

Demolab — contents

Runbooks	2
Getting help	3
The vocabulary	4
The folder structure	5
Authoring slides	6
Writing style	7
The contract	8
Getting started	9
Introduction	11
Why demolab	14
Publishing your lab	16
Updating the engine	18
The command line	20
Anatomy of an experiment	22
Writing a writeup	25
Configuring your lab	28
Using another language	30

Runbooks

7 July 2026

Runbooks are demolab's **procedure layer**: named, repeatable jobs the agent runs step by step, showing each result and confirming before it moves on. Where a guide is always-on reference, a runbook is invoked when you want it: type its name in capitals (**LINT**, **DOCTOR**, **GETTING-STARTED**) and the agent opens it and drives it. Say **HELP** for the full menu. The canonical text is the Markdown source, linked below.

Getting in

- **GETTING-STARTED**: set up a fresh lab end to end: scaffold, brand, pick a stack, get it building and optionally published, then land a first experiment.
- **TOUR**: a guided walk through an existing repository, so you (or a new collaborator) learn where everything lives.

Checking the work

- **LINT**: audit the prose *and* the figures against the house style.
- **DOCTOR**: audit the repository's structure against the rules.
- **RED-TEAM**: attack a result's validity, hard, to surface what would break it.
- **STEELMAN**: build the strongest honest case for a result.
- **GROUND-CLAIMS**: back every claim with a run or a citation.
- **NEXT**: suggest what to run next.

Bringing work in

- **MIGRATE-CODE**: wrap an existing codebase, one experiment at a time, without rewriting the science.
- **MIGRATE-STACK**: switch the language your tools are written in (MATLAB, R, Julia, Octave).
- **FROM-JUPYTER**: convert a notebook into a tool, a runner, and a write-up.
- **FROM-PAPER**: reproduce a paper's result from scratch.

Housekeeping

- **EMBED-DOCS**: use demolab as a docs subfolder inside another project.
- **UPDATE**: bump the `demolab-cli` dependency to the latest engine and review what changed between versions.

Their always-on counterpart, the **guides**, each has its own page in this collection.

Getting help

7 July 2026

Stuck, or found a bug? Try self-service first: the runbooks and guides (`demolab docs` lists them, `demolab docs <NAME>` opens one) cover most operating questions. When you still need a human, here's where to go.

Where to ask

GitHub issues are the place: bugs, feature requests, and questions about the framework. Issues are searchable, so your question helps the next person and the fix lands in the open.

Install the GitHub CLI (`gh` — e.g. `brew install gh`, then `gh auth login`) and your coding agent can file issues for you: describe the problem and ask it to submit — it will gather the details below, write the report, and post it with `gh issue create`.

Writing a report that gets answered

Before opening an issue, run the doctor (say “*doctor the repo*”), check the guides and runbooks, and search existing issues. Then include, in order:

- **What you ran and what happened:** the exact command and the *full* error output, not a paraphrase.
- **Toolchain versions:** `uv --version`, `typst --version`, `demolab version`.
- **The commit:** `demolab` stamps the git SHA into the page/PDF footer, so paste that line or run `git rev-parse --short HEAD`.
- **Framework or content?:** state whether it's the engine (a `demolab` bug) or your own tool/experiment/writing. Ask if you're unsure.
- **A minimal repro** if you can: the smallest writing or tool that triggers it.

You can also just type `SUPPORT` in capitals and the coding agent will walk you through all of this interactively.

The full reference lives in `SUPPORT.md`.

The vocabulary

7 July 2026

demolab has a small vocabulary for its parts, and using the words precisely saves a lot of confusion. This page defines the ones you will meet most, with the distinctions that are easy to mix up called out explicitly. If you would rather be walked through it interactively, type GLOSSARY in capitals and the coding agent will take you term by term.

The core pieces

- **Tool:** a small program (Python by default) holding *reusable* science: a model or solver behind a small CLI that writes the contract files. It emits data, not plots, and reuse is the bar for making one.
- **Experiment:** a runner plus writeup sharing an id: `experiments/exp<NNN>.py` runs a tool and stages results, and `writings/exp<NNN>.typ` is the writeup. A tool is *reusable* science; an experiment is *one run* of it. Do not confuse the two.
- **Runner:** the `experiments/exp<NNN>.py` file: it runs a tool's CLI (or computes inline), renders figures from the data, and stages the record. It never imports a tool.
- **Article:** a prose-only writeup, `writings/ar<NNN>.typ`, with no runner and no data pipeline. Contrast an experiment, which does run one.
- **Writing:** any `writings/<id>.typ`: an experiment's writeup, an article, or a deck. It is a *meta* plus *body* pair the engine discovers and publishes.
- **Deck:** a `writings/<id>.slide.typ` Touying slide deck. It is paged-only, compiled to a stand-alone PDF and grouped under the `slides` collection (or its `meta.collection`, if set): listed, but not an entry.
- **Entry:** a published item that becomes its own page, meaning an experiment or an article. Decks are listed but are not entries.

The record and its wiring

- **Contract:** the file-based, language-neutral interface between a tool and an experiment: the tool writes a fixed set of files, the runner reads them by running the tool's CLI.
- **Artifacts:** the committed record under `artifacts/`: `artifacts/data/<id>/` holds a run's figures and `numbers.json`, `artifacts/pdfs/` holds shareable PDFs, and `artifacts/site/` is the gitignored web build.
- `numbers.json`: the aggregated, committed record of a run: each command's config plus headline metrics plus a provenance stamp. Writings read it so the numbers cannot drift.
- **Provenance:** the block stamped into every run: git commit SHA, a *dirty* flag, and a UTC timestamp, surfaced as a page and PDF footer.
- **Collection:** a group of entries sharing a `collection:` slug in their *meta*, grouped together on the homepage.

The full reference lives in GLOSSARY.md.

The folder structure

7 July 2026

A demolab repo has a place for everything, and the layout is the same in every repo. Once you know the four content directories, the engine, and the root files, you can find anything by name. A fresh lab is born from `uvx demolab-cli init` in an empty folder, which lays down the whole tree below — everything in it is yours.

```
demolab/
├─ tools/           the science - one directory per tool (tool.py + tests)
├─ experiments/     the runners - one expNNN.py per experiment
├─ writings/        the writeups - one .typ per entry, by id
├─ artifacts/       the committed record of every run
│  └─ data/<id>/    figures + numbers.json - the neutral record
│  └─ pdfs/         compiled PDFs (shareable)
│  └─ site/         the built web site - GITIGNORED
├─ demolab.yaml     branding + collections - marks the lab root
├─ HOUSESTYLE.local.md optional house-style overrides
├─ .demolab/        staged engine assets - GITIGNORED
├─ AGENTS.md · CLAUDE.md · README.md · pyproject.toml · .github/
└─ temp/           short-lived run scratch - GITIGNORED
```

The four content directories

These are yours. `tools/` holds the science, one directory per tool, each with its CLI in `tool.py` and a test beside it. `experiments/` holds the runners: one `expNNN.py` per experiment, which invokes a tool, renders figures, and stages its output. `writings/` holds the writeups, one `.typ` file per entry: an `expNNN.typ` reads an experiment's run, an `arNNN.typ` is a prose-only article, and an `arNNN.slide.typ` compiles to a standalone slide deck. `artifacts/` is the committed record: `data/<id>/` keeps each run's figures and `numbers.json`, and `pdfs/` keeps the shareable PDFs. Both of those are in git; `artifacts/site/` and `temp/` are gitignored because the build regenerates them.

What ties an experiment together is its *id*, not a folder. The same `exp000` threads through all three: `experiments/exp000.py` runs it, `artifacts/data/exp000/` records it, `writings/exp000.typ` writes it up.

The engine and the root files

The engine is the *black box*. It holds the Typst publisher, the scaffold templates, the agent runbooks, and the guides, and it lives inside the installed `demolab-cli` package rather than in your tree, so you never hand-edit it. The CLI stages the few files Typst needs to read — `lib.typ` and the web assets — into the gitignored `.demolab/` directory at the lab root, and refreshes it automatically. Updating demolab is an ordinary dependency bump: `uv lock --upgrade-package demolab-cli && uv sync`.

Two root files carry your configuration: `demolab.yaml` for branding and collections (it also marks the lab root), and the optional `HOUSESTYLE.local.md` for house-style overrides. The remaining root files, `AGENTS.md`, `CLAUDE.md`, `README.md`, `pyproject.toml`, and the CI workflow, are thin stubs laid down once by `init` — yours to edit, and never touched by an update.

Learning it interactively

Type **STRUCTURE** in capitals and the coding agent will walk you through this tree, path by path, in your own repo.

The full reference lives in `STRUCTURE.md`.

Authoring slides

7 July 2026

A deck in demolab is a talk, not a writeup. It lives at `writings/<id>.slide.typ` and is built with Touying into a standalone PDF. The `.slide.typ` filename is the marker: the build compiles the deck on its own, links it from the site, and deliberately excludes it from the HTML pages and the book. Touying is paged-only and does not survive HTML export, so a deck stays a PDF and nothing else.

One structural difference from a prose entry: a deck declares `#let meta` so the site can list and link it, but it never declares a `#let body`. Its figures come from the same committed run outputs the writeups read, out of `artifacts/data/`, never from ad-hoc images.

Size against the canvas

A slide is a fixed rectangle: 842 by 474 points at 16:9. Once the title header and footer margins take their share, you have roughly 350 points of usable height on a titled slide. That number is the budget you lay everything out against. Size images in absolute points, not percentages: a percentage resolves against whatever region encloses it, which inside a grid cell is rarely the whole slide, so percent-sized figures come out unpredictably small and can overflow. Give each figure row a height that suits its shape, sum the rows, gutters, and captions, and confirm the total stays under budget. If a slide is mostly whitespace the figures are too small; if things do not fit, split the slide rather than shrinking text below readable sizes.

Check the page count

Overflow is silent. Oversized content does not clip or raise an error; Touying quietly spills it onto an extra page, and the deck grows without warning. So after every layout change, count the pages and compare against the number of slides you meant to have. Render the deck to images and count them, rather than trusting cached metadata from Finder or Spotlight, which goes stale. Then eyeball the dense slides at higher resolution to catch clipped captions or terms swallowed into a subscript.

Lift layouts from the catalog

You do not re-derive a layout from scratch. Every layout is a named, tested block in the gallery deck, and the guide carries an index of names paired with when to use each one: bullets, two-column comparisons, code panels, equation-with-terms, figure grids, big-number statements, section dividers, and more. You find the name, copy that block out of the gallery, and swap in your own content. The gallery holds the single page-count-checked copy of each layout, and a test keeps the index and the gallery in sync, so an entry can never point at a missing block.

You can also just type `SLIDES` in capitals and the coding agent will walk you through all of this interactively.

The full reference lives in `SLIDES.md`.

Writing style

7 July 2026

HOUSESTYLE is the taste demolab writes with, so the whole lab reads as one voice. It is **style**, softer than the hard invariants, and you follow it unless you have a reason not to. If you type HOUSESTYLE in capitals, the coding agent will walk you through it interactively.

How the prose reads

A write-up is written for a scientist, not a changelog. The title is a plain sentence about the science, never prefixed with the entry id or date, since those already sit in the meta strip. Lead with the claim in plain language, then let the figure and the numbers carry the evidence, and keep the motivation short. Report results honestly, including partial and negative ones, folded into the prose or a caption rather than hidden. One notable habit: no em-dashes in running prose. They are the loudest tell of machine-written text, so a comma, colon, parentheses, or full stop stands in.

How math is set

Equations are written in native Typst so they render as selectable math on the web and typeset in the PDF, never pasted as images. The single most important rule: after an equation, define *every* symbol in a list. A reader should never have to guess what a term means. For “approximately” use $\approx$ in prose and **approx** in math, never a tilde, because in Typst a tilde is a non-breaking space and quietly vanishes.

Figures, captions, and numbers

Every figure is drawn by a runner from the tool’s data and embedded in both the web page and the PDF from one rendered asset, so the plot style lives in one shared place instead of being reset per figure. Line plots go to vector SVG, dense or photographic content to high-dpi PNG, always on a white background with alt text. Ink is near-black by default; a single accent colour is earned only to separate traces sharing an axis. Captions are written to be read cold, standing on their own: a lead clause saying what the figure shows, a body naming each axis with units and defining every symbol, then one honest takeaway.

Numbers a run produced are never hand-typed, in the prose or in a caption. They come from the run’s recorded output, so a stated result always traces back to the computation and cannot drift. Likewise, citations use the citation helpers rather than typed brackets, so numbering, links, and hover-popovers stay correct as you reorder.

Making it your own

This is demolab’s *default* style, and you can override it. Drop a HOUSESTYLE.local.md in the repo root: by default your rules add to and supersede these, or you can tell it to replace them entirely and inherit none of demolab’s taste. Either way the underlying tool-to-experiment contract still holds; only the style is yours to bend.

The full reference lives in HOUSESTYLE.md.

The contract

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`.

Getting started

7 July 2026

demolab is a lab notebook for computational science. You write the science as code, an experiment runs it and emits data, a write-up reads that data, and one build turns the whole repository into a website and a set of PDFs. Numbers and figures come from the run, never retyped, so a result on the page can't quietly drift from the code that produced it. A coding agent operates it in plain language; you stay in the loop.

Install

Open a coding agent in an empty folder and paste:

Run `uvx demolab-cli init` here, then follow its GETTING-STARTED runbook strictly.

It runs the GETTING-STARTED runbook and sets everything up with you: the toolchain — `uv` for Python and `Typst` for publishing — then your own lab, born from the `demolab-cli` package, and your first experiment. You drive everything through the `demolab` command, which wraps `uv` and `typst`; you never call `pip` or `python` directly.

Your first run

The agent drives this: GETTING-STARTED stands the lab up, then builds your first experiment with you — your own science, or a suggested starter — and you watch it land on a live page. By hand, the loop is:

```
uvx demolab-cli init # lay down the lab: root files + writings/ experiments/ tools/
artifacts/
demolab dev          # serve the site at localhost:3000, live-reloading on save
uv run python experiments/exp000.py # run an experiment end to end (once you've
written one)
```

Change a parameter, run the experiment again, and watch the page update — figures and numbers, no prose touched. That's the whole point. `demolab build` compiles everything once; `demolab test` runs the suite. (Want a worked example to model your first experiment on? `demolab docs STARTERS` prints the reference dir — `monte-carlo-pi` is the canonical starter; build it as your own.)

How a lab is laid out

Four content directories, each with one job:

- `tools/`: the **science**. Reusable models and solvers, one directory per tool. A tool emits **data** (a `numbers.json` plus the data behind each figure), never plots, and stays language-agnostic.
- `experiments/`: the **runners**. One `expNNN.py` per experiment: it calls a tool (or computes inline), renders the figures into `artifacts/data/expNNN/`, and stages a `numbers.json` of the headline metrics.
- `writings/`: the **write-ups**. One `.typ` per entry. It reads the run, `#let run = json("/artifacts/data/expNNN/numbers.json")`, embeds the figures, and pulls every number from that file. Prose-only articles (like this one) and slide decks live here too.
- `artifacts/`: the committed **record**: `data/<id>/` (figures + numbers) and `pdfs/` (a PDF per entry, plus a bound book). The built website (`site/`) is regenerated by CI, so it's gitignored.

The engine itself lives in the installed `demolab-cli` package: a black box you never edit, updated with an ordinary dependency bump.

The loop

The discipline is one rule with teeth: *nothing on the page is typed by hand*.

1. A tool computes and writes data files.
2. A runner turns that data into figures and a `numbers.json`.
3. A write-up reads the `numbers.json` and embeds the figures.
4. `demolab build` compiles the repository into a website, a PDF per entry, and a book.

Change the code, rebuild, and the prose, figures, and numbers update together. Every result carries the git provenance of the run that made it.

Publish

One build emits three things from a single source: a **website** (HTML with real, selectable math), a **PDF per entry**, and a bound **book**. To put it online, `demolab deploy-setup` drops a GitHub Pages workflow; enable Pages in the repository settings and push. The site uses relative links, so it works under any path.

Driving it with an agent

`demolab` is meant to be run by a coding agent. You type a **name in capitals** and it acts:

- **HELP**: list everything it can do.
- A **guide** name (`RULES`, `HOUSESTYLE`, `SLIDES`, `STRUCTURE`, `GLOSSARY`, `SUPPORT`): it walks you through that reference.
- A **runbook** name (`LINT`, `DOCTOR`, `MIGRATE-STACK`, ...): it runs that procedure step by step.

The reference layers are documented right here in this collection: a page for each **guide** (always-on conventions), plus a **runbooks** index (the on-demand procedures).

Introduction

9 July 2026

Here is an observation. Software engineers use coding agents constantly; scientists, far less so. That gap is not about talent or curiosity, it is about tooling: the agent arrived in a form built for shipping software, not for running experiments and writing them up. demolab is an attempt to close that gap. It is a lab notebook for computational science in which the science lives as code, an agent operates it in plain language, and one build turns the whole repository into a website and a set of PDFs. This page makes the case for it, in the same two acts as the talk it is drawn from: first the agents, then the workflow built on them.

Coding agents got good

A coding agent is not autocomplete. It reads a repository, runs programs, edits files, and checks its own work; it can do anything a terminal can do. The line runs autocomplete (2021), to chat (2023), to agents that do real work (2025), and only the last step made it useful for anything as unforgiving as science.

The clearest evidence is SWE-bench Verified [1, 2], a benchmark of real, human-validated GitHub issues that an agent must resolve end to end. The trajectory, shown below, is the point: at the benchmark's 2024 launch the best systems closed about a third of the issues, and by late 2025 that figure had climbed past three quarters. A tool that was a novelty two years ago now does a large fraction of a working engineer's tickets unaided. Many such agents exist, commercial and open, so there is no vendor lock-in.

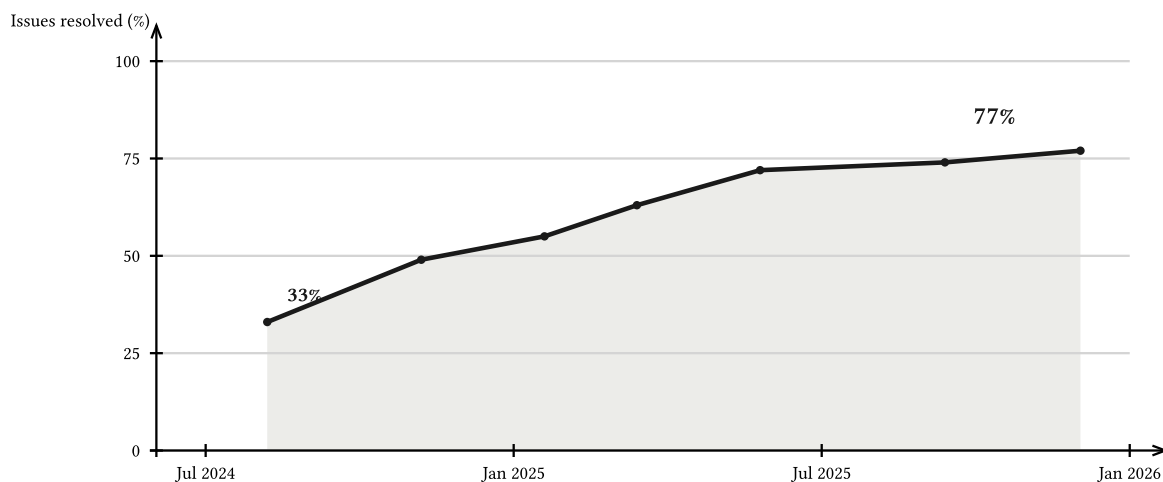


Figure 1: Share of real, human-validated GitHub issues a coding agent resolves end to end, on SWE-bench Verified (swebench.com), from the benchmark's 2024 launch to late 2025. The vertical axis is percent of issues resolved; the horizontal axis is date. The resolved fraction climbs from 33% to 77% in about eighteen months. These are software bugs, not science, so the level does not transfer to research; the slope is what matters.

It would be dishonest to sell only the strengths. Agents do the same work faster, clear the backlog of things you never had time for, document as they go, and read a paper straight into runnable code, and they work far better over a whole repository than over a snippet. They are also confidently wrong: they invent APIs, numbers, and citations. They leave you with comprehension debt, code you shipped but do not understand. Their judgement is weak; an agent will not kill a bad experiment on its own. So the honest summary is short. A coding agent

is a powerful tool that needs supervision, and a framework worth having is one that supplies the rails.

What demolab is

demolab is that framework. It writes code, runs programs, and documents the results, and it holds the whole thing to one loop: run, produce data, write it up, publish. One build compiles the repository into a website (with real, selectable math) and a PDF per entry plus a bound book. The discipline underneath is a single rule with teeth: nothing on the page is typed by hand. Numbers and figures come from the run, so a result cannot quietly drift from the code that produced it, and every result carries the git provenance of the run that made it. It is reproducible by default, and it stays reproducible a year later because one command rebuilds all of it.

Those properties answer real failure modes. Results drift when numbers are retyped and figures go stale; here they come from the run. Reproducibility rots until, a year on, nothing builds; here one command rebuilds everything. Code and paper drift apart into separate repositories; here the experiment and its write-up live together. And agents need rails; here they operate in plain language while the science stays yours.

A few principles hold the shape together:

1. **Tools compute, experiments analyse.** A clean split between the reusable science and the runner that exercises it.
2. **One experiment is a runner plus a write-up.**
3. **Components talk through files, never imports.** A tool emits data; a runner reads it.
4. **Raw data is disposable; the record is committed.**
5. **Bring any stack.**
6. **Provenance is built in.**
7. **Documentation is everything.**

The shape of a repository

A lab is four content directories, each with one job:

- **tools/:** the science, as reusable models and solvers. A tool emits data (a `numbers.json` plus the data behind each figure) and stays language-agnostic.
- **experiments/:** the runners. One `expNNN.py` per experiment calls a tool, renders the figures, and stages a `numbers.json` of the headline metrics.
- **writings/:** the write-ups. One `.typ` per entry reads the run and embeds its figures and numbers. Prose articles like this one, and slide decks, live here too.
- **artifacts/:** the committed record, holding the figure data and a PDF per entry.

The engine itself ships inside the `demolab-cli` package, a black box you never edit and update with a dependency bump. Publishing is built on Typst, a modern take on LaTeX: instant incremental compiles instead of slow multi-pass ones, readable errors with line numbers instead of macro soup, and one source that renders to both web and PDF instead of PDF alone. Because Typst imports files, the numbers on the page are the numbers from the run.

Driving it with an agent

demolab is meant to be run by a coding agent, though you can operate every command by hand. You type a name in capitals and it acts. **HELP** lists everything it can do. A guide name (`RULES`, `HOUSESTYLE`, `SLIDES`, `STRUCTURE`, `GLOSSARY`, `SUPPORT`) walks you through an always-

on convention. A runbook name (`GETTING-STARTED`, `LINT`, `DOCTOR`, `FROM-PAPER`, `MIGRATE-STACK`, and the rest) runs an on-demand procedure step by step: read a paper into code, convert a notebook, migrate the language, update the engine. Underneath it all is an ordinary CLI, so `uv run python experiments/exp000.py` runs an experiment, `demolab dev` serves the site with live reload, and `demolab build` compiles the lot. The default stack is Python, but tools talk through files rather than imports, so you can switch to MATLAB, R, Julia, or Octave by asking, and the contract and the typesetting stay put.

Where it's going

Everything above is `demolab` today: implemented and ready to use. Two directions define where it goes next.

The first is autoresearch. Give the agent a goal and some compute, and it runs the loop itself: goal, code, compute, read the data, document, iterate. Because the rails already exist, every attempt lands as a committed experiment and write-up with provenance and a published trail, for free. It is built, and still being tested.

The second is scale. Version one is one person's lab notebook. The next step is a lab on a GitHub organisation: many researchers, one shared flow, with questions and answers, code review, and shared code compounding across the group. The notebook becomes the lab.

References

1. Jimenez CE, Yang J, Wettig A, Yao S, Pei K, Press O, Narasimhan K (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR 2024*. doi:10.48550/arXiv.2310.06770
2. OpenAI (2024). Introducing SWE-bench Verified. Live leaderboard at swebench.com.

Why demolab

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.

Publishing your lab

11 July 2026

A demolab repo compiles to plain files, and publishing is deliberately boring: build once, put the output on a static host. demolab ships with *no* active deploy workflow; going public is an explicit opt-in, one command plus a couple of clicks in the GitHub UI. This page walks the whole path.

What a build emits

demolab build

One build turns the repository into two committed-or-served outputs under `artifacts/`:

- `artifacts/site/`: the website. HTML with real, selectable math and **relative links**, so it works at a domain root, under a subpath, or straight off a file system. This directory is regenerable build output (`demolab clean` deletes it), so it stays out of git; CI rebuilds it on every deploy.
- `artifacts/pdfs/`: a PDF per entry, plus the bound book, and any standalone slide decks compiled by `demolab slides`.

Everything below is just ways of getting `artifacts/site/` onto a server.

Opting in to GitHub Pages

demolab deploy-setup

This copies two workflow files from the engine into your repo's `.github/workflows/`:

- `deploy.yml`: the production deploy. On every push to `main` (or a manual dispatch) it installs Typst and uv, runs `uv sync` and `uv run demolab build`, and publishes `artifacts/site/` to the root of a `gh-pages` branch as a single snapshot commit.
- `preview.yml`: per-PR previews. Every pull request gets its own built copy of the site under `pr-preview/pr-N/` on the same `gh-pages` branch, a sticky comment on the PR linking to it, and an automatic teardown when the PR closes or merges. So a reviewer reads the rendered pages and PDFs at a URL, not a raw diff.

Both files are engine templates: rerun `demolab deploy-setup` after an engine update to refresh them.

The GitHub-side clicks

Three settings can't be scripted, so the command prints them and you click. In the repo on GitHub:

1. *Settings* → *Pages* → *Source*: choose **Deploy from a branch**, then branch `gh-pages / (root)`. The branch won't exist until the first push to `main` runs the workflow; flip the setting once it does.
2. *Settings* → *General* → *Pull Requests*: enable **Automatically delete head branches**.
3. (Recommended) *Settings* → *Branches*: protect `main`.

Then commit the workflows and push: `main` deploys the site, and each PR gets its own preview URL.

The branch-based source matters. The workflows use the older build-then-commit-to-`gh-pages` flow rather than the GitHub-Actions Pages flow, precisely so production and the PR previews

can share one branch: the production deploy prunes stale files but explicitly preserves `pr-preview/`, and the two use separate concurrency groups so they never race a commit.

One caveat: pull requests from forks get a read-only token and can't write `gh-pages`, so previews are skipped for external contributors. That's expected, and fine for a personal lab; don't "fix" it by switching the trigger to `pull_request_target`, which would run untrusted PR code with a write token.

Custom domains

Set your domain in *Settings* → *Pages* as usual. GitHub records it by writing a `CNAME` file to the `gh-pages` branch, and `deploy.yml` excludes that file from its cleanup, so the domain survives every deploy. Nothing demolab-specific to configure.

Any static host

GitHub Pages is the packaged path, not a dependency. The site is plain files with relative links and no server-side anything, so any static host works the same way:

```
demolab build
# then upload artifacts/site/ to Netlify, Cloudflare Pages, S3, your own nginx, ...
```

Point the host at `artifacts/site/` as the publish directory and you're done.

Inside another project

If your lab isn't a standalone repo but the docs folder of a bigger project, the recipe changes: `deploy-setup`'s workflows assume the lab is the repo root, so you skip them and add a Pages workflow to the **host** repo instead, modelled on `deploy.yml`. The EMBED-DOCS runbook walks that path end to end.

Updating the engine

11 July 2026

Updating demolab is a dependency bump, not a migration. The engine is the `demolab-cli` package: build code, Typst templates, runbooks, guides. Your lab is content. The two never mix, so pulling a new engine can't disturb anything that's yours.

The engine/content split

An update touches exactly one thing: the installed package. Everything the update leaves alone falls into two zones.

- **Yours, never touched:** `writings/`, `tools/`, `experiments/`, `artifacts/`, `demolab.yaml`, `HOUSESTYLE.local.md`, `pyproject.toml`, and the root stubs (`AGENTS.md`, `README.md`, `CI`). Branding (the wordmark and PDF titles) lives in `demolab.yaml`, outside the package, so an update can't touch your lab's identity.
- **Machine-managed:** the gitignored `.demolab/` staging directory at the lab root. The CLI refreshes it automatically; you never hand-edit it (see below).

Ask your agent

Say **UPDATE** (or “update demolab”) to your coding agent and it follows the `UPDATE` runbook: check the installed version against the latest release, show you the changelog entries in between, bump the dependency, rebuild, and commit the lockfile change. One hard gate: if the **major** version differs, the agent stops and asks before bumping, because a major release may need edits to your own content.

By hand

The whole update is one line, then a rebuild:

```
uv lock --upgrade-package demolab-cli && uv sync
demolab build && demolab test
```

`demolab version` prints the installed engine; the newest release is on PyPI.

What changed

The engine ships its own changelog. Read it before or after the bump:

```
demolab docs CHANGELOG --print
```

(Before updating, the same entries are behind the Changelog link on the PyPI page.) Each entry says what the release added or fixed, and a major entry says exactly which of your content it may require you to edit.

How versioning works

demolab uses SemVer: a **patch** is fixes, a **minor** is a new backward-compatible capability, and a **major** is a break that may need edits to your content (the tool ↔ experiment contract, the meta schema, or import paths). Two mechanisms keep the version honest:

- `uv.lock` pins the exact engine version, so a fresh clone of your lab rebuilds with the same engine until you bump it. Reproducibility is the lockfile's job.
- On the first `demolab build` after a version change, the CLI notices, refreshes the staged `.demolab/` directory automatically, and prints a one-line notice pointing at the changelog. Nothing for you to reconcile.

If a build breaks after an update

First read the changelog entry for the version you just pulled: if the release was a major, its entry lists the content edits it expects, and that is almost always the fix. Then say **DOCTOR** to your agent, which audits the repo against the conventions and points at anything out of contract. And because the update was one committed change to `pyproject.toml` and `uv.lock`, reverting that commit and running `uv sync` puts you back on the previous engine while you sort it out.

The command line

11 July 2026

`demolab` is one command runner wrapping `uv` (Python) and `typst` (publishing); you never call `pip` or `python` directly. Every command finds the lab by walking up from wherever you are, like `git` finding its root, so it works from any subdirectory. Run `demolab` with no arguments for the live catalog; this page mirrors its groups. Most of these are driven by a coding agent following a runbook. The four a human types daily are `dev`, `run`, `build`, and `test`.

Start here

init Start a new lab in the current directory: root files, the content structure, a tests CI workflow, and a first git commit (skipped inside an existing repository or without a git identity). Refuses a non-empty directory and refuses to run inside an existing lab; `--force` overrides both. Typically run once, by the agent, at the top of the GETTING-STARTED runbook.

docs Bare `demolab docs` prints the agent manual plus the menu of runbooks and guides; `demolab docs <NAME>` prints that document's path, and `--print` prints its contents instead. The menu also lists `AGENT`, `CHANGELOG`, `DEMO`, and `STARTERS`. This is the agent's orientation command; a human rarely needs it.

Setup

install Install the lab's Python dependencies (`uv sync`). Rarely needed by hand: the commands that need the venv go through `uv` and create it on demand.

version Print the installed engine version.

Scaffolding

Agent territory: these run inside runbooks, not day to day.

scaffold (Re)lay the working-tree structure (`writings/`, `experiments/`, `tools/`, `artifacts/`, plus config). Non-destructive: it never clobbers an existing file, so it doubles as a repair command. For a worked first experiment, model on a starter rather than installing bulk content: `demolab docs STARTERS` prints the dir (`monte-carlo-pi` is the canonical one).

deploy-setup Opt in to GitHub Pages: copies `deploy.yml` (build main to the `gh-pages` branch) and `preview.yml` (per-PR previews) into `.github/workflows/`, then prints the remaining GitHub-UI clicks, which can't be scripted.

The loop

Running an experiment There is no `run` command — invoke the runner directly. `uv run python experiments/exp000.py` executes `experiments/exp000.py` in the lab's venv; the runner stages figures and a `numbers.json` under `artifacts/data/exp000/`. Daily driver.

new Not a scaffolder: it prints how to start a new experiment, which is to ask your coding agent to model one on an existing runner + writeup pair.

Publishing

dev Serve the site with hot-reload and in-browser build errors. Takes an optional port (default: first free from 3000). `--demo` serves the packaged docs site from a scratch copy under `temp/` instead of your lab; `--landing` (only with `--demo`) previews its landing page too. For your own landing page, put a `landing.typ` at the lab root and plain `demolab dev` renders it. The other daily driver:

`demolab dev 4000 --demo --landing`

build Build the whole bundle once: the website into `artifacts/site/` and the PDFs plus the bound book into `artifacts/pdfs/`. What CI runs.

slides Compile standalone Typst decks (any `writings/*.typ` without the meta+body shape) to `artifacts/pdfs/`.

playground Launch a lab's interactive Streamlit app (`experiments/playground.py`), if it has one, at `http://localhost:8501`.

Quality & housekeeping

test Run the Python test suite (pytest). A fresh lab with no tests yet passes rather than fails. Run it before committing.

clean Delete the regenerable build output: `temp/` and `artifacts/site/`. The committed record under `artifacts/data/` and `artifacts/pdfs/` is untouched.

Anatomy of an experiment

11 July 2026

The contract states the rules; this page shows them, on the smallest example that still exercises every part. Take `exp000`, an experiment that estimates π by Monte Carlo, and walk it end to end: the runner that produces the run, every file it leaves behind, and the write-up that reads them. It is not something that ships in your tree, it is the minimal shape a real experiment takes. Nothing here is new machinery, it is the same tool-to-experiment contract (RULES §4) seen from the ground.

The science is one line. Throw N random points into the unit square, count the fraction that land inside the quarter circle, and four times that fraction estimates π :

$$\pi \approx 4f, \quad f = \frac{1}{N} \sum_{i=1}^N \mathbb{1}[x_i^2 + y_i^2 \leq 1]$$

where N is the number of sampled points, f the fraction that land inside the quarter circle, $(x_i, y_i) \sim \text{Uniform}(0, 1)^2$ the i -th sample point, and $\mathbb{1}[\cdot]$ the indicator, 1 when a point is inside the unit quarter circle and 0 otherwise.

An experiment is two files plus a committed record: a runner `experiments/exp000.py`, a write-up `writings/exp000.typ`, and the `artifacts/data/exp000/` directory the runner stages. The science itself lives one layer down, in the reusable tool `tools/montecarlo/tool.py`, which the runner reaches only by running its command line.

The runner

A runner does four things: call a tool (or compute inline), render the figures, aggregate the headline numbers, and stamp a reproducer. `exp000.py` declares which tool commands it bundles and where things live:

```
TOOL = ROOT / "tools" / "montecarlo" / "tool.py"
TEMP = ROOT / "temp" / "montecarlo"
ARTIFACTS = ROOT / "artifacts" / "data" / "exp000"
COMMANDS = ("pi",)
```

It reaches the tool as a subprocess, never an import, which is what keeps the tool generic and the contract language-neutral (RULES §4.5):

```
subprocess.run([sys.executable, str(TOOL), *args], check=True)
```

Seeding lives in the tool, not the runner. The sampler draws its points from a generator seeded by a `--seed` argument (default 0), and that value is written into the run's `config.json`, so the same seed reproduces the same estimate and the same scatter. Plotting is the runner's job (RULES §4.2): `plot_scatter` reads `temp/montecarlo/pi/pi.csv` and saves the point cloud as PNG, each point coloured by whether it fell inside the quarter circle. The tool emits data; the figure is always redrawable.

Finally `main()` runs the command, renders the figure, aggregates the numbers, and drops the reproducer:

```
for command in COMMANDS:
    run_tool(command)
plot_scatter("pi", ARTIFACTS / "scatter.png")
numbers_path.write_text(json.dumps(collect_numbers(), indent=2) + "\n")
provenance.write_run_sh(ARTIFACTS)
```

The data it drops

The command writes a fixed scratch set into `temp/montecarlo/pi/` (RULES §4.3), overwriting the previous run: `config.json` (the argparse args), `output.json` (the metrics), a `manifest.json` naming the headline metrics, an `output.log`, a scratch `run.sh`, and the run's `pi.csv` (one row per sampled point). That whole directory is gitignored scratch. The committed record is what the runner stages into `artifacts/data/exp000/`, three files:

- `scatter.png`: the sampled points, coloured inside versus outside the quarter circle, raster for a dense point cloud.
- `numbers.json`: the aggregated headline numbers and config, read by the write-up.
- `run.sh`: the committed reproducer, `exec uv run python experiments/exp000.py`.

`collect_numbers` reads the command's `manifest.json` to learn which fields are headline, then merges its `config.json` with exactly those fields. It never hardcodes a metric name:

```
config = json.loads((TEMP / "pi" / "config.json").read_text())
output = json.loads((TEMP / "pi" / "output.json").read_text())
manifest = json.loads((TEMP / "pi" / "manifest.json").read_text())
numbers = {
    "config": config,
    **{f: output[f] for f in manifest["headline_metrics"]},
}
```

With a single command the record collapses to one flat object: a `config` block (the flat args, plus a `_provenance` stamp) and the headline metrics beside it.

```
{
  "config": {
    "command": "pi", "n": 100000, "seed": 0,
    "_provenance": { "commit": "db62207...", "dirty": true,
                    "generated_at": "2026-07-05T11:39:06+00:00" }
  },
  "pi_estimate": 3.1417,
  "abs_error": 0.000107
}
```

A multi-command experiment keys this object by command instead (RULES §4.6); here the one command collapses to the single entry above.

The `_provenance` block is stamped by the tool when it sets up the run: the git commit that produced the numbers, a `dirty` flag for uncommitted code, and a UTC timestamp (RULES §4.7). It rides along in `numbers.json`, so every published result records exactly which code made it.

The write-up that reads it

`exp000.typ` never types a number. It opens the record once,

```
#let run = json(data-file("exp000/numbers.json"))
```

then pulls everything from `run`. The estimate in the prose is an interpolation, not a literal:

```
#calc.round(run.pi_estimate, digits: 4)
```

The `scatter` embeds straight from the staged asset (`image(data-file("exp000/scatter.png"), ...)`), the parameter table comes from `#numbers-table(run, ...)`, and the git footer from `#provenance-footer(run.config)`. Change `--n`, re-run, rebuild, and the prose,

figure, and numbers move together. That is the whole point: a number on the page cannot drift from the code that produced it (RULES §5.4).

What `uv run python experiments/exp000.py` does

One command, start to finish:

1. `uv run python experiments/exp000.py` executes the runner in the lab's `uv`-managed `venv`.
2. The runner calls the tool once as a subprocess (`montecarlo pi`). The call writes its scratch file set into `temp/montecarlo/pi/`.
3. The runner renders `scatter.png` from `pi.csv` into `artifacts/data/exp000/`.
4. It aggregates `numbers.json` (config plus headline metrics) and drops `run.sh`.
5. `demolab build` later compiles `writings/exp000.typ`, which reads that `numbers.json` and embeds the figure, into the web page and the PDF.

CI does not run experiments, so `artifacts/data/exp000/` is committed: that record, not the ephemeral `temp/`, is what reaches the site (RULES §5.3).

Start your own

You now have the shape. Estimating π by Monte Carlo is the smallest thing that still touches every part of the contract; a real experiment swaps in real science and keeps the frame. To make one, copy the shape: a runner modeled on `exp000.py`, a write-up modeled on `exp000.typ`, and a tool if the science is worth reusing (a genuine one-off can compute inline and stamp its own provenance, RULES §4.1). Or just ask the coding agent: `demolab new` prints the same advice, and the *NEXT* and *GETTING-STARTED* runbooks walk an agent through standing up a first experiment step by step.

Writing a writeup

11 July 2026

A writeup is one `writings/<id>.typ` file: a `meta` block and a `body` block. Everything else is helpers from `lib.typ`, imported root-relative. This page is a working reference, organised by what you are trying to do. Its examples use a minimal experiment, `exp000`, that estimates π by Monte Carlo (drop N seeded points in the unit square, count the fraction inside the quarter circle). For how the prose should read, see [Writing style](#) this is the mechanics.

Start a new entry

Two top-level definitions make a writeup, and `build.py` discovers entries by them. The filename stem is the entry id: `exp000.typ` becomes `exp000`, served at `exp000.html` with a PDF at `pdfs/exp000.pdf`. `meta` carries `title`, `date`, and the optional `description`, `collection`, `status`, and `order`. `order` is an integer: any entry carrying one makes its collection list in that curated order (ascending) instead of newest-first, and unranked entries trail.

```
#let meta = (  
  title: "Estimating  $\pi$  by Monte Carlo sampling",  
  date: "2026-05-13",  
  description: "One line, shown under the title and on listings.",  
  collection: "estimators",  
  status: "final",  
  order: 2,  
)
```

```
#let body = [ ... ]
```

Write headings normally with `==`. Each one gets a slug id on the web, so any section is deep-linkable (`exp000.html#methods`) with a permalink that appears on hover.

Pull in the run

Numbers and figures come from the run, never retyped. `data-file(rel)` resolves a path under `artifacts/data/`, and `json(...)` reads the run's `numbers.json` into a value you index. That object holds a `config` block (the run's args plus a `_provenance` stamp) and the headline metrics beside it, here `pi_estimate` and `abs_error`.

```
#import "/.demolab/lib.typ": data-file, numbers-table  
#let run = json(data-file("exp000/numbers.json"))
```

Interpolate a metric inline with `#calc.round(run.pi_estimate, digits: 4)`. To print the run's parameters and metrics as a table, pass it to `numbers-table`: it lists the `config` as parameters and the remaining keys as metrics, straight from the run.

```
#numbers-table(run, title: "Monte Carlo run parameters")
```

Place figures and video

A data figure is a standard Typst `#figure` wrapping an `#image` of a tool-rendered asset under the run's data directory. Give every image an `alt:` line so the web is accessible. The caption should stand on its own, and any number in it comes from the run too.

```
#figure(  
  image(data-file("exp000/scatter.png"), width: 100%, alt: "Sampled points coloured  
by whether they fall inside the quarter circle"),  
  caption: [#calc.round(run.config.n, digits: 0) points in the unit square, coloured  
by the
```

```
    inside-circle test, giving  $\pi \approx \text{\#calc.round(run.pi\_estimate, digits: 4).}],$ 
)
```

A rendering plays on the web with `video`, referenced by `basename` (`build.py` emits every mp4 as a bundle asset). In the PDF it becomes a short note pointing at the web edition.

```
#import "/.demolab/lib.typ": video
#video("samples.mp4", caption: [Samples accumulating as the estimate converges on
 $\pi$ .])
```

Write math

Write real Typst math: inline with `$...$`, display with a block. It renders as selectable MathML on the web and is typeset in the PDF, so never paste an equation as an image. Define every symbol after the equation.

```
$ \hat{\pi} = 4 \cdot \frac{1}{N} \sum_{i=1}^N \mathbb{1}[x_i^2 + y_i^2 \leq 1] $
```

where $\hat{\pi}$ is the estimate, N the number of samples, $x_i, y_i \sim \text{Uniform}(0, 1)$ the coordinates of the i -th sample, and $\mathbb{1}[\cdot]$ the indicator that is 1 when the point falls inside the quarter circle and 0 otherwise. The estimate converges as $\hat{\pi} \approx \pi$ for large N .

Cite prior work

Two helpers manage numbering, linking, and the web hover-popover, so never hand-type a bracket or a list. `#cite(1, 2)` renders [1, 2] and links each number to its entry; attach it directly to the word it follows, with no space. `#reference-list(items)` renders the numbered References section, each item a dict with free-Typst `text` and an optional `doi` that links to `https://doi.org/<doi>`. Numbering is positional: keep the inline numbers in step with the list order.

```
#import "/.demolab/lib.typ": cite, reference-list
```

The estimator's error falls as `$\frac{1}{\sqrt{N}}$`, the standard Monte Carlo rate`#cite(1)`.

```
#reference-list((
  (text: [Author A, Author B (2020). Title. _Journal_ 12:34.], doi: "10.1000/xyz"),
))
```

Mark work in progress

Set `status` to a lifecycle value: `draft`, `building`, `revising`, or `final`. `final` (the default) shows nothing, so only unfinished work is flagged; the badge appears next to the date everywhere the entry lists. When a figure's asset is not ready yet, `pending-figure` reserves its footprint so the page does not reflow when the real plot lands, and it still numbers as a normal "Figure N".

```
#import "/.demolab/lib.typ": pending-figure
#pending-figure(
  caption: [Absolute error against sample count N.],
  note: [re-run in flight],
  ratio: 16 / 9,
)
```

Swap it back to a real `#figure` once the asset exists: a `pending-figure` left in a `final` entry is a lint smell.

Stamp provenance

Close the body with the git-commit footer. Pass the run's `config`, which carries the `_provenance` block the runner stamped: `provenance-footer` prints the short commit and the build date, so a reader can trace the page back to the exact code that made it.

```
#import "/.demolab/lib.typ": provenance-footer  
#provenance-footer(run.config)
```

That is the whole authoring surface. For the prose, math, figure, and caption conventions that keep the lab reading as one voice, see [Writing style](#).

Configuring your lab

11 July 2026

A lab is configured by one file: `demolab.yaml` at the repository root. It does two jobs. It sets the **branding** (the site’s name, tagline, and byline) and it registers the **collections** that group your writings on the homepage. Every key inside it is optional: the engine merges what you write over its own defaults, so a missing key just means the default. The file itself, though, is not optional. It is the **lab marker**: the `demolab` command finds the lab root by walking up from wherever you are until it hits a `demolab.yaml`, the way git finds `.git`. Edit it freely, but don’t delete it.

Updates never touch this file. Like `HOUSESTYLE.local.md`, it is yours; the engine reads it and fills in the rest.

Branding

Six keys, each with an engine default:

- **name**: the site’s title. Renders as the homepage heading and in every page’s browser title. Default: `Demolab`.
- **description**: a one-line tagline shown under the homepage title. Default: none (no tagline line at all).
- **author**: the lab’s owner. Renders as a byline (“by Your Name”) under the homepage title, and as the `<meta name="author">` tag on every web page. Default: none.
- **contact**: an email address or URL. If set, the byline links to it: an address containing `@` becomes a `mailto:` link, anything else a plain link. Ignored unless **author** is also set. Default: none.

So a personal lab’s byline is two lines:

```
author: Ada Lovelace
contact: ada@example.com
```

- **book-title**: the document title of the bound book, `pdfs/book.pdf`. Default: `Demolab – the book`.
- **contents-title**: the heading over the book’s table of contents. Default: `Demolab – contents`.

Collections

Every writing declares a **collection**: in its meta, and the homepage lists one row per collection, linking to a page of its entries. A collection needs no registration to work: an unregistered slug is simply title-cased for display (`neuron-models` becomes “Neuron Models”). The **collections:** map in `demolab.yaml` upgrades that: per slug, a **label** (the display name) and a **description** (shown under the label on the homepage and at the top of the collection’s own page).

collection-order: is a list of slugs setting the homepage order. Collections you leave out of the list still appear; they trail after the listed ones. Two special cases:

- A writing with **no collection**: in its meta lands in a collection called `uncategorized` (displayed “Uncategorized”).
- A slide deck with **no collection**: lands in `slides`.

Curated ordering

Within a collection page, entries normally list newest first (settled work leads: final entries before drafts, then newest id first). For a collection that reads in a deliberate sequence, like this

documentation, you can curate the order instead. This is not a `demolab.yaml` key; it lives in each writing's meta:

```
#let meta = (  
  title: "Getting started",  
  // ...  
  order: 1,  
)
```

`order`: is an integer rank. The moment **any** entry in a collection carries one, that whole collection becomes curated: its page lists entries by ascending `order`, and unranked entries trail in id order. Rank all of a curated collection's entries, not a few; a half-ranked collection reads as an accident.

A worked example

The `demolab.yaml` behind this documentation site is:

```
name: Demolab  
book-title: Demolab – the book  
contents-title: Demolab – contents  
description: A lab notebook for computational science – reproducible results,  
published and citable.  
  
collection-order: [documentation, slides]  
collections:  
  documentation:  
    label: Documentation  
    description: "How demolab works, in reading order: what it is and why, setting  
up, the contract, the mechanics of experiments and writeups, and running the lab day  
to day."  
  slides:  
    label: Talks & slides  
    description: Deck PDFs from talks – paged-only, linked as PDF.
```

Delete a key and its default takes over; add a collection when a new theme emerges. The config is small on purpose: branding, grouping, order, and nothing that could contradict what the runs actually produced.

Using another language

11 July 2026

demolab ships Python: the skeleton lays down a uv environment, plotting helpers, and a provenance stamp, and every worked reference is Python. But Python is the opinionated example, not a requirement. The contract that connects a tool to an experiment is file-based and language-neutral, so your science can stay in whatever language it already lives in.

Why any language works

A tool is never imported — it is reached by **running its command line** (a subprocess), and it communicates only through files: `config.json`, `output.json`, `manifest.json`, a `run.sh` reproducer, and its figure data (`.csv`, `.json`, `.npz` — your choice of format). Any executable that parses arguments and writes those files is a valid tool. The write-up reads `numbers.json`, and Typst typesets from that plus the rendered figures — neither cares what produced them. So the only language-bound layer is the tool itself. See The contract for the full boundary.

Two ways in

Pick the smallest change that gets your language into the lab.

1. **Hybrid — tools in your language, Python stays the glue.** Write `tools/<tool>/tool.<ext>` in MATLAB, Julia, R, or another language; the runner still shells out to it and keeps a minimal uv environment purely to stage figures and plot. Least churn, and the recommended path — you rewrite only the science.
2. **Full switch — tool and runner in your language.** Rewrite the runner too, so it renders its own figures and writes `numbers.json`, then run that runner on the new runtime and drop the Python dependencies. Julia is the natural fit here: it can own the tool, the runner, and the plotting. Take this only if you want Python out entirely.

What never changes

Either path leaves the rest of demolab untouched: the engine (Typst, inside the `demolab-cli` package), your `writings/*.typ` (they read `numbers.json` exactly as before), `artifacts/`, and the `numbers.json` schema. Provenance still holds — you reimplement the stamp (git commit, dirty flag, UTC timestamp) in the new language’s run-setup helper.

What your language needs

Three primitives, which every mainstream scientific language has: run non-interactively from a command line, write JSON for the contract files, and write figure data (a CSV is enough) — plus a unit-test runner. MATLAB (`matlab -batch`), Octave, Julia (`JSON3`, `CSV`), and R (`jsonlite`, `write.csv`) all qualify.

Ask your agent

Say *MIGRATE-STACK* (or “use Julia instead of Python”) and your agent follows the *MIGRATE-STACK* runbook: it confirms the runtime, ports the tool contract into your language, wires the runner entry point and `demolab test` to it, and offers the hybrid path first.