Data Analysis Initiation Checklist

A 55-item walkthrough for zzcollab projects

Ronald 'Ryy' G. Thomas

2026-04-29

A six-phase, 55-item checklist for initiating a reproducible data analysis inside a zzcollab research compendium. Each item names a verifiable end state with a short paragraph describing what to verify and how to verify it. The checklist applies to clinical trial data, observational cohorts, simulation studies, and any other tabular analysis that fits the zzcollab compendium layout.

Table of contents

2026-04-29 07:24 PDT

Instructions

Work through items sequentially. Each item names a verifiable end state, followed by a short annotation paragraph describing what to verify and how to verify it. Reference items by number when discussing them with collaborators or in commit messages (for example, 'Item 12 is failing in the new tinytest file').

Items that do not apply to a particular project should be marked 'N/A' with a one-line reason rather than removed. The numbering is meant to be stable across projects so that two analysts can compare their progress unambiguously.

Phase	Items	Deliverable
1	1-8	A buildable workspace with documented raw data
2	9-22	A cleaned, validated derived dataset
3	23-31	An EDA report with Table 1 and trajectory plots
4	32-38	Tested R functions for summary, modelling, plotting
5	39-45	A primary-analysis report with sensitivity analyses
6	46-55	A reproducible compendium ready for archival

Phase 1: Project Setup

1. **Verify `Dockerfile` exists and builds successfully.**

   Run `zzc docker --build` from the project root. Confirm the build completes without error and that `docker images <project>` shows a recent image. The Dockerfile defines the computational environment that every subsequent step will run inside; an unbuildable Dockerfile blocks every phase that follows.

2. **Verify `renv.lock` contains required packages.**

   Run `list-renv-packages.sh` or inspect `renv.lock` with `jq`. The lockfile should list only direct dependencies at this stage; transitive dependencies are resolved by `renv::restore()` inside the container. The lockfile is the second of the Five Pillars and pins the package versions that make the analysis reproducible.

3. **Verify `.Rprofile` has appropriate R options.**

   Open `.Rprofile` and confirm it calls `source('renv/activate.R')` and sets `options(renv.config.auto.sn = TRUE)`. The zzcollab template also configures auto-restore on session start so that a fresh container picks up the lockfile automatically.

4. **Verify source code directories exist.**

   Run `ls -d R/ analysis/scripts/ analysis/data/ tests/` to confirm the rrtools directory skeleton is in place. Create any missing directories with `mkdir -p`. The layout is the contract the rest of the checklist relies on; if `R/` does not exist, Phase 4 has nowhere to put its functions.

5. **Verify raw data is in `analysis/data/raw_data/` (read-only).**

   Confirm the raw data files are present at the documented path. Set them read-only with `chmod 444 analysis/data/raw_data/*` to prevent accidental modification. Never overwrite raw data; if a correction is needed, treat it as a new dataset and document the change in the data README.

6. **Update `analysis/data/README.md` with data source, date received, and use restrictions.**

   Document who provided the data, when it was received, any IRB or ethics-board protocol numbers, and whether the data contain PHI or require a data use agreement. This file serves as provenance metadata for the raw data directory and is the first artefact a future reader will look for.

7. **Record data dictionary: column definitions, units, coding schemes.**

   Create `docs/data-dictionary.md` (or append to the data README). For each column in the raw dataset, record the variable type, allowed values or range, units, and the clinical or scientific interpretation. Treat the data dictionary as a contract between the data provider and the analyst: the integrity tests in Phase 2 should validate the contract, not the data.

8. **Install analysis packages: `tidyverse`, `here`, `janitor`, `skimr`.**

   Enter the container with `make r`, then run `install.packages(c('tidyverse','here','janitor', 'skimr'))`. After installation, run `renv::snapshot()` to record the exact versions in `renv.lock`. Add each package to the DESCRIPTION Imports field so the declared dependency set matches the installed environment.

Phase 2: Data Ingestion and Validation

9. **Create `R/load_data.R` with a function to read raw data.**

   Write a `load_raw_data()` function that calls `readr::read_csv()` (or the appropriate reader) on the raw file. Some CSV exports carry a UTF-8 BOM in the header; `read_csv()` handles this automatically. Use `here::here()` for the file path so the function works from any working directory.

10. **Create `inst/tinytest/test_data_integrity.R`.**

    Create a tinytest file that sources `load_raw_data()` and runs assertions against the resulting data frame. Each subsequent item (11-17) becomes one or more `expect_*` calls in this file. Run with `tinytest::run_test_file()`. The test file is the living version of the data dictionary contract.

11. **Test: expected number of columns.**

    Use `expect_equal(ncol(dat), N_EXPECTED_COLS)` to confirm the raw data has the documented number of columns. Define `N_EXPECTED_COLS` at the top of the test file as a named constant referencing the data dictionary. A mismatch signals that the upstream data export has changed or that the file was corrupted during transfer.

12. **Test: expected column names present.**

    Define a character vector of the expected column names from the data dictionary and assert `expect_true(all(expected %in% names(dat)))`. A failure indicates the data provider has renamed a field or that an export script has changed; either way, the analysis cannot proceed without resolving it.

13. **Test: ID columns have no NAs and follow expected format.**

    Confirm that participant or unit identifier columns have no missing values. Use `expect_true(all(!is.na(dat$id_col)))`. Optionally assert that IDs follow the documented format (positive integers, fixed-width strings, or a regular expression pattern). A missing identifier breaks every downstream join.

14. **Test: pre-specified categorical variables match allowed levels.**

    For each categorical variable in the data dictionary (treatment arm, study site, cohort), assert that the observed values are exactly the documented allowed set. Use `expect_true(all(dat$x %in% allowed_levels))`. Catches typos, encoding drift, or coding changes by the data provider before they propagate into models.

15. **Test: demographic categorical variables match coding scheme.**

    For demographic factors (sex, race or ethnicity, education category) assert membership in the documented coding scheme. Investigate any unexpected values; do not silently recode them. Differences from the documented set are sometimes legitimate (a new site recruited a wider population) and sometimes errors; only the data provider can tell.

16. **Test: time or visit variables fall in the pre-specified set.**

    For longitudinal designs, assert that visit identifiers fall in the pre-specified set of timepoints. For continuous timestamps, assert plausible date ranges (`expect_true(all(dat$visit_date >= study_start & dat$visit_date <= today()))`). Out-of-range timestamps often indicate a parsing error rather than a data error.

17. **Test: numeric columns are within plausible ranges.**

    For each numeric variable in the data dictionary, assert clinically or scientifically plausible bounds. Use `expect_true(all(x >= lo & x <= hi, na.rm = TRUE))` so that missing follow-up data does not trip the check. A failure means either the data dictionary is wrong or the data have changed since the dictionary was written; both are worth knowing early.

18. **Create `analysis/scripts/01-clean-data.R`.**

    This script sources `load_raw_data()`, applies cleaning steps (Items 19-21), and writes the result to derived data (Item 22). Structure it as a linear pipeline using `|>` so the transformations are auditable in sequence. Avoid mutating state that lives outside the pipe; the script should be readable top to bottom.

19. **Clean column names using `janitor::clean_names()`.**

    Pipe the raw data through `janitor::clean_names()` to convert all column headers to snake_case. This normalises any BOM-affected first column name and makes all names consistent for downstream code. Update the data dictionary to record both the raw and cleaned name for every column.

20. **Convert categorical variables to factors with explicit levels.**

    Convert each categorical variable to a factor with explicitly ordered levels. Place the reference level first (control before treatment, baseline before follow-up) so model contrasts produce intuitive coefficients. Document the choice of reference level in a code comment so a future reader does not have to re-derive it.

21. **Handle missing values explicitly and document approach.**

    Identify which columns have NAs and whether missingness is structural (for example, follow-up measurements absent at baseline) or unexpected. Document the decision for each pattern in a code comment: kept as-is, imputed, or flagged for exclusion. Do not silently drop rows; an inadvertent `na.omit()` can change the analysis population without notice.

22. **Save cleaned data to `analysis/data/derived_data/`.**

    Write the cleaned data frame with `readr::write_csv()` or `saveRDS()` to `analysis/data/derived_data/cle`
    All downstream scripts read from derived data, never from raw. This ensures the cleaning step is the single point of transformation and that the raw files remain untouched.

Phase 3: Exploratory Analysis

23. **Create `analysis/scripts/02-eda.R`.**

    Create the EDA script as the second step in the linear pipeline. It reads cleaned data from derived data, never from raw, and produces tabular and graphical summaries. Like `01-clean-data.R`, structure it as a linear pipe so the order of operations is auditable. Source the script from a fresh R session to confirm it runs end to end.

24. **Generate summary statistics by treatment group.**

    For each numeric outcome and each pre-specified grouping (treatment arm, sex, time point), produce mean, SD, median, IQR, min, max, and count of non-missing observations. The `dplyr::summarise + across` pattern or `skimr::skim |> dplyr::group_by(group)` gives a compact tabular report. Save the resulting summary as a CSV in derived data so it can be referenced from the report without re-computation.

25. **Check baseline balance.**

    At the baseline timepoint (or whichever visit pre-dates the intervention), compare distributions across treatment arms. A Table 1 of demographics and baseline outcomes is usually sufficient. For randomised trials, imbalance at baseline is informative for downstream covariate adjustment, not a basis for re-randomisation; for observational studies it informs the propensity model.

26. **Visualise outcome distributions.**

    Plot histograms or density curves for each numeric outcome, faceted or coloured by group. Inspect for floor and ceiling effects, bimodality, heavy tails, and outliers. Use `ggplot2` with `facet_wrap` or `facet_grid` for cross-classification. Save plots to `analysis/figures/` at fixed dimensions for inclusion in the report.

27. **Identify potential outliers or data quality issues.**

    Flag observations that fall outside the plausibility ranges from Item 17, or that deviate substantially from the within-group distribution. Boxplots and dotplots reveal outliers efficiently. Document each flagged observation in a tracking file with the decision (keep, exclude, query data provider) so that downstream analyses are auditable.

28. **Create `analysis/report/01-eda.Rmd` (or `.qmd`).**

    The first report is a stand-alone Rmd or qmd that knits to HTML or PDF. It loads cleaned data, runs the EDA script (or reads its outputs from derived data), and presents the results in narrative form. Place the report alongside any project-specific bibliography and citation style. Render it with `quarto render` or `rmarkdown::render()` inside the container.

29. **Document sample size and missingness patterns.**

    Report the number of participants enrolled, randomised, and analysed at each timepoint. Provide a CONSORT-style flow diagram for clinical trials (the `consort` R package or a hand-drawn `DiagrammeR` diagram). For longitudinal studies, summarise per-visit attrition. Tabulate missingness by variable and by visit so the reader can judge the analytic implications.

30. **Include baseline characteristics table (Table 1).**

    Generate Table 1 with `gtsummary::tbl_summary()` or `table1::table1()`. Stratify by the primary grouping variable. Report continuous variables as mean (SD) or median (IQR) depending on distribution; categorical variables as n (%). For randomised trials, avoid statistical tests of baseline balance; they are uninformative when randomisation succeeded.

31. **Include outcome trajectory plots by visit and group.**

    For each primary outcome, plot the mean (or median) trajectory over visits, with separate lines for each group and error bars or ribbons for variability. Spaghetti plots showing individual trajectories are a useful supplement when sample sizes are modest. Save figures at publication-ready dimensions (typically 6x4 inches for single-column, 8x5 inches for full-page).

Phase 4: Analysis Functions

32. **Create `R/summarize_outcomes.R` with descriptive statistics functions.**

    Define functions that compute descriptive statistics from cleaned data and return tidy data frames. Each function takes a data frame and returns a data frame, never a printed object. Document arguments and return values with `roxygen2`. Place these functions in `R/` rather than in analysis scripts so they can be unit tested and re-used across reports.

33. **Create `R/model_outcomes.R` with statistical modelling functions.**

    Define modelling functions that take a data frame and an outcome name and return a fitted model object (or a tidy summary). Wrap each call to `lm()`, `lmer()`, or `glm()` in a function so the model specification is captured in code rather than reproduced inline. Return `broom::tidy()` output alongside the raw model when the downstream consumer wants both estimates and the underlying object.

34. **Create `R/plot_outcomes.R` with visualisation functions.**

    Define plotting functions that take a data frame and return a ggplot object. Plotting functions should not call `ggsave()` internally; that is the caller's responsibility. Returning the ggplot object lets reports compose multiple panels with `patchwork` or arrange them flexibly. Document the expected column names with `assertthat::assert_that()` at the top of the function body.

35. **Create `inst/tinytest/test_summarize.R`.**

    For each function in `summarize_outcomes.R`, write at least one tinytest case using a small, hand-constructed data frame whose summary statistics can be computed by hand. The test asserts that the function output equals the hand-computed result. Include at least one test per public function and at least one for missing-data handling.

36. **Create `inst/tinytest/test_model.R`.**

    Test modelling functions against simulated data with known parameters. Generate a small dataset where you control the true coefficient, fit the model, and assert that the recovered estimate is within tolerance. Use `set.seed()` for reproducibility. Test both happy paths (well-conditioned data) and edge cases (perfect collinearity, single-level factor).

37. **Test edge cases: missing data handling.**

    For each function, write tests that pass in data with NAs in different columns. Assert the documented behaviour: do summarise functions skip NAs by default, propagate them, or error out? Whichever behaviour is chosen, lock it down with a test so future refactors do not silently change it.

38. **Test edge cases: single-group scenarios.**

    Test that grouping functions behave sensibly when a stratum has zero observations or only one level. Assert either a graceful empty result or a clear error message. Hidden bugs in these edge cases often appear late in a project when a pre-specified subgroup turns out to be empty in the actual sample.

Phase 5: Primary Analysis

39. **Create `analysis/scripts/03-primary-analysis.R`.**

    The third script in the linear pipeline. It loads cleaned data, applies the modelling functions from `R/model_outcomes.R`, and writes fitted model objects and tidy result tables to `analysis/data/derived_data/`. Avoid putting modelling logic directly in the script; the script should orchestrate, while `R/` holds the reusable code.

40. **Implement the pre-specified analysis plan.**

    Follow the statistical analysis plan (SAP) line by line. For each pre-specified hypothesis, fit the documented model with the documented covariates and the documented contrast. Code each pre-specified comparison so that re-running the script produces the registered result. Mark any deviation from the SAP with a code comment that begins `# SAP DEVIATION:` and document the reason.

41. **Save model objects to `analysis/data/derived_data/`.**

    Persist each fitted model with `saveRDS()` so that downstream report code does not have to re-fit the model. Use a naming convention such as `model_<outcome>_<model_type>.rds`. Also save the `broom::tidy()` and `broom::glance()` outputs as CSV for direct inclusion in tables. The derived data folder becomes the single source of truth for model results.

42. **Create `analysis/report/02-results.Rmd` (or `.qmd`).**

    The results report loads the saved model objects and renders the result tables and figures. It should contain no model-fitting code; if it does, the script and report can drift apart. Cite the SAP and the EDA report so the reader can follow the linear chain of reasoning from raw data through derived data to statistical inference.

43. **Document primary outcome analysis in the report.**

    Present the primary outcome model first, with a results paragraph that includes the point estimate, confidence interval, and inferential statistic for the pre-specified primary contrast. Match the wording in the SAP. Include the model formula in a methods footnote so the reader can reproduce the analysis from text alone.

44. **Document secondary outcomes in the report.**

    Present each secondary outcome with the same structure as the primary: model, contrast, estimate, CI, and inferential statistic. Acknowledge the multiple-comparison context; either pre-specify a correction (Bonferroni, Holm, hierarchical testing) or label the analyses as exploratory and report unadjusted estimates with that caveat.

45. **Document sensitivity analyses in the report.**

    For each modelling assumption that could plausibly affect the conclusion, run a sensitivity analysis and document its result alongside the primary. Common sensitivity analyses include: complete-case versus multiple imputation, alternative covariate sets, alternative outcome scales, alternative time-window definitions. The reader should leave the report knowing which conclusions are robust and which depend on a specific modelling choice.

Phase 6: Documentation and Reproducibility

46. **Update `DESCRIPTION`: add a meaningful Title and Description.**

    Replace placeholder Title and Description fields with content that accurately summarises the project. Title is one sentence, fewer than 65 characters. Description is one to three sentences naming the study population, primary outcome, and analytic approach. R CMD check enforces basic format requirements; rendered correctly, these fields populate package metadata visible to anyone who installs the project from source.

47. **Update `DESCRIPTION`: add or verify Authors.**

    Use the Authors@R field with `person()` entries listing each contributor's role (`aut`, `cre`, `ctb`) and ORCID where available. The creator (`cre`) is the contact for installation problems. R CMD check enforces a valid Authors@R; do not silently leave the placeholder 'Your Name'.

48. **Verify all package dependencies listed in `DESCRIPTION`.**

    Cross-check the Imports field against the packages actually loaded in `R/` and `analysis/scripts/`. Use `renv::dependencies()` to list every package referenced. Any package called but not declared will fail in a clean container. The DESCRIPTION should list direct dependencies only; `renv.lock` pins the transitive closure.

49. **Create or update the project `README.md`.**

    The README is the entry point for anyone (including future you) reading the repository for the first time. State the research question, the data source, the directory layout, and the commands to reproduce the analysis. Include a directory tree (output of `tree -L 2` trimmed to the meaningful top level). Link to the rendered report PDF if hosted online.

50. **Document reproduction instructions in the README.**

    Provide a copy-pasteable sequence of shell commands that reproduces the analysis from a clean checkout. Typically: `git clone`, `make docker-build`, `make r`, source the scripts in order, render the reports. Test these instructions on a machine that does not already have the project set up; instructions that work only on the author's laptop are not reproduction instructions.

51. **Rebuild Docker image: `make docker-build`.**

    After all DESCRIPTION and renv changes, run `make docker-build` to produce a fresh image. Tag it with the current git SHA so future readers can identify the exact build that produced the published results. The build should complete with no warnings about missing packages; treat warnings as errors at this stage.

52. **Verify reproducibility: run all scripts in a fresh container.**

    Run the entire pipeline in a fresh container started from the new image. Source `01-clean-data.R`, `02-eda.R`, `03-primary-analysis.R` in order, then render each report. The end-to-end run is the canonical reproducibility test; passing it certifies that the Docker image, renv lockfile, and source code together produce the documented outputs.

53. **Confirm all outputs match expected results.**

    Compare the freshly produced derived data, figures, and report PDFs against the previously committed versions. Bit-for-bit reproducibility is a strong signal; numerically identical floating-point results without timestamp or render-metadata noise is the realistic target. Any divergence that is not explained by a documented code change requires investigation before publication.

54. **Run `make test` and verify all tests pass.**

    All tinytest and testthat suites must pass in the fresh container. A failing test at the publication stage is a stop-the-line event: investigate the failure, fix the code or the test, and re-run the entire reproducibility check. Do not skip or deactivate a failing test to clear the gate.

55. **Final review: all Five Pillars complete and consistent.**

    The zzcollab Five Pillars (Dockerfile, renv.lock, .Rprofile, source code, data) must be coherent: the Dockerfile references the same R version that `renv.lock` was created against; `.Rprofile` activates renv so the lockfile loads automatically; source code in `R/` and `analysis/` runs in that environment; data is read-only and documented. Walk through each pillar one last time and confirm that the artefacts on disk tell a single, consistent story about how to reproduce the analysis.

Total items: 55. Last updated 2026-04-29.