```
"""Authors
-------
Dylan Madisetti*  <dylan@marimo.io>    ORCID 0000-0003-1269-4989
Myles Scolnick    <myles@marimo.io>    ORCID 0009-0003-4254-9632
Akshay Agrawal    <akshay@marimo.io>   ORCID 0009-0004-6472-0055

* corresponding
Affiliation: marimo, Coreweave
"""
import marimo as mo
import hashlib
```

7ms

setup cell

Hash all the things 🌿

Your notebook kernel just died, and you lost all your data. *Now what?*

Interactive notebooks are key for research, visualization, and education- but expensive cell re-execution and lost state after restarts remain painful. This poster presents a caching approach that makes notebook restarts instant, reliable and easily shared as HTML interactives.

```
mo.vstack([
```

The problem

Notebooks underpin scientific Python, but most do not survive a clean rerun.

of 1.4M public Jupyter notebooks
~25%
re-execute top-to-bottom without error

of the same corpus
~4%
reproduce the original results on rerun

> **The opportunity**
> marimo notebooks are notebook deterministic dataflow graphs. Expensive recomputation can be skipped if state can be shown to be unchanged.

`problem`

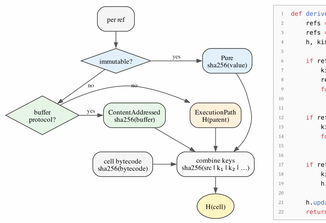
The idea

A stable cache key must depend on *all of the things* that could affect a cell's output. Hashing only raw input breaks (realloc moves pointers, weakrefs, opaque C objects); hashing source alone misses side effects. Computing a key requires a cascade:

- **(a)** the *compiled cell body* (bytecode — comments & formatting don't count),
- **(b)** a *content address* for every reference directly addressable (primitives, ndarrays, etc.)
- **(c)** the *keys of the parent cells* that own the references it can't.

Key dispatch

Branch	When	How it hashes
Pure / Stateful	primitives, UI state, `mo.state`	hash the value directly
ContentAddressed	buffer-protocol objects (ndarray, Arrow)	hash the **raw buffer**
ExecutionPath	cell-owned, non-addressable	substitute the **producer's** $H(c)$




```
def derive_key(cell, graph, scope) -> HashKey:
    refs = ScopedVisitor(cell.module).refs          # static analysis
    refs = normalize_scope(refs, scope)             # UI/state -> values
    h, kind = sha256(), "Pure"                      # no refs: pure cell

    if refs:                                        # content-address first
        kind = "ContentAddressed"
        refs, serial = content_serialize(refs, scope)
        for ref in sorted(serial):
            h.update(serial[ref])                   # buffer-protocol view

    if refs:                                        # substitute producers
        kind = "ExecutionPath"
        for parent in producing_cells(refs, graph):
            h.update(derive_key(parent, graph, scope).digest)

    if refs - scoped_refs:                          # same-cell context
        kind = "ContextExecutionPath"
        h.update(hash_module(cell.context))

    h.update(hash_module(compile(cell.module)))     # bytecode, not source
    return HashKey(h.digest(), kind)
```

Per-ref dispatch (left) feeds one sha256 combiner with the cell's compiled-body hash to produce H(c); the recurrence over the full cell is abridged from `BlockHasher.__init__`.

References

1. Pimentel et al. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. *MSR*, 2019.
2. Agrawal & Scolnick. marimo: An Open-Source Reactive Notebook for Python. 2023.
3. Makelov. Mandala: Compositional Memoization for Scientific Data Management. *SciPy*, 2024.
4. Jenks. DiskCache: Disk and File Backed Cache for Python. 2016.
5. Joblib Developers. Joblib: Running Python Functions as Pipeline Jobs. 2024.
6. Merkle. A Digital Signature Based on a Conventional Encryption Function. *CRYPTO '87*, 1988.
7. Michie. "Memo" Functions and Machine Learning. *Nature*, 1968.
8. Guo & Engler. Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. *ISSTA*, 2011.
9. Li et al. Kishu: Time-Traveling for Computational Notebooks. *VLDB*, 2025.
10. Dolstra. The Purely Functional Software Deployment Model (Nix). PhD thesis, 2006.

In practice

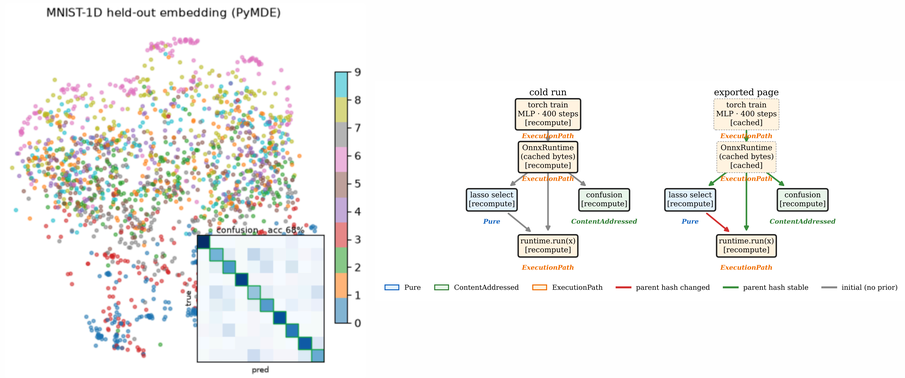
The following examines two concrete examples of how this caching approach pays off in real notebooks. Through caching, both of these notebooks are **portable all the way to the browser**, and retain their interactivity. The notebook targets 2 real usecases: (1) restoring an expensive state, and (2) pre-computing a state space for sharing or exploration.

Boxes below are filled by dispatch branch (Pure, ContentAddressed, ExecutionPath); a **dashed** border marks a cell restored from cache, a **solid** one runs live.

> **New in marimo `0.24.14`**
> `marimo export html-wasm --execute` bundles the blobs into static HTML with caching enabled.

1 · Restore expensive state

In this notebook we train a simple 1D MNIST torch classifier and cache the model weights. While which could be easily done by explicitly saving the model to disk, with cached runtime it's just implicit. A fun consequence is that we can run this expensive notebook in browser. **pytorch does not have an emscripten target**, but cache restoration leverages OnnxRuntime-WASM.

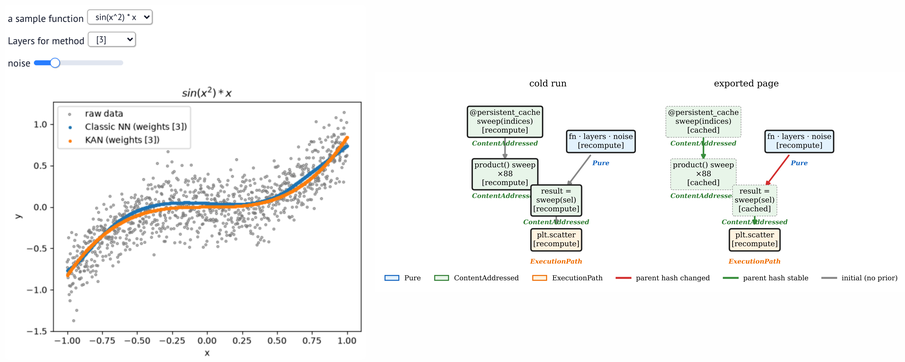


Left: live embed (interactive in the shared island export) — the PyMDE embedding of the held-out set with the region confusion matrix overlaid; both render from cached predictions, torch-free. **Right:** one persistent_cache block restores a live OnnxRuntime on a cache hit, so torch / pymde / mnist1d never import while the consumers re-run live.

Live demo — scan to run the exported WASM notebook in your browser; the model restores from cache with no torch.

2 · Precompute a state space

This notebook we demonstrate how caching unlocks easily sharing results. The full notebook precomputes a discrete state space of 88 combinations of function, layer scheme, and noise level. Caching the results allows the user to explore the full state space in a live embed, with every pick served from cache and no torch required (and share with collaborators!)

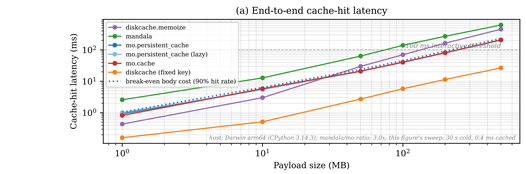


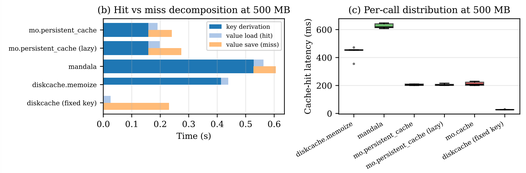
Left: live embed (interactive in the shared island export) — pick a function, layer scheme, and noise level; the fit scatter updates from a torch-free cache hit. **Right:** a product() sweep warms every entry of the discrete UI state space (88 blobs), so any pick is served from cache and only the plot re-runs.

Live demo — scan to explore the full precomputed sweep; every dropdown/slider pick is an instant torch-free cache hit.

Is it worth it?

Yes! Caching allows for portability in only a way an execution graph can provide. However, the concept of caching is not particular to marimo, the following examines alternative approaches to caching in Python scientific computing and compares their performance to marimo's. We time the full *edit-upstream to value-bound-in-Python* wall clock against the two directly comparable systems: **mandala** (SciPy '24; keys via `joblib.hash`) and **diskcache** (byte-keyed, the no-derivation floor) using numpy `float64` payloads from **1 MB to 500 MB**.





(a) hit latency vs payload with the 100 ms interactive line and a 90%-hit break-even curve; (b) hit (key+load) vs miss (key+save) at 500 MB; (c) per-call spread. `diskcache.memoize` fails past its SQLite blob ceiling.

```
mo.hstack([
```

interactive threshold
< 100 ms
held across exploratory payloads

cache-hit latency
≲ mandala
at every measured size

`results`

- Every persistent method holds the **100 ms** interactive threshold up to ~49 MB payloads.
- marimo's hit is **within noise of mandala** at every size and **strictly faster at 500 MB**: it hashes the `ndarray`'s contiguous buffer directly, while mandala pickles first (≈**3×** end-to-end on an M4 Max, ≈**1.2×** on a Linux x86-64 server).
- Value load tracks the **diskcache floor** (diskcache can optionally **not** provide a key derivation).
- `diskcache.memoize` **fails outright past its SQLite blob ceiling**.
- At a 90% hit rate, caching pays once the cell body costs more than ≈10% above the hit curve.

> **Limitations**
>
> - Library versions and python versions must be consistent (this can be loosened with `pin_modules=False`)
> - Mutable refs that **bypass the DAG** (alias/closure mutation) can still poison downstream.
> - Unpickling is **code execution** — a poisoned cache is a real risk but marimo does leverage Ed25519-signed blobs to establish a chain of trust.
> - Side effects are folded in *explicitly* via handles: `mo.watch.file`, `mo.watch.directory`.

Takeaway 🌿

- **Compiled body + content-addressed refs + parent-cell hashes** -> a Merkle DAG
- caches invalidates at subtree granularity. Native to the reactive notebook.
- **Zero user effort**, competitive with scientific-Python memoizers — and
- **portable all the way to the browser.**

This poster is a live marimo export.

this poster live

github/marimo-team