

Why demolab

ar019 · 11 July 2026

A fair question about any new tool is why the old ones will not do, and here the old ones are genuinely good. Notebooks are everywhere; Quarto and knitr solved number interpolation years ago; a Makefile and a static site generator can wire almost anything to almost anything. So the question deserves a straight answer, not a pitch. demolab exists to make one narrow guarantee: a published result cannot quietly drift from the code that produced it. Everything below is that guarantee measured against the alternatives, including the cases where the alternatives win.

What notebooks do to a publishable result

Notebooks are excellent at the job they were built for: poking at data, trying something, seeing the plot the moment it exists. The trouble starts when a notebook becomes the **record** of a result, because three of its habits work against a record.

1. **Hidden state.** Cells run in any order, and the kernel’s state is the history of executions, not the file. A notebook that reads cleanly top to bottom can still be unreproducible because cell seven ran before cell three ever did. “Restart and run all” fixes this, but it is a discipline, not a property: nothing enforces it, and the stale outputs baked into the file look identical to fresh ones.
2. **Unseeded runs.** Nothing asks for a seed. Re-run the notebook and the number changes, and often nobody notices, because the old output is still sitting in the `.ipynb` looking authoritative.
3. **Retyped numbers.** The markdown cell says the firing rate is 90 Hz because someone read that off an output cell and typed it. The code changes; the sentence does not. Prose and computation now disagree, silently, on the same page.

None of this is an argument against notebooks as scratch paper; they are unbeatable scratch paper. It is an argument against them as the thing you publish. demolab’s answer is structural rather than disciplinary: an experiment is a plain script that runs top to bottom or not at all, its outputs land in a committed record, and the writeup reads its numbers from that record (`json("/artifacts/data/<id>/numbers.json")`), so there is no retyping step at which drift can enter. So much science starts in a notebook that demolab ships a runbook for exactly this conversion: say **FROM-JUPYTER** to an agent and it turns a notebook into a seeded, linear experiment with the numbers wired through.

What literate documents get right, and where demolab differs

Quarto, knitr, and org-mode made the correct diagnosis long ago: if a number appears in prose, it should be computed into place, not typed. A code chunk runs, its value lands in the sentence, and the retyping failure disappears. demolab keeps that idea whole; on this point it is a descendant, not a rival.

The difference is coupling. In a literate document, the document **is** the computation: rendering it re-runs the chunks, and when a chunk takes a week you reach for the cache (Quarto’s `freeze`, knitr’s `cache=TRUE`) and start managing invalidation by hand. Caching is a patch on a coupled model. demolab decouples the two acts instead. A run is one event: the experiment executes, and drops its complete record into `artifacts/data/<id>/`, the rendered figures, a `numbers.json` of headline metrics, and a `run.sh` that reproduces the run. The writeup is a separate file that only **reads** that record. Rebuilding the site never re-runs the science; CI compiles pages from

the committed record and executes no experiments at all. Fixing a typo and repeating a week of computation are different acts, and the system knows it.

Three more differences follow from the same split.

1. **Provenance is stamped, not remembered.** Every run records the git commit it came from, a dirty flag if the working tree had uncommitted changes, and a UTC timestamp; the stamp flows into `numbers.json` and renders as a footer on the published page. A result that outlived its code says so.
2. **One pass, three targets.** A single Typst compile emits the website (with real, selectable math), a PDF per entry, and a bound book, all reading the same committed numbers. There is no separate PDF pipeline to fall out of step with the web.
3. **The operator is an agent, not a build system.** You do not maintain the glue; you talk. Runbooks like **LINT**, **DOCTOR**, and **RED-TEAM** are procedures a coding agent follows step by step, and underneath them is an ordinary CLI (`demolab build`, `demolab dev`, plus `uv run python experiments/expNNN.py` to run one) you can drive by hand.

Which answers the Makefile option too. A Makefile plus a static site generator can wire run to figures to pages, and for a build graph it is the right tool. But the drift guarantee is a **document-level** rule: no number on the page is typed by hand. A Makefile sees files and timestamps; it cannot see inside a sentence. You would end up maintaining the web tooling, the PDF pipeline, and the provenance stamping yourself, and the one thing you built it all for would still rest on authors' good behaviour.

When demolab is the wrong tool

An honest positioning piece has to name the cases where you should not use it.

- **Quick, throwaway exploration.** If you are poking at a dataset to see whether there is anything in it, open a notebook. `demolab`'s own rules warn against manufacturing ceremony for one-off code, and the same judgment applies to the framework itself: the contract earns its keep only when a result is headed for the record.
- **Teams wedded to notebook infrastructure.** If your group runs on JupyterHub, schedules work through papermill, and reviews `.ipynb` diffs, the switching cost is real and the drift you would prevent may not repay it. `demolab` assumes you are willing to write experiments as scripts; if the team is not, the tool will be fought rather than used.
- **Documents that do not publish computed results.** The guarantee protects numbers that a run produced. A blog, a methods overview, a documentation site with no computation behind it has nothing to protect, and a plain static site generator is the simpler tool.
- **Interactive published pages.** The site `demolab` emits is deliberately static: figures, numbers, math, provenance. Exploration lives in a local playground, not on the page. If your result is an interactive widget, publish it with something built for that.

The summary is short. Use a notebook to find the result; use `demolab` to publish it. If the number on the page came from a run, `demolab` makes it structurally impossible for that number to lie about which run; if it did not come from a run, you likely do not need `demolab` at all.