reads the rule file, evaluates each rule against the log lines, records a count for the table, and stores the first matching line records for the context modal. Rule maintenance is therefore a configuration task. To add a new check, a maintainer adds one YAML entry under the appropriate language file and assigns a key, label, pattern, regex mode, and severity. On the next log check, the scanner reads the file and the new count column becomes available through the same review-table mechanism.

EXAMPLE

The example folder contains both SAS and R programs. After loading the standard folder, BatchTool detects 10 MAIN rows: five ADaM programs and five TFL programs. Two rows are SAS programs and eight rows are R programs. In Figure 8, the same application has completed log scanning; non-zero cells become review entry points, while zero-count cells are de-emphasized.

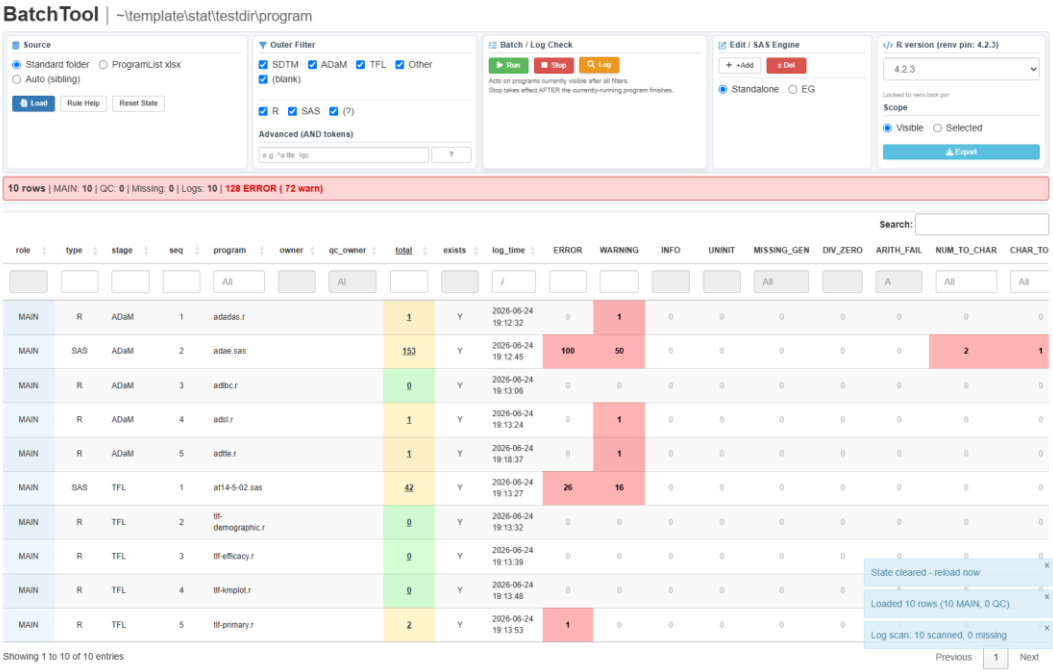

Figure 8. Review table after log scanning the example folder.

The next two figures show the review action itself. The user clicks a non-zero rule cell, and Shiny opens a modal with the log path, rule label, hit index, matching line, and surrounding context. The modal behavior is identical for SAS and R logs because both routes satisfy the same log-path and scanner contract.

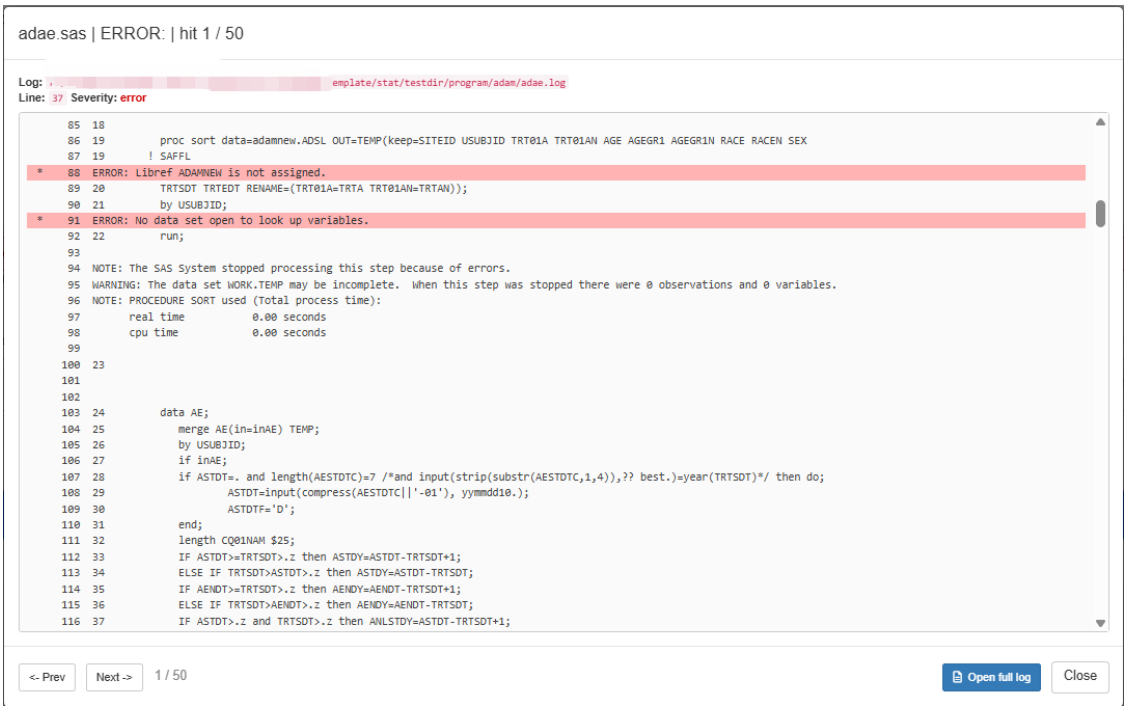

Figure 9. SAS log context opened from a counted ERROR cell.

Figure 10. R log context opened from the same review workflow.

CONCLUSION

BatchTool demonstrates a compact architecture for mixed SAS and R batch operations. Its main design principle is to keep the operational route explicit: discover programs, normalize rows, execute them serially, write one log per program, scan the logs with external rules, and review the results in Shiny. The example shows that SAS Foundation, SAS Enterprise Guide, and R can be coordinated through one

browser interface while leaving the study program folder and the rule dictionary understandable to a programming team. The current implementation is intended for Windows study environments, and its serial execution model is a deliberate choice to keep order, status, and log paths easy to audit.

REFERENCES

YAML Language Development Team. "YAML Ain't Markup Language: Libraries." Available at https://yaml.org/libraries/.

R Consortium. "submissions-pilot3-adam-to-fda." GitHub repository. Available at https://github.com/RConsortium/submissions-pilot3-adam-to-fda.

CDISC. "sdtm-adam-pilot-project." GitHub repository. Available at https://github.com/cdisc-org/sdtm-adam-pilot-project.

CONTACT INFORMATION

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

Meng Zhang
Clinical Trial Service Co., Ltd.
eamon.zhang12@gmail.com

Nan Zhao
Aistarfish Technology Co., Ltd.
zhaonn132@163.com