

The contract

ar016 · 7 July 2026

RULES is the single source of truth for how a demolab repo is put together. It covers two things at once: the principles a repo follows, and the contract that lets an experiment call a tool and turn its output into a published page. This is the plain-English tour. If you want the coding agent to walk you through it live, just type **RULES** in capitals and it will.

The toolchain

demolab publishes with a small, fixed stack. Python runs through `uv`, you never call `python` directly, you say `uv run python ...`, and `uv sync` after pulling. Publishing is done entirely with the `typst` CLI: it compiles the whole repository into a website and a set of PDFs in one pass, no Node or bundler involved. Common commands are wrapped by the `demolab` CLI, so reach for `demolab build` or `demolab dev` rather than the raw tools. Python is the default, not a hard requirement: the contract below is file-based and language-neutral, so the tool layer could be MATLAB, R, or Julia instead.

The engine/content firewall

A demolab repo has a clear line between framework and your work. The engine, the Typst build, the runbooks, the guides (this file included), and the scaffold, is a black box: it lives inside the installed `demolab-cli` package (plus the machine-managed `.demolab/` staging directory at the lab root), you never edit it, and updating it is an ordinary dependency bump. Everything in your tree is yours: root files like `demolab.yaml` are your overrides, and everything under `tools/`, `experiments/`, `writings/`, and `artifacts/` is 100% yours, freely deletable and replaceable. An update can't overwrite any of it, because the framework never lives in your tree.

The tool-to-experiment contract

A *tool* holds reusable science and speaks only through files: a runner reaches it by running its CLI as a subprocess, never by importing it. Tools emit machine-readable *data*, not finished plots; drawing the figure is the runner's job, so a result can always be redrawn. A tool writes a fixed file set (`config.json`, `output.json`, `manifest.json`, and its data) into scratch under `temp/`; the runner reads the manifest to learn the headline metrics, renders the figures, and aggregates everything into a committed `numbers.json` under `artifacts/data/`. Because each write-up reads its own `numbers.json` and figures straight from that run, a number on the page can't drift from the code that produced it.

To **add an experiment**: add or reuse a tool subcommand, write an `expNNN.py` runner that declares its commands and renders figures, then write an `expNNN.typ` write-up that reads the run. To **add a tool**: give it a directory under `tools/`, reuse the `setup_run_dir` / `write_output` pattern, and ship tests. A **writing** is just a `meta` + `body` pair in a `.typ` file, and an article needs no tool at all.

The full reference lives in `RULES.md`.