

Anatomy of an experiment

ar023 · 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.