

Offloading R Simulations to a Home Server

Ronald G. Thomas

2026-10-04



A simulation that needs eight hours should not need a laptop to stay open for eight hours.

Introduction

Most of our simulation studies are small enough to run on a laptop and large enough to be inconvenient there. A factorial grid of a few hundred cells, each with a few hundred replicates, takes an afternoon or a night. During that time the laptop has to stay awake, plugged in, and on the network, and it is not much use for anything else. Closing the lid ends the run.

The fix we settled on is unglamorous: an always-on machine at home that accepts simulation jobs, runs them in exactly the environment the project specifies, and hands the results back. This post describes how it is built. It is written for a small research group or a single analyst with a spare computer, not for a cluster, and it reuses the container images that a zzcollab compendium already defines.

Motivations

- Long simulations tie up a workstation and fail when it sleeps.
- A result should be traceable to the exact code and environment that produced it, wherever it ran.
- The project's container image already pins R, system libraries and packages; running anywhere else would discard that guarantee.
- An always-on home server is cheap, and one may already exist for backups.
- Checking on a job should not require logging in to the server.

Objectives

1. Submit a simulation from a workstation with one command and close the laptop.
2. Run it on the server in the project's own container image, from an exact commit, with the project's data, under CPU and memory limits.
3. Receive a phone notification when the job ends.
4. Fetch results, with a provenance record, into a separate folder that never overwrites working files.



What This Does Not Cover

This is not a batch scheduler. There is one queue, jobs run one at a time, and there is no fair-share accounting, no GPU support and no multi-node parallelism. A university cluster with Slurm is the right tool when those matter. It is also not a cloud setup: nothing here leaves the house except the push notification, and that carries no project details.

Prerequisites

- **An always-on server on the home network.** Ours started as a repurposed 2016 laptop running Linux Mint 22, already serving as the backup host (see the security series post on the home server). A small desktop with wired Ethernet is a better choice, for reasons the ‘Things to Watch Out For’ section makes concrete.
- **A stable address for it.** We use Tailscale, so scripts never depend on the router’s address assignments.
- **SSH key access** from each workstation to the server.
- **zzcollab compendia**, each with a `Dockerfile` and `renv.lock`, so every project defines its own image.
- **Podman** on the server (rootless), and Docker or Podman on the workstations.

How It Fits Together

Piece	Where	Job
<code>simrun</code>	workstation	Submit, watch, fetch; talks to the server over SSH
<code>simrun-server</code>	server	Receives commands; manages the queue and run folders
<code>task-spooler</code>	server	A one-at-a-time job queue
<code>simjob</code>	server	Runs one job from start to finish
Podman, rootless	server	Runs the project’s image without a root daemon
BuildKit, rootless	server	Builds images from zzcollab Dockerfiles
ntfy	phone	Push notification when a job ends

The workstation never needs a shell on the server for normal use. Every `simrun` command is one SSH call to `simrun-server`, which prints plain text the client parses.

Setting Up the Server

Rootless containers and a queue

The packages are all in the distribution’s repositories:

```
sudo apt-get install -y podman uidmap passt slirp4netns \
  fuse-overlayfs task-spooler
sudo loginctl enable-linger "$USER" # user services survive logout
```

Rootless Podman needs subordinate user and group ID ranges for the account (`/etc/subuid`, `/etc/subgid`); most distributions create them with the user. Running rootless matters on a machine that also holds backups: there is no root daemon and no `docker` group, so a job that misbehaves can reach the account it runs as and nothing else.

Two small configuration details saved time later. First, zzzcollab Dockerfiles name base images Docker-style (`rocker/verse`), and Podman refuses unqualified names unless told where to look:

```
# ~/.config/containers/registries.conf
unqualified-search-registries = ["docker.io"]
short-name-mode = "permissive"
```

Second, the scratch area for runs (`~/compute`) should be excluded from the server's own backup, since everything in it can be regenerated from a commit and the data store, and results are fetched back anyway.

Resource limits

On a server that also runs backups and file sharing, a simulation must not take every core. With `cgroups v2`, `systemd` delegates the `cpu` and `memory` controllers to user sessions on current distributions, which lets rootless Podman enforce limits per container:

```
podman run --cpus=3 --memory=12g --cpu-shares=512 ...
```

We cap jobs at three of four threads and 12 of 16 GB, leaving room for the nightly backup and the file-sharing service.

A builder that understands zzzcollab Dockerfiles

Recent zzzcollab Dockerfiles use a heredoc in a `RUN` instruction to warm up the TeX installation. Docker's builder (BuildKit) supports that syntax; the builder inside Podman 4.9, the version Ubuntu 24.04 and Mint 22 ship, does not, and stops with 'Unknown instruction'. Rather than add a third-party repository for a newer Podman, we run BuildKit itself as a rootless container and copy its client binary out of the same image, pinned by digest:

```
IMG=docker.io/moby/buildkit@sha256:<digest>
ctr=$(podman create "$IMG")
podman cp "$ctr:/usr/bin/buildctl" ~/.local/bin/buildctl
podman rm "$ctr"
podman run -d --name buildkitd --privileged --cpus=3 --memory=12g \
-v buildkit-state:/var/lib/buildkit "$IMG"
```

Under rootless Podman, `--privileged` grants privileges inside the user namespace only; BuildKit needs them to create its own build sandboxes. Builds stream straight into Podman's image store:

```
buildctl --addr podman-container://buildkitd build \
  --frontend dockerfile.v0 \
  --local context=. --local dockerfile=. \
  --output type=docker,name=localhost/myproject:abc123 | podman load
```

A notification channel

ntfy is a free push service with iOS and Android apps. A topic needs no account; its name is the only secret, so it should be long and random. The server keeps the topic in a file only its owner can read, and a notification is one `curl` call. Messages carry only the job number, status and duration, never a project name, because they pass through a public server.

The Life of a Job

Submitting

```
cd ~/prj/res/my-simulation-study
simrun submit -- Rscript analysis/scripts/01-run-factorial.R
# job 12 queued on simhost (res/my-simulation-study @ 4e1c2a9, 3 cores, 12 GB)
```

`simrun` refuses to submit with uncommitted changes, because a job always runs a commit, never the working folder; a job that silently ignored local edits would be worse than one that refused. An explicit `--commit REF` runs a named commit regardless of local edits.

The client pushes the commit to a bare repository on the server, then asks `simrun-server` to queue the job. The server writes a small job file (project, commit, command, limits) and hands `simjob <id>` to `task-spooler`.

The environment fingerprint

Each image is named by a hash of the commit's `Dockerfile` and `renv.lock`. The same hash is what `zzcollab` already writes into a label on the images it builds (`zzcollab.dockerfile.hash`), so the workstation and the server agree on whether an image matches a commit without comparing anything else.

If the server has no image for that fingerprint, there are two ways to get one:

- **Build on the server (default).** `simjob` builds with BuildKit before running the job; `simrun status` shows the job as `building`. This needs nothing from the workstation. For a rocker/verse-based compendium with 214 R packages, the first build took 15.5 minutes on our two-core server, against 25 minutes on an Apple silicon workstation, which has to emulate `amd64`. Later jobs from the same `Dockerfile` and `renv.lock` reuse the image.
- **Ship from the workstation (`--ship`).** `simrun` checks that the local image's label matches the commit, then streams it across:

```
docker save myproject | zstd -T0 -3 -c | \
ssh simhost 'simrun-server image-load localhost/myproject:abc123 zstd'
```

Shipping runs the job in byte-for-byte the environment you tested with; building on the server reproduces it from the same pinned inputs, which is close but not identical, because Ubuntu system packages inside the image are not pinned. The provenance record stores the image digest either way. Images compress about two to one, which matters when the network is the bottleneck.

Data

Research data does not live in git. The server keeps a permanent data store with one folder per project, mirroring the project's own layout, and the workstation updates it deliberately:

```
simrun data push analysis/data
```

When a job starts, `simjob` unpacks the commit with `git archive` and then copies the project's data over it with `rsync --ignore-existing`, so a tracked file always wins over a data file of the same name, and the data store itself is never written by a job.

Running

```
podman run --rm --name simrun-12 \
--userns=keep-id:uid=$IMG_UID,gid=$IMG_GID \
--cpus=3 --memory=12g --cpu-shares=512 \
-v "$RUN/work:/home/analyst/project" \
-w /home/analyst/project \
-e SIM_CORES=3 \
"$IMAGE" Rscript analysis/scripts/01-run-factorial.R
```

`--userns=keep-id` with the image user's IDs maps the server account onto the container's non-root user, so files the job writes into the bind mount stay owned by the server account rather than by an anonymous subordinate ID. `SIM_CORES` tells the R code how many cores it has; `parallel::detectCores()` inside a container still reports the host's count, which would oversubscribe the cap.

Results and provenance

Before the run, `simjob` records every file in the working tree with its size and modification time; afterwards it records them again. Every new or changed file is a result and is copied to `results/`. Nothing has to be declared in advance. A `provenance.yml` records the job, project, commit, command, image name and digest, host, cores, memory, submission, start and finish times, duration, exit status and data source:

```
job: 12
commit: 4e1c2a9d0b7f3e6a1c5d8b2f9e0a4c7d3b6e1f80
command: Rscript analysis/scripts/01-run-factorial.R
image_digest: sha256:9b1f...
cores: 3
memory_gb: 12
duration_seconds: 11460
exit_status: 0
```

(An illustrative record; the values are invented.)

`simrun fetch 12` copies the results, the log and the provenance file into an arrival area inside the project, `analysis/remote_runs/job-12-4e1c2a9/`, and changes nothing else. A second step, `simrun apply 12`, copies the results to the paths a local run would have written, typically `analysis/data/derived_data/`, so a result's location does not depend on which machine produced it. Apply refuses, and copies nothing, if any result would overwrite a file that differs from it: the job ran the commit that was submitted, and the working folder may have moved on while it ran. The provenance stays behind in `analysis/remote_runs/`, so an applied file can always be traced to the commit and image that produced it.

Watching and notification

```
simrun status
# job 12   running   my-simulation-study 4e1c2a9 2h14m
simrun watch 12   # redraws every five seconds until the job ends
simrun log 12 -f   # live output
```

The status line gives the state, the elapsed time and, once finished, the exit status. How far a script has got is whatever the script itself prints, read with `simrun log -f`; a `message()` per completed block of work is enough. While the server is building a project's image, the status line instead shows the current build step and how long it has been running, for example 'step 9/11: restoring R packages (renv), 2m20s in this step'. A percentage would mislead here: downloading the base image and restoring the packages take nearly all of a build.

When the job ends, `simjob` posts to the `ntfy` topic: 'simrun job 12 done: Finished OK in 3h11m', or 'failed: Exit 1 after 0h04m'.



Why There Is No Progress Bar

An earlier version drew a progress bar with an estimated finish time. Nothing outside an R process can know how far through a loop it is, so the script had to report it: either two `simrun`-specific helper calls around the main loop, or the standard `progressr` package, picked up by an R startup file that `simjob` injected into every job. Both worked in tests. Both also had costs out of proportion to a one-queue home server: the first put `simrun`-specific lines into research code, and the second required `progressr` in every project's `renv.lock`, worked only with `future`-based parallel loops (not `parallel::mclapply`), and failed silently when either condition was missed. The bar was removed. A job's state, elapsed time, log and end-of-job notification answer the practical question, which is whether it is still running.

Daily Workflow

Task	Command
Update the project's data on the server	<code>simrun data push analysis/data</code>
Submit a job	<code>simrun submit -- Rscript path/to/script.R</code>
Submit and send the local image	<code>simrun submit --ship -- ...</code>
See all jobs	<code>simrun status</code>
Watch one job	<code>simrun watch 12</code>
Follow its output	<code>simrun log 12 -f</code>
Stop it	<code>simrun cancel 12</code>
Bring results back	<code>simrun fetch 12</code>

Task	Command
Put them where a local run would	<code>simrun apply 12</code>
Run the same job again	<code>simrun resubmit 12</code>
Free its space on the server	<code>simrun clean 12</code>
Show settings and where each came from	<code>simrun config</code>
Use another host for one command	<code>simrun --host ALIAS status</code> (give the same --host to every command about that job)

Defaults (server alias, cores, memory, project root) can live in a small `key = value` file, `~/.config/simrun/config`, which the client parses rather than executes. Environment variables override it, and command-line options override both, so a one-off change never requires editing the file.

More Than One Host

One server is enough until a second, faster machine turns up. A second host is set up exactly like the first (a script collects the steps, and with a flag it also stops a laptop from sleeping with its lid closed), and a client chooses between hosts with `--host`, an SSH alias, before the command:

```
simrun submit -- Rscript analysis/scripts/simulation.R          # default host
simrun --host fast submit -- Rscript analysis/scripts/simulation.R
simrun --host fast status
simrun --host fast fetch 3
simrun --host fast apply 3
```

The hosts are independent. Each has its own queue, its own images and its own job numbers, so the same `--host` goes with every command about a job. Data are per host too (`simrun --host fast data push`), and a project's first job on a new host builds its image there. Because job numbers restart at 1 on every host, a fetched folder carries the alias unless the host is the configured default: `job-3-<sha>` for the default, `job-3-<sha>-fast` for any other.

Default cores and memory can differ per host in the config file:

```
host = fast
cores = 3
mem = 12
cores@fast = 8
mem@fast = 24
cores@small = 3
mem@small = 12
```

The per-host sizes are not a luxury. A cap larger than the machine fails: a job asking for 8 CPUs on a 4-thread host does not start.

Choosing between hosts

Specifications did not settle which host to prefer, so we measured: a fixed amount of single-threaded numerical work (1,200 regression fits on generated data), timed inside R, two repetitions per setting, BLAS pinned to one thread so that the core count alone sets the parallelism. The two hosts were a 2016 laptop (2 cores, 4 threads) and a 2023 ultrabook (10 cores, 12 threads). Seconds for the whole run:

Cores	2016 laptop	2023 ultrabook
1	139.7	68.9
3	78.5	38.3
4	69.3	n/a
8	n/a	22.0
12	n/a	18.3

- Per core the newer machine is 2.0 times faster, and 3.8 times faster at each machine's full core count.
- Scaling is far from linear on both: 1.8 times at 3 cores on the newer machine, 3.1 at 8 and 3.8 at 12; 2.0 at 4 on the older, which has two physical cores. Asking for more cores than pay for themselves wastes allocation, so the newer host's default is 8, not 12.
- Short runs flatter the newer machine. Under a five-minute all-core load its clock fell from about 2.9 to about 2.1 GHz within half a minute and stayed there, at a moderate 64 to 69 C, so its sustained advantage was 2.7 times, not 3.8. That points to a power limit rather than heat (inferred; power was not measured). The older machine held its clock but ran at 88 to 96 C throughout.
- Every run, on both machines and at every core count, produced the same checksum of the results.

Letting Other Machines Submit Jobs

The first client has a full login on the servers. Other workstations do not need one. A client's key can be pinned in the server's `authorized_keys` to a forced command:

```
restrict,command="/home/USER/.local/bin/simrun-gate" ssh-ed25519 AAAA... simrun-laptop2
```

`simrun-gate` is a short Python script. It splits the original command into words without any shell, and allows exactly two things: running `simrun-server` with arguments, which is what every `simrun` command does, and `git-receive-pack` on a bare repository directly under `~/compute/repos`, which is how a commit arrives. Everything else is refused. On both servers, from two workstations, we checked that `id`, `ls`, `sh -c`, a file download with `scp`, a shell request and a remote port forward are all refused, that `simrun status` works, and that a complete job (push through the gate, run, fetch, apply) gives the same checksum as the first workstation's.

A restricted key is narrower than a login key, not harmless: it can still submit jobs, which run code in a rootless container.

What the restriction exposed

Once a key may reach only `simrun-server`, `simrun-server` is the security boundary, and it had been written for a caller we trusted. Three weaknesses that had not mattered before surfaced while testing:

1. **The submitted command was evaluated by the host.** The server stored the command string, and the job runner later turned it into an argument list with `eval`, on the host and outside the container. The real client escapes the string, but a hostile client could send `$(...)`. Fix: the server splits the string with Python's `shlex`, re-quotes each word, and stores only that.
2. **The data upload could write anywhere in the account.** It was an archive unpacked into the data store, and a symlink member could redirect a later member to any path the account can write. Fix: the server stages the archive and refuses links and `..` or absolute paths before unpacking.
3. **Inputs were not validated.** A commit written like an option reached `git archive`, and `..` in a project name reached the filesystem. Fix: each input is checked against a strict pattern before use.

We checked all of this with hostile inputs in a throwaway root: 21 cases, including the `eval` one, where the stored command becomes five inert words and nothing runs. Ordinary jobs then ran on both hosts unchanged. The lesson generalizes: once a service is reachable through a restricted credential, its own input handling is the boundary, and 'only my client talks to it' stops being an argument.

Reaching the hosts from outside the house

A host's name on the home network (a multicast `.local` name) follows its address but depends on multicast DNS, which failed on the first lookup after a quiet period on both clients we tested, then worked. Each client therefore also has an alias that uses the host's overlay-network address, which works from anywhere the overlay is switched on, with `HostKeyAlias` set to the `.local` name so that one verified host key serves both. The overlay route was first checked only by pointing the alias at the LAN addresses. It has since carried a real job from the primary workstation (the one on the dedicated account, described above, ran through it while at home), and a status call from the other two workstations. It has not been tested from outside the house.

Reference

Job states and exit statuses

A job moves through `queued`, `building` (only when the server first builds an image for a given `Dockerfile` and `renv.lock`) and `running`, and ends as `done` (exit status 0), `failed` (any other status) or `cancelled` (removed from the queue before it started).

Exit status	Meaning
0	The command succeeded
10	Unpacking the commit on the server failed
11	Copying the project's data into the run failed
20	The project image could not be built or loaded
anything else	The command's own status, passed through: an R error is usually 1, a job killed at the memory limit 137, a job stopped on request 143

Keeping the runner's own failures in a small, distinct range means a failed job can be triaged from its status alone: 10 to 20 is the infrastructure, anything else is the simulation.

What the command sees inside the container

Variable	Value
<code>SIM_CORES</code>	Cores the job may use; use it for parallel workers
<code>SIMRUN_JOB</code>	The job number
<code>SIMRUN_COMMIT</code>	The full commit hash. The run is unpacked with <code>git archive</code> and has no <code>.git</code> folder, so a script that calls <code>git rev-parse</code> gets nothing and should read this instead
<code>ZZCOLLAB_AUTO_SNAPSHOT</code>	<code>false</code> , so <code>zzcollab</code> 's exit-time <code>renv</code> snapshot does not rewrite <code>renv.lock</code> and report it as a result

The last row came from the first real test: the compendium's `.Rprofile` snapshotted `renv.lock` when R exited, and the rewritten lock file appeared among the job's results.

Moving to a New Server

The workstation addresses the server through an SSH alias, named by the `host` setting in the config file (default `simhost`). Replacing the server means pointing that setting, or the alias, at the new machine; nothing in `simrun` or in any project changes. We plan to replace the 2016 laptop with a small Apple silicon desktop, which raises one real issue: `zzcollab` images built on `rocker/verse` exist only for `amd64`, while `rocker/r-ver` is published for both `amd64` and `arm64`. On an Apple silicon server those images run under emulation, as they already do on Apple silicon workstations. Moving the `zzcollab` base to a multi-architecture image (`r-ver` plus the TeX and Quarto layers that `verse` adds) would make them native everywhere. On macOS, Podman runs inside a small Linux virtual machine whose size becomes the resource limit, and Podman 5 builds heredoc Dockerfiles itself, so the BuildKit workaround is only needed on the current server.

Things to Watch Out For

1. **Podman 4.9 cannot build heredoc Dockerfiles.** Symptom: ‘Unknown instruction: SET’ at the first RUN << step. Fix: build with BuildKit (above), or ship the image with `--ship`.
2. **Unqualified image names fail under Podman.** Symptom: ‘short-name did not resolve to an alias’. Fix: set `unqualified-search-registries` in `registries.conf`.
3. **Wi-Fi makes a poor server link.** Our server’s built-in adapter sat on a weak 2.4 GHz link, and measured SSH throughput was 2 to 3.5 MB/s whether over Tailscale or directly on the LAN. Shipping a 3.3 GB image took 21 minutes. A wired connection fixes this and helps backups too.
4. **detectCores() lies inside a container.** It reports the host’s cores, not the cap. Use `SIM_CORES`.
5. **Empty fields vanish in tab-separated shell parsing.** Tab counts as whitespace for IFS, so `read` collapses consecutive tabs and shifts every later field. Write a placeholder such as `-` for empty fields.
6. **Files written as the container user.** Without `--userns=keep-id`, output in the bind mount belongs to a subordinate ID the server account cannot delete. Map the account onto the image’s user.
7. **A laptop as a server runs hot, or slows down.** We measured both behaviors on two laptops under five minutes of full load: the older one held its clock but sat at 88 to 96 C, and the newer one dropped its clock by about a quarter within half a minute at a moderate 64 to 69 C (a power limit, we infer). Watch temperature and clock speed on the first long job, and expect sustained throughput below the short benchmark.
8. **Job numbers restart on every host.** Job 3 on one host is not job 3 on another. Give every command about a job the same `--host`, and keep the host in the fetched folder’s name (the client does this for any host other than the default).
9. **A cap larger than the machine does not start.** `--cpus=8` on a 4-thread host is refused. Set per-host defaults (`cores@ALIAS`) instead of one global number.
10. **Jobs share an account with everything else on the host.** Rootless containers confine a job to the server account, not away from it. On the home server that account also owned the backup repositories, so a hostile or runaway job could have deleted them (append-only protection is enforced by the backup keys, not by file permissions). The fix was a dedicated unprivileged account for jobs: it has its own home, queue, container storage and data store, no `sudo`, and no ownership of anything else. The cost is that container images are per account, so the first job after the move rebuilds its image (about six minutes for the demonstration project). Where a separate account is not practical, prefer a host that holds nothing else.

What Did We Learn?

Lessons Learned

- The container image a compendium already defines is the right unit of reproducibility for remote runs too; inventing a second environment for the server would have undone it.
- Refusing to run uncommitted code is a feature. It makes every result traceable and costs one commit.

- Treating ‘whatever changed in the working tree’ as the result set removes a whole class of configuration: no list of output paths to maintain.
- The network, not the server, was the slowest part of the system.
- Measure hosts before preferring one. The newer machine was twice as fast per core, but scaling was far from linear on both and its sustained rate was below its burst rate, so the best setting (8 of 12 cores) was not the obvious one.
- A restricted credential turns the service behind it into the security boundary. Restricting client keys exposed three weaknesses in a server we had written for a trusted caller; hostile-input tests, not review, found them.



Limitations

- One queue, one job at a time, on a small machine: throughput is modest by design.
- A job interrupted by a server restart is marked failed and must be resubmitted; there is no checkpointing.
- Building on the server reproduces an image from pinned inputs but not byte for byte; use `--ship` when that difference matters.
- Notifications pass through a public service and are therefore kept free of project details.
- A restricted client key can still submit jobs, so it limits what a stolen key can reach (no shell, no other files) but not what it can run inside a container.
- Hosts are independent: nothing balances load or moves a queued job from a busy host to an idle one. The user chooses.

Opportunities for Improvement

- A local container registry on the server would send only missing layers; most compendia share the same large base image.
- A multi-architecture zzcollab base image would make every project native on Apple silicon.
- A systemd or Quadlet unit for the BuildKit container would make it start with the server rather than on first use.
- A small status page, readable from a phone, could complement the notification.

Wrapping Up

The whole system is four small scripts (client, server, runner and key gate), two containers and a queue per host. It turns ‘leave the laptop open overnight’ into a commit, a command and a notification, and it keeps the reproducibility guarantees a zzcollab compendium already makes.

Takeaways:

- Run remote jobs in the project’s own image, from an exact commit.
- Name images by a hash of `Dockerfile` and `renv.lock`, and let the server build or receive them on first use.
- Keep research code free of runner-specific lines; a log and a notification are enough on a single-queue server.
- Keep data in a server-side store that jobs read but never write.
- Put the server on a wire.
- Measure a second host before making it the default, and set cores and memory per host.
- Restrict client keys to the one service, and test that service with hostile input: it is now the boundary.

See Also

- [A Home Server for a Five-Machine Research Fleet](#): the server this runs on, its firewall and addressing
- [Research Backup Architecture: Borg, Cloud Copies, and Time Machine](#): why the job scratch area is excluded from backups
- [Security Foundations for a Multi-Machine Research Fleet](#): one key per relationship, which the restricted client keys extend
- [A tiered CI strategy for zzcollab research compendia](#)
- Podman rootless tutorial: https://github.com/containers/podman/blob/main/docs/tutorials/rootless_tutorial.md
- BuildKit: <https://github.com/moby/buildkit>
- task-spooler: <https://viric.name/soft/ts/>
- ntfy: <https://ntfy.sh>

Reproducibility

Tested configuration:

Component	Version
Hosts	two machines, a 2016 laptop and a 2023 ultrabook
Host OS	Linux Mint 22.3 (Ubuntu 24.04 base) on both
Podman	4.9.3, rootless, cgroups v2
BuildKit	v0.33.1 (pinned by digest)
task-spooler	1.0.1
Key gate and input checks	Python 3.12
Workstations	three Macs, macOS 26 (Docker Desktop on the first)
Last verified	2026-10-12

Verified: an end-to-end job with a shipped image (run under limits, results and provenance fetched, push notification) on 2026-10-03; on 2026-10-04, a server-side BuildKit build of a full **rocker/verse** compendium (15.5 minutes), a smoke run in that image, and a parallel 96-cell run (71 seconds, exit 0); on 2026-10-05, after the progress bar was removed, a fresh build and parallel smoke run of a small compendium (2.5 minutes in all, exit 0) with the build-step line observed during the build. On 2026-10-11, the second host: an image built there in about 3 minutes and a result byte-identical to the first host's and to a workstation's from separately built images; the benchmark in the text; the restricted-key refusals (`id`, `ls`, `sh -c`, `scp` download, a shell request and a remote port forward, on both hosts, from two workstations; joe's full set was completed on 2026-10-12); the 21 hostile-input checks of the server; and a full job from each of two other workstations through restricted keys. After the jobs moved to the dedicated account (2026-10-15), one job was built and run there and its result file was byte-identical to two earlier runs, and the status call through the restricted key was repeated from both other workstations. Not verified: that the dedicated account is unable to read or delete the backup repositories (file modes and group membership were inspected; no attempt to read them as that account was made); the overlay-network (Tailscale) route from outside the house, and a job (not just a status call) through it from the other two workstations; and the explanation of the newer host's slowdown as a power limit, which is inferred.

AI involvement. The scripts and this draft were written with an AI coding assistant (Claude Code) in an interactive session. Design decisions (rootless containers on a backup host, data kept on the server, results in a separate folder, phone notification, dropping the progress bar, building versus shipping images) were the author's, as was the review of every claim against the running system.

R version 4.6.1 (2026-06-24)

Platform: aarch64-apple-darwin25.4.0

Running under: macOS Tahoe 26.6.2

Matrix products: default

BLAS: /opt/homebrew/Cellar/openblas/0.3.34/lib/libopenblas-p0.3.34.dylib

LAPACK: /opt/homebrew/Cellar/r/4.6.1/lib/R/lib/libRlapack.dylib; LAPACK version 3.12.1

```
locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats      graphics  grDevices  utils      datasets  methods   base

loaded via a namespace (and not attached):
[1] compiler_4.6.1 fastmap_1.2.0 cli_3.6.6      tools_4.6.1
[5] htmltools_0.5.9 parallel_4.6.1 otl_0.2.0      yaml_2.3.12
[9] rmarkdown_2.31 knitr_1.51      jsonlite_2.0.0 xfun_0.58
[13] digest_0.6.39  rlang_1.3.0     evaluate_1.0.5
```

Rendered on 2026-10-15 at 12:31 PDT. Source: ~/prj/ryyblog/posts/zc-offload-simulations-home-server/analysis/report/index.qmd

Let's Connect

Have questions, suggestions, or spot an error? Let me know.

- **GitHub:** [rgt47](#)
 - **LinkedIn:** [Ronald Glenn Thomas](#)
 - **Email:** rgthomas47@gmail.com
-

Related posts in this cluster

This post is part of the *ZZCOLLAB Reproducible Compendia* series. Recommended reading order:

1. Post 01: [Reproducible Blog Posts with ZZCOLLAB](#)
2. Post 02: [Constructing a reproducible blog post using zccollab tools](#)
3. Post 03: [From Markdown to Blog Post: A ZZCOLLAB workflow](#)
4. Post 04: [Sharing R Code via Docker: R Markdown Reports](#)
5. Post 05: [A 55-Item Initiation Checklist for zccollab Data Analyses](#)
6. Post 06: [Seven Required Elements for a zcc Manuscript report.Rmd](#)
7. Post 07: [A tiered CI strategy for zccollab research compendia](#)
8. Post 08: [GitHub Actions workflows for zccollab research compendia](#)
9. **Post 09: Offloading R Simulations to a Home Server** (this post)