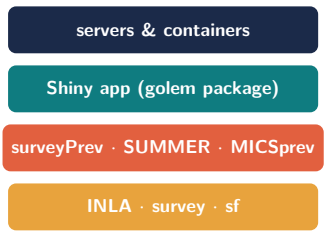
Under the Hood: Building the sae4health Platforms

R Shiny apps, servers, packages, data pipelines,
static websites, and AI in the workflow

The sae4health team

Technical session

Small Area Estimation for Improving Health in LMICs



What this session covers

1. The family of platforms, and how they relate
2. The software stack, layer by layer
3. Inside a Shiny app: golem, modules, state
4. Where the data lives, and why
5. Reproducible environments with **renv**
6. Deployment: Posit Connect and Docker
7. sae4lmic: a results-only static website
8. AI in the development workflow
9. Lessons learned, and advice for builders

Audience: anyone who wants to build, maintain, or extend a platform like these. Everything here comes from the actual repositories (`wu-thomas/sae4health`, `LucyXuu/SurveyPrevRShiny`, `richardli/SAE4LMIC`, `UW-Statistics/gatesweb`) as of November 2026.

The platform family

One engine, several front ends

Platform	What it is	Built with	Hosted at
DHS app	Interactive SAE for 150+ DHS indicators, 60 countries; guided model choice	**sae4health** (R package, CRAN, AGPL-3)	Posit Connect at UW
MICS app	Same workflow for MICS surveys; Nigeria 2021 preloaded with 48 indicators	**SurveyPrevRShiny** + **MICSprev**	Posit Connect at UW
Variants	WHO version (`version='DHS-WHO'`, WHO shapefiles), country editions	Same code, run-time options	Connect / local
sae4lmic	Pre-computed, model-checked estimates for 13 countries, ∼20 indicators	R pipeline → MkDocs Material	GitHub Pages
Docs site	Documentation, videos, methods, gallery, blog	Static site	`sae4health.stat.uw.edu`

The apps are **tools** (users fit models). The sae4lmic site is a **dissemination** product (users read results). They need very different engineering.

The repositories by the numbers

	sae4health (DHS app)	**SurveyPrevRShiny** (MICS app)
R source files	40	30
Lines of R code	27,968	17,296
Shiny modules (`mod_*.R`)	15	14
Packages in `renv.lock`	219	248
R version in lockfile	4.4.3	4.5.1
INLA version	25.12.16	26.06.07
Commits	140	37
First commit	Mar 2025	Dec 2024
Repo size on disk	∼10 MB	∼1.1 GB

- The two biggest modules are **report generation** (2,894 lines) and **model selection** (2,518 lines), not the maps.
- The MICS repo is large because it ships **shapefiles** (GADM: 359 MB) and preloaded data inside the repo.
- Development is a team effort: about ten commit identities (several people under more than one email) in the DHS app.

Counted from the cloned repositories, November 2026. Commit counts are from `git log` and are not shown on GitHub's own page.

The software stack

Layers, from statistics to the browser

Browser Shiny's JavaScript client; **leaflet** maps, **plotly**, **DT** tables, custom JS handlers

Host Posit Connect (UW) or a Docker container running `shiny::runApp`

App an R package built with **golem**: modules, R6 state, **shinydashboard** UI

Workflow **surveyPrev** (indicators, direct/FH/unit models), **SUMMER**, **MICSprev** (MICS indicators)

Engines **INLA** (Bayesian fits), **survey** (design-based), **sf**/GDAL/GEOS/PROJ (geometry)

pinned by **renv**

Data DHS/MICS recodes, GPS, shapefiles; metadata from the DHS API (**rdhs**)

Each layer is maintained separately. A bug can sit in any of them, so it pays to know which layer you are in.

Why build the app on top of packages?

Statistics live in packages

- **surveyPrev** holds the indicator definitions and the model code (`directEST`, `fhModel`, `clusterModel`).
- The same functions power the app, the sae4lmic pipeline, and scripted analyses. One fix reaches all three.
- Methods are tested and versioned independently of the UI.

The app is also a package

- **sae4health** is on CRAN: `install.packages("sae4health")`, then `run_app()`.
- Users without a web connection, or with their own shapefiles, run it locally.
- Package structure gives documentation, dependency declarations and tests for free.

Dependencies on **GitHub development versions** (**surveyPrev**, **SUMMER**, **MICSprev**) are convenient but fragile: the CRAN release of **surveyPrev** lacks functions the pipeline uses, and an old copy in a user library can silently shadow the new one.

Inside a Shiny app

golem: a Shiny app as an R package

```
sae4health/
  DESCRIPTION   # deps, version
  NAMESPACE
  app.R         # entry for Connect
  R/
    app_ui.R  app_server.R
    run_app.R
    mod_*.R     # 15 modules
    fct_*.R     # helpers
    utils_*.R
  inst/app/www/ # CSS, JS, Rmd
  data/         # .rda metadata
  tests/testthat/
  renv.lock  Dockerfile
```

- **Modules** (`mod_country_specify`, `mod_model_selection`, `mod_res_visual_ridge`, …) each pair a UI and a server function with their own namespace. One sidebar item is roughly one module.
- `fct_*` files hold non-reactive logic (data input, plotting, report helpers) that can be tested outside Shiny.
- `inst/app/www` serves static assets: CSS, a JavaScript message handler, method write-ups as `.rmd`.
- **golem options** switch behaviour at launch, without code changes:

```
run_app(server_link = "https://.../sae4health/")
run_app(version = "DHS-WHO")
run_app(legend_color_reverse = TRUE)
```

Shared state across modules: R6 + reactive values

```
MetaInfo <- R6::R6Class("MetaInfo",
  public = list(
    DHS.country.meta = NULL,
    DHS.survey.meta  = NULL,
    initialize = function() {
      self$DHS.country.meta <- reactiveVal(NULL)
      self$DHS.survey.meta  <- reactiveVal(NULL)
    }))
# also CountryInfo (selection, data)
# and AnalysisInfo (fitted models)
```

- Three R6 objects (`MetaInfo`, `CountryInfo`, `AnalysisInfo`) are created per session and passed to every module.
- Each field is a `reactiveVal`, so any module that reads it re-runs when another module changes it.
- This avoids a web of module return values, but every module can now change shared state. Naming discipline matters.

The three classes hold 39 reactive fields between them. Draw the dependency graph before adding one more.

Making a statistics app feel responsive

Feedback

- A custom JS handler (`controlSpinner`) shows messages such as "Loading IR Recode…" while R works, via `session$sendCustomMessage`.
- The sparsity check runs before any model, so users do not wait for a fit that should not run.
- Every visualization has an **interactive** (leaflet/plotly) and a **static** (ggplot) version. Static is faster and is what downloads and reports use.

Limits to know

- Uploads: `shiny.maxRequestSize` raised to 550 MB, since DHS zips are large.
- INLA fits run inside the user's R session. A long fit blocks that session's other requests.
- Server sessions time out when idle. A user who leaves for lunch loses fitted models, so the app offers model-object downloads.

Report generation: the largest module

- One click produces a multi-page PDF: maps, key statistics, rankings and method notes for the fitted models.
- Pages are assembled with **grid**, **gridExtra** and **patchwork** and written with `grDevices::pdf()`. No LaTeX is needed on the server.
- The report is available in **English, French, Spanish and Portuguese**. Text and labels are looked up per language.
- At 2,894 lines, it is bigger than any visualization module.

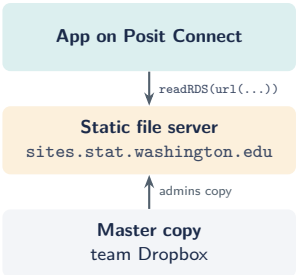
Design choice: avoid R Markdown/LaTeX rendering at run time. It adds a heavy system dependency (TeX) and a failure point on every server and container.

The sae4lmic pipeline, which runs offline, *does* use R Markdown (`report/*.Rmd`) for its country PDFs.

Where the data lives

Microdata cannot ship with the app

- DHS and MICS microdata are **access-restricted**: users register and request each survey.
- So the code repository holds **no recode data**. It holds metadata only: country codes, survey lists, indicator dictionary, DHS API estimates.
- Three ways data gets in:
  1. User **uploads** the DHS zip (local or web).
  2. **Server version**: the app fetches processed recodes the team is permitted to host.
  3. **Preloaded** in the app (Nigeria MICS 2021).



The server version: a plain HTTP file server

```
recode.path <- paste0(golem::get_golem_options()$server_link,
                      '/DHS_survey_dat/', country_code, '/DHS_', svy_year,
                      '/', required_recode[i], '.rds')
recode.data <- readRDS(url(recode.path))
```

- Data is a folder tree of `.rds` files, one per recode (IR, KR, BR, …) and GPS file, per survey.
- Only the recodes the chosen indicator needs are fetched, and already-loaded ones are skipped.
- No database and no API to maintain: just a static web folder.

Testing it locally

Copy part of the folder tree to your machine, then:

```
python3 -m http.server 8000
run_app(server_link =
  "http://localhost:8000/")
```

Preloading in the MICS app

- For Nigeria, each indicator is processed **ahead of time** by **MICSprev** into one file, `data/MICS/<ID>_2021.RData` (54 files), plus a cluster file with GPS locations.
- A generation script reads each `.sav` recode once, loops over every indicator, and prints a per-indicator success table.
- In the app, a country list (`no_input_country`) hides the upload inputs and turns the button into **Load Survey Data**.
- Missing file? The app shows an "unavailable, try another year" message instead of crashing.

Processed indicator files are much smaller than raw recodes, but they still derive from restricted data. Decide where they may live *before* committing them to a public repository.

Load each `.RData` into a private environment and take whatever object it holds. This makes loading robust to how the file's object was named.

Shapefiles: the other big data problem

- Sources: **GADM** (via **geodata**), **geoBoundaries**, **WHO** boundaries, and the DHS survey's own regions.
- Stored as `.rds` or `.rda` **sf** objects so the app does not need to read shapefiles at run time.
- Clusters are assigned to areas by **point-in-polygon** on GPS coordinates.

What goes wrong

- Boundaries change between surveys. Ethiopia split Admin-1 regions between 2019 and 2024, so 2019 results had to be re-run on new boundaries, with the new regions treated as unplanned domains.
- GADM files are large. They are why the MICS repository is over 1 GB.
- Names differ across sources, so a linkage table (e.g. `WHO_shp_linkage`) is needed.

Reproducible environments with renv

renv in one slide

- `renv.lock` records every package, its exact version, and where it came from (CRAN, Posit Package Manager, GitHub, the INLA repo).
- Each project gets its own library. `renv::restore()` rebuilds it from the lockfile on a new machine.
- `renv::snapshot()` writes the current library back to the lockfile. `renv::record("pkg")` updates one entry.
- `.Rprofile` runs `renv/activate.R` so the project library loads automatically.

Package sources in the lockfiles

	DHS	MICS
Repositories	218	242
GitHub	1	6

GitHub packages in the MICS app: **surveyPrev**, **SUMMER**, **MICSprev**, **surveyPrevGithub** and two others.

The lockfile is what makes "it works on my laptop" also true on Posit Connect, in Docker, and on a new team member's laptop.

What actually goes wrong on a fresh `renv::restore()`

Problem	Why	Fix
INLA fails to download	Not on CRAN; very large binary; "testing" track with date versions (26.03.19 = 19 March 2026)	`options(timeout = 600)`; `restore(exclude = "INLA")`, then install INLA from its own repo; `renv::record("INLA")`
sf, terra won't compile	Source builds need GDAL/GEOS/PROJ system libraries	Install **binaries** (`options(pkgType = "binary")`), then `renv::record()` them
Recommended packages "out of sync"	Your R ships newer **Matrix**, **MASS**, **survival**…	Usually safe to keep yours; record them
R version mismatch	Lockfile 4.4.3, laptop 4.5.1	Install the matching R with `rig`, or accept newer binaries
GitHub rate limits	Many GitHub packages restored at once	Set a `GITHUB_PAT`

From onboarding a new developer onto the MICS app, September 2026.

A restore recipe that works

```
# 1. Everything except INLA
options(timeout = 600)
renv::restore(exclude = "INLA")

# 2. Spatial package won't compile? Take the binary, record it
options(pkgType = "binary")
utils::install.packages(c("sf", "terra", "units", "s2", "lwgeom"))
renv::record(c("sf", "terra", "units", "s2", "lwgeom"))

# 3. INLA from its own repository
install.packages("INLA", repos = c(getOption("repos"),
  INLA = "https://inla.r-inla-download.org/R/testing"), dependencies = TRUE)
renv::record("INLA")

# 4. Check, then launch
renv::status(); golem::run_dev()
```

Recording a newer version changes *your* lockfile. Do not commit it unless the team agrees to move everyone forward.

Deployment

Posit Connect at UW Statistics

- The department runs **Posit Connect** at `rsc.stat.washington.edu`, plus a development server, `rsc-dev`.
- Publishing bundles the code and `renv.lock`. Connect restores the packages on the server and runs `app.R`.
- Apps are addressed by an **appId**. Redeploying to the same id keeps the URL (`/sae4health/`, `/app_direct/sae4mics/`).
- Connect handles processes, logs, access and vanity URLs from its dashboard.

```
rsconnect::deployApp(
  appDir  = ".",
  account = "<netID>",
  server  = "rsc.stat.washington.edu",
  appId   = <copied appId>)
```

`rsconnect/` records where each copy was deployed, so the next `deployApp()` updates the same app.

Deploy to **rsc-dev** first. A broken production app is what workshop participants see.

Docker: the same app, anywhere

```
FROM rocker/shiny:4.5.1
RUN apt-get install ... libgdal-dev libgeos-dev \
    libproj-dev libudunits2-dev pandoc ...
COPY renv.lock renv.lock     # cached layer
RUN --mount=type=secret,id=github_token \
  ... renv::restore(exclude = "INLA") \
  && install.packages("INLA", repos = INLA_repo)
COPY --exclude=renv . .
ENV RENV_PATHS_LIBRARY=/srv/shiny-server/renv/library
EXPOSE 3838
CMD ["R","-e","...shiny::runApp(app, port=3838)"]
```

- **rocker/shiny** gives R and Shiny Server; `apt` adds the system libraries **sf** needs.
- Copying `renv.lock` *before* the code means the slow package layer is rebuilt only when the lockfile changes.
- Restore uses **Posit Package Manager Linux binaries**, which avoids compiling old pinned versions.
- A **BuildKit secret** passes a GitHub token without leaving it in the image.

Continuous builds with GitHub Actions

```yaml
on:
  push:
    branches: [ "main" ]
  workflow_dispatch:
  schedule:
    - cron: "0 6 * * 0"
env:
  REGISTRY: ghcr.io
# build -> push -> sign (cosign)
```

- Both app repos build a Docker image on every push to `main`, and again **every Sunday at 06:00 UTC**.
- The weekly rebuild catches upstream breakage (a new INLA, a removed package) before users do.
- Images go to the **GitHub Container Registry** and are signed with **cosign**.
- The build cache (`type=gha`) keeps rebuilds fast.

A container that builds is not an app that works. Add a smoke test that starts it, loads a country, and fits one direct estimate.

Four ways to distribute the same app

Web app	CRAN	GitHub	Docker
Posit Connect; nothing to install; server-hosted data	`install.packages()`; offline use; user uploads data	development version; newest features; `install_github`	one command; identical environment; self-hosting

- The web app is the main channel: most users only need a browser.
- CRAN adds review and a citable version (**sae4health** 1.2.3), but non-CRAN dependencies (INLA) must be installed separately.

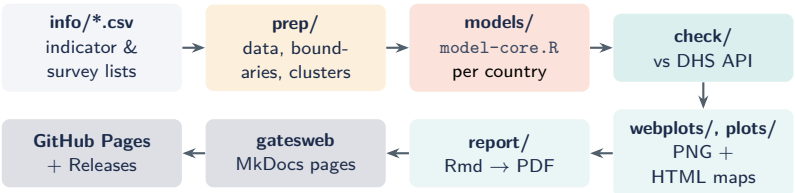
sae4lmic: a results-only website

Why a results-only site?

- Many users want **numbers**, not a modelling tool. Going through model choice one indicator at a time is still a lot of work.
- https://sae4lmic.stat.uw.edu/ publishes **pre-computed, model-checked** estimates.
- 13 African countries, ∼20 indicators in four groups: maternal health, child health, social determinants, health outcomes.
- Each indicator page: interactive maps, ridge plots, change over time; each country: a PDF report and a CSV of all estimates.

The key design decision: no R at view time. The site is **static**: Markdown pages, saved images and saved HTML widgets. It cannot crash under load and costs nothing to host.

The pipeline: from recodes to web pages



- Two hand-maintained CSVs (`infolist.csv`, `surveyslist.csv`) drive every script. Adding an indicator starts there.
- Results are saved with **qs** for fast reading; a tracker table records which core script produced each survey's results.
- Country fixes go in `model-<Country>.R`, never in `model-core.R`. An administrator merges them after review.
- National estimates are checked against the DHS API before anything is published.

The website: MkDocs Material on GitHub Pages

```yaml
site_name: Multi-Indicator Small
  Area Estimation Resource
theme:
  name: material
  features: [navigation.tabs,
    navigation.instant, search.suggest]
nav:
  - Kenya:
    - Overview: .../kenya.md
    - Maternal Health:
      - 4+ ANC Visits: .../RH_ANCN_W_N4P.md
```

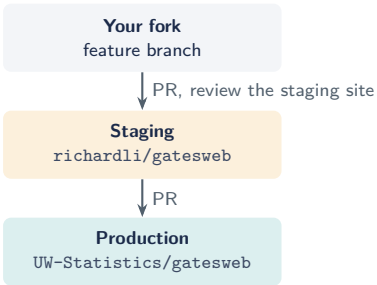
- **MkDocs** turns Markdown into a site; the **Material** theme adds tabs, search and mobile layout.
- 274 Markdown pages and 1,869 PNG figures, all **generated by R scripts**, not written by hand.
- A GitHub Action runs `mkdocs build --strict` on every push to `main` and deploys to GitHub Pages. A custom domain comes from a `CNAME` file.
- `--strict` turns warnings (a broken link) into a failed build, which leaves the **old site** up.

Fitting big files into free hosting

Content	Where it lives	Why
Static PNG maps and ridge plots	Main repo, tracked with **Git LFS**	Keeps the Git history small; needs `git lfs install` on every clone
Interactive leaflet maps (self-contained HTML)	Separate repos (`gatesweb_html1`, `_html2`), each with its own GitHub Pages site; embedded with `<iframe>`	Too many large files for one Pages build; split alphabetically by country
Country PDF reports and CSV estimates	**GitHub Release** assets, linked from pages	Not part of the site build at all; replace with `gh release upload --clobber`
Build machine disk	The Action deletes Android, .NET and other preinstalled tools first	The runner runs out of space otherwise

If LFS is not installed, images are committed as ~130-byte pointer files. The push succeeds and the site shows broken images.

Publishing through staging



- Preview locally first with `mkdocs serve`.
- An **open** PR changes nothing; Pages builds only from `main`.
- The HTML-map repos are pushed directly, without staging.
- Release assets are independent of branches, so download links work on both sites.

Regenerated images keep their file names, so browsers show cached copies. Check a *new* file name, or a private window, before concluding that a deploy failed.

AI in the development workflow

Where we have used AI assistants

Task	What the assistant did
Environment set-up	Walked a new developer through a failing `renv::restore()` (INLA downloads, GDAL, R versions) to a running app
Docker and CI	Rewrote the MICS Dockerfile: binary restore, INLA installed separately, a token passed as a build secret
New MICS indicators	Planned and coded Nigeria-only indicators in **MICSprev**, with a country "gate" so other countries are unaffected
Preloading data	Rewrote the MICS app's data-loading module and wrote the script that generates the 54 preloaded files
SPSS → R translation	A reusable skill that turns MICS SPSS tabulation syntax into one R function per indicator (the "indicator factory")
Website maintenance	Runbooks and release scripts for adding countries, fixing boundaries (Ethiopia), and debugging deploys
Workshop materials	These decks, including taking figures directly from the running apps

Assistant: Claude (Anthropic), used through chat and agentic coding sessions, 2026.

Guardrails that made it work

Write the rules down

- A `CLAUDE.md` file gives the assistant the architecture, the function contract every indicator must follow, and the git workflow.
- Scratch work goes in a folder listed in `.git/info/exclude` (local-only), so it can never be committed.
- **Never commit `.sav` microdata**, anywhere.

Make it auditable

- Translate SPSS **literally**; mark anything ambiguous with a `# FLAG:` comment citing file and line.
- Check every indicator against the **published survey report** before it enters the app.
- Work on feature branches in a fork; humans review every PR.

The assistant does not have access rights to survey data, and should not get them through you. Share syntax, variable lists and error messages, not microdata.

What to watch for

- **Context is everything.** Answers improve sharply once the assistant has read the actual repository rather than guessing at it.
- **Versions drift.** Package names, function arguments and INLA versions change; check suggestions against the lockfile and the docs.
- **Statistics need a statistician.** Domain definitions (denominators, age windows, unplanned domains) are where silent errors live.
- **Two copies of a script** cause most "it didn't change" bugs, with or without AI. Keep one source of truth.
- **Ask for the check, not just the change**: a `stopifnot()`, a summary table, a dry run.
- **Keep a runbook.** Writing the steps down (with the assistant's help) is what lets the next person repeat them.

Lessons and advice

A live example: the app depends on what it loads at start-up

- While preparing these slides, the hosted DHS app showed "`object 'DHS.country.meta' not found`" on the Data Upload page, and the session disconnected on **Load Data from Server**.
- Several helpers find metadata with `get(..., envir = parent.frame())`. If the object is not where the helper expects, the session ends.
- Users see only a greyed-out page.

Defences

- Pass shared objects explicitly, not through `parent.frame()`.
- Wrap data loading in `tryCatch()` with a user-facing message.
- Add an automated test (e.g. **shinytest2**) that loads one preloaded survey after every deploy. The DHS app has one **testthat** file today.
- Watch Connect's logs after each deploy.

Observed on the production app, 7 November 2026.

Advice for anyone building a platform like this

1. **Put the statistics in a package**; keep the app thin.
2. **Build the app as a package** (golem) with modules and testable helpers.
3. **Pin everything** with renv, and rebuild a container weekly.
4. **Separate code from data**, and data access from data hosting.
5. **Stage before production**: rsc-dev, a staging site.
6. **Static where you can**: a results site needs no server.
7. **Test the path users take**, not just the functions.
8. **Write runbooks.** Teams change; the steps must outlive them.

> The hardest problems were rarely statistical. They were installation, boundaries, file hosting, and keeping several copies of the same thing in sync.

Where this could go next

Packages

- A data package of survey information: population fractions, urban/rural fractions, census information, special maps and non-standard designs.

Platforms

- A multi-survey app combining DHS and MICS.
- More result-only sites, linked back to the modelling apps.
- Automated deploy tests and monitoring for the hosted apps.

Resources

- **DHS app:** https://rsc.stat.washington.edu/sae4health/ **Source:** https://github.com/wu-thomas/sae4health
- **MICS app:** https://rsc.stat.washington.edu/app_direct/sae4mics/ **Source:** https://github.com/LucyXuu/SurveyPrevRShiny
- **sae4lmic site:** https://sae4lmic.stat.uw.edu/ **Site source:** https://github.com/UW-Statistics/gatesweb
- **sae4lmic pipeline:** https://github.com/richardli/SAE4LMIC
- **Documentation:** https://sae4health.stat.uw.edu/
- **Paper:** Wu, Dong, Xu, Li & Wakefield (2025). *sae4health: An R Shiny application for small area estimation in LMICs.* arXiv:2505.01467
- **Tools:** golem (https://thinkr-open.github.io/golem/), renv (https://rstudio.github.io/renv/), MkDocs Material (https://squidfunk.github.io/mkdocs-material/)