Open Access

SciPy 2026

July 13 - July 19, 2026

Proceedings of the 25th Python in Science Conference
ISSN: 2575-9752

Hash all the things: Caching for fast notebook restarts

Dylan Madisetti[1], Myles Scolnick[1], and Akshay Agrawal[1]

[1]marimo

Abstract

We describe a caching mechanism that lets reactive notebooks restart without re-running their expensive cells. The mechanism is built into marimo, a reactive Python notebook that models the notebook as a dataflow graph. Each cell's cached result is identified by a key built from fingerprints (hashes) of the cell's code and inputs: an input whose bytes are accessible is hashed directly, and any other input is represented by the key of the cell that produced it, computed the same way. Because each cell's key folds in the keys of the cells it depends on, editing one cell invalidates the cached results of exactly the cells downstream of it. Cached values are stored on disk and loaded only when accessed, so they can be reused across independent runs of the same notebook. They are also bundled into marimo's static export, a standalone web page (HTML) that runs the notebook through WebAssembly (WASM), so readers whose only Python runtime is a browser can open a notebook with its expensive results and trained models already computed. In microbenchmarks over payloads of varying size, a marimo cache hit is comparable to widely used Python caching libraries, with a speedup on certain hardware, while asking almost nothing of the user.

Keywords reactive notebooks, content-addressed caching, memoization, reproducibility, dataflow, marimo, Python

1. Introduction

Notebooks underpin much of scientific Python, but most notebooks cannot be re-executed from scratch. In a survey of 1.4 million public Jupyter notebooks, only about a quarter re-executed top to bottom without raising an error [1]. Traditional notebooks like Jupyter, built on the imperative `ipykernel`, are essentially read-evaluate-print loops (REPLs), in which each cell execution mutates a shared global state. As a result, the notebook accumulates hidden state, and the outputs saved in the notebook file can differ from the outputs that a top-to-bottom run would produce.

Reactive notebooks close this gap by treating each cell as a node in a dataflow graph. The notebook determines, for each cell, which variables it reads (its *references*, or `refs`) and which variables it defines (its *definitions*, or `defs`). From these, a reactive notebook derives a deterministic execution order based on data dependencies rather than on the order of cells on the page. Running a cell removes the cell's previous variable bindings from memory, updates the dataflow graph if the cell's code changed, and re-runs the cells that depend on it, which minimizes hidden state. Notable reactive notebooks include Pluto.jl [2], Observable [3], and Livebook [4]. Reactive notebooks descend from a longer tradition of direct-manipulation programming environments [5], in which editing the source is itself the act that updates the running program. marimo [6] is a reactive notebook for Python, and its caching mechanism is the subject of this paper.

To mitigate unnecessary re-runs of expensive cells, reactive notebooks offer runtime configuration, such as lazy executors that mark cells as stale instead of running them; however,

Published Aug 10, 2026

Correspondence to
Dylan Madisetti
dylan@marimo.io

these primitives require some intervention from the user. In this paper, we show how to exploit the deterministic execution order that reactive notebooks enforce to build a caching mechanism that, under the right conditions, automatically eliminates unnecessary recomputation of expensive cells.

Caching for notebooks and for Python is not new; we review prior systems in Background and Related Work. Each of them asks the user to opt in at some boundary, such as a decorated function, a document chunk, or a whole session. One benefit of specializing caching to reactive notebooks in particular is that they already draw boundaries around every cell, decreasing the cognitive overhead the user is subject to.

The caching mechanism we propose, and whose implementation we share, was designed to satisfy three properties.

- Skip expensive recomputation when a cell's references and source are unchanged.
- Preserve reactive determinism by reusing a result only while it stays valid and never serving a stale (false-positive) hit.
- Make cached artifacts transportable through marimo's static WASM/HTML export.

Out of scope are full session restoration in the sense of Kishu [7], distributed execution, and reproducibility of arbitrary Python notebooks [1]. Our contribution is deterministic reuse between potentially cross-platform notebook sessions that follow marimo's reactive principles.

2. Background and Related Work

Hashes. A *hash* is a short, fixed-size fingerprint computed from data; any change to the data yields a different fingerprint. With a cryptographic hash function, finding two different inputs with the same fingerprint is computationally infeasible, so matching fingerprints can be treated as matching data. A value is *content-addressed* when it is identified by a hash of its bytes: two values with the same bytes receive the same identity.

Memo functions. Michie's memo functions [8] are the canonical description of function caching: a cached function skips recomputation when a key derived from its inputs matches a stored key, and returns the stored value instead. To cache a *cell* the same way, the key must reconstruct identically under identical conditions. That requirement forces the central question of this paper: what are a cell's inputs?

A cell's inputs. A reactive notebook gives two answers. Statically, it derives a dependency graph from the source, and that graph is stable across runs. At runtime, it knows the values currently bound in memory. The cache key must therefore be a function of the graph the source defines and of the values present when the cell runs, not of any particular execution trace. Cell-level dataflow tracking has an earlier antecedent in [9].

Runtime-tracing for reactive notebooks. IPyflow and nbsafety [10], [11] take a different route to reactivity: they retrofit it onto Jupyter by tracing execution to record which variables each cell reads and writes, and then flagging or re-running cells whose inputs have become stale. Runtime tracing has drawbacks, however: the traced graph reflects only the executions it has observed, so it can differ from session to session on the same notebook. A graph derived from the source code, like marimo's, avoids both problems.

Build systems. Our key construction borrows from build systems. Build Systems à la Carte [12] decomposes a build system into a *rebuilder*, which decides when to re-run a task, and a *scheduler*, which decides the order. In those terms, marimo's cache is a rebuilder that decides by comparing keys hashed from a cell's code and inputs (Cache Keys), paired with marimo's existing reactive scheduler, and it keeps no persistent record of past builds. The

recursive key construction comes from the Nix package manager [13], [14]. Nix identifies each package by a hash of all of its inputs, and those inputs include the hashes of the packages they were built from, so one package's identity recursively covers its entire dependency tree. We apply the same construction to a reactive notebook instead of a static package graph. Data-engineering systems apply the same discipline at coarser granularity: Bauplan and Nessie hash pipeline stages [15], and the workflow engines Nextflow (`-resume`) and Snakemake (`--cache`) key task results on code, parameters, and input hashes [16], [17].

Caching for Python and notebooks. Caching tools for Python and notebooks draw their opt-in boundaries at different places. IncPy modifies CPython, the standard Python interpreter, to memoize function calls automatically [18]. knitr caches chunks of literate documents, with dependencies declared by hand [19]. jupyter-cache re-executes a notebook wholesale when any code cell changes [20]. Streamlit asks the user to choose between two caching decorators, `cache_data` for values that can be identified by their content and `cache_resource` for values that can only be identified by what produced them [21]; marimo's key construction makes this choice automatically (Section 1).

Memoizers for scientific Python. mandala [22] is the closest analog to our mechanism. It memoizes calls inside a `with storage:` context, computes content addresses with `joblib.hash`, and records which calls produced which values. diskcache [23] stores values under raw byte keys that the caller must construct themselves, so it measures the cost of storage alone and serves as a control in our measurements. joblib's `Memory` [24], the standard persistent memoizer in scientific Python, keys on the pickled arguments of each decorated function. Kishu [7] and ElasticNotebook [25] checkpoint and migrate notebook state; they complement a cache rather than compete with one. Only mandala and diskcache are directly comparable to marimo's mechanism, and we benchmark against both in Evaluation.

3. Cache Keys

In computational caching, a false positive — restoring a value that the code would not have produced — is unacceptable. A false negative — failing to find a stored value and recomputing it — merely wastes time, and is usually acceptable as the user can understand the criteria for a value to be cached. The key derivation must therefore change whenever a value the cell reads changes, and it should not change under superficial edits such as reformatting or comment changes.

Consider two obvious ways to derive a key. The first is to hash the value of every reference the cell reads. A marimo notebook could attempt this, since at runtime it exposes both the dataflow graph and the reference values bound in memory. This derivation fails because the key cannot always be computed: some Python values expose nothing stable to hash. An object's memory address is neither stable nor meaningful as an identity; weak references report which object they point to, not what it contains; opaque C-extension objects expose neither their underlying bytes nor a stable text representation. The second derivation is to hash the cell's source bytes alone. This fails in the opposite way: the key is always computable, but it does not change when the cell's inputs change, which produces exactly the false positives we ruled out above. It misses inputs that arrive through side effects such as the filesystem, the network, or the wall clock.

marimo combines the two derivations: a value that can be hashed is hashed, and a value that cannot is identified by the code that produced it. Each derivation covers the other's failure, since the first keeps the key sensitive to inputs and the second keeps the key computable. Build systems handle unhashable artifacts the same way: an artifact that exposes no content to hash is identified by the build step that produced it [13]. marimo applies this idea recursively over the dataflow graph. Section 1 gives the precise construction. Section 2

then examines invalidation: when a cell is edited, only the cached results of the cells downstream of it become invalid, and recomputing keys takes time proportional to the number of cells affected. Side effects, which neither derivation sees, remain only partially covered; we return to this limitation in Limitations and Discussion.

3.1. Constructing the key

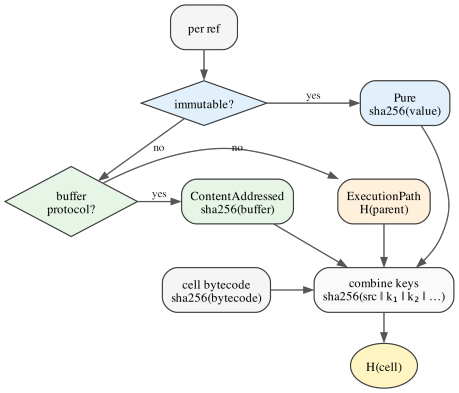
In this section, we describe how a cell's cache key is computed from two parts: its source code and its references. Throughout, we write $H(c)$ for the key of cell c.

3.1.1. *Hashing source code*

A cell's references identify what flows into it, and its code determines what it computes from those inputs. The code must therefore enter the key. Without it, two cells that read the same input, such as `y = x + 1` and `y = x - 1`, would share a key, and one could be served the other's cached result. Hashing the code also provides the most basic invalidation a user expects: editing a cell changes its key, so the cell's own stale result is never reused.

marimo hashes the compiled bytecode rather than the source text. Compilation discards exactly the edits that cannot change behavior, such as comments and formatting, so those edits do not invalidate the cache. The trade-off is that bytecode is specific to the Python version. Cached results therefore do not transfer across interpreter upgrades, and marimo's browser export (Section 3) requires the exporting interpreter to match the browser's Python version.

3.1.2. *Hashing references*




```python
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)
```

Figure 1. *Hashing references. Left: each reference is tried against three strategies in order, and the first that applies yields that reference's hash. An immutable value is hashed directly (labeled `Pure`). A value that exposes its bytes through the buffer protocol has those bytes hashed (`ContentAddressed`). Any other value contributes the key of the cell that produced it (`ExecutionPath`). The reference hashes and the hash of the cell's compiled body are then combined into a single hash: the cell's key, $H(c)$. Right: the same derivation applied to a whole cell. The listing reuses these labels to classify the whole cell's key by the strongest fallback it needed, and adds `ContextExecutionPath`, a special case described in the text.*

Each reference is tried against the three strategies. The first strategy that applies yields the reference's hash; we call this decision the *key dispatch*. The strategies — direct hashing of immutable values, content addressing, and producer substitution — are described below, and the dispatch algorithm is illustrated in Figure 1.

Immutable values. An immutable value, such as a number, a string, or a frozen collection, is hashed directly. The figure labels this case `Pure`. Interactive inputs reach this case through one normalization step: before hashing, a reference to a user interface (UI) element such

as a slider is replaced by the input's current value (line 3 of the listing), so the key changes when the reader moves the slider.

Content-addressed values. A value that exposes its underlying bytes through Python's buffer protocol — the standard mechanism for exposing an object's raw bytes without copying — has those bytes hashed. The figure labels this case `ContentAddressed`. This case is important for scientific computing: it covers NumPy ndarrays and other objects advertising NumPy's array interface. The hash is computed from the contiguous buffer without serialization, an idiom borrowed from joblib [24] and mandala [22].

Producer substitution. Any other value exposes nothing stable to hash, but a known upstream cell produced it. Instead of hashing the value, marimo uses the producing cell's key as the reference's hash. The producing cell's key is built by this same construction, so it covers the producer's code, the producer's inputs, and, by the same rule, everything upstream of them. A change anywhere in the value's ancestry therefore changes the reference's hash. The figure labels this case `ExecutionPath`.

3.1.3. *Combining hashes*

All hashing in this construction uses SHA-256, as Figure 1 shows. The hashes of the code and of the references are combined into one in the combine step at the bottom of the figure. Every reference hash, taken in sorted reference-name order so that the result does not depend on the order in which references were processed, is fed into a single SHA-256 computation together with the bytecode hash. The resulting digest is the cell's key, $H(c)$. Because SHA-256 is a cryptographic hash function (Background and Related Work), two keys match only when the code and every reference contribution match, so key equality is a safe stand-in for "this cell would compute the same values."

3.1.4. *Caching blocks and functions*

The cached unit is not always a whole cell: it can be a block of code inside a cell, or a function (Using marimo's cache). A reference defined earlier in the same cell as a cached block has no parent cell whose key could stand in for it. Instead, the code surrounding the block is folded into the key, a special case of producer substitution that the implementation calls `ContextExecutionPath`. For a cached function, the same dispatch classifies the function's arguments at call time.

3.2. Invalidation

The construction of Section 1 is recursive: a cell's key can contain its producer's key, which can contain that producer's key in turn. In practice, no key is ever derived by walking the whole ancestry. When a cell c finishes executing, marimo records its key $H(c)$, and when a downstream cell's key later needs $H(c)$ for producer substitution, marimo reuses the recorded value. Each cell's key is therefore computed at most once per execution.

A cached result is *invalidated* when its cell's key changes: the new key no longer matches any stored entry, so the next lookup misses and the cell recomputes. Invalidation is not an action that marimo performs, and nothing is deleted. The old entry simply stops being found.

Cache Keys opened by ruling out false positives: the cache must never serve a value the code would not have produced. For invalidation, that means never missing a change: whenever re-running a cell could produce a different value, the cell's key must have changed. To keep false negatives rare, invalidation should also be limited: an edit should not invalidate results it cannot affect, and updating keys should be cheap. The two subsections below establish each property in turn.

3.2.1. *Invalidation never misses a change*

For a reference hashed by content, the property is immediate: if the bytes change, the hash changes, and so does every key built from it. The case that needs an argument is producer substitution, which judges a value by its origin rather than its content. Could the producing cell's key stay the same while the value it produced changes? In a reactive notebook, no. A value can change only if the cell that produced it re-runs, and a cell re-runs only when its own code or its own inputs change — exactly the ingredients of that cell's key. An unchanged producer key therefore implies an unchanged value, provided cell bodies are deterministic; side effects are the exception, and Limitations and Discussion discusses them. This is the argument behind the reactive determinism property of Introduction: a value served from the cache is the value that re-running the cell would produce. This argument is what requires a reactive notebook.

Caching never requires the value itself to be hashable, only that the producing cell have a key. The substitution is a special case of Hughes's lazy memo functions [26]: two values are treated as equal because they come from the same execution of the same code, not because their contents were compared.

3.2.2. *Invalidation is limited and cheap*

Wherever producer substitution occurs, a cell's key contains the keys of cells upstream of it. The notebook's keys therefore form a Merkle directed acyclic graph (DAG) [27], a structure in which each node's fingerprint depends on the fingerprints of the nodes it builds on. Git commits use the same construction. Two properties follow. First, editing a cell can change keys only in the cells downstream of the edit, so every other cached result in the notebook remains valid. Second, the change does not always reach everything downstream: where a re-run cell produces byte-identical values, its content-addressed consumers keep their old keys, and the propagation stops there. Recomputing keys after an edit takes time proportional to the number of cells whose keys change, because every other cell's recorded key is simply reused.

3.2.3. *A worked example*

Figure 2 traces the construction on the four-cell PyTorch graph written out below: a seed, a random tensor generated from the seed, a small neural network constructed independently, and a forward pass that applies the network to the tensor. Between them, the cells exercise all three strategies: the seed is hashed as an immutable value, the tensor is hashed through its buffer, and the network (`TinyNet`), which exposes no bytes to hash, is represented by the key of the cell that constructed it. The italic label under each cell in the figure classifies that cell's own key, as computed by marimo's hasher: `a` is `Pure` (no references), `b` and `c` are `ContentAddressed` (every reference hashed by value), and `d` is `ExecutionPath` (one reference substituted a producer's key).

```
# a · seed          # b · random input              # c · model              # d · forward pass
seed = 7             x = rng(seed).normal(64)       model = TinyNet()        y = model(x)
```

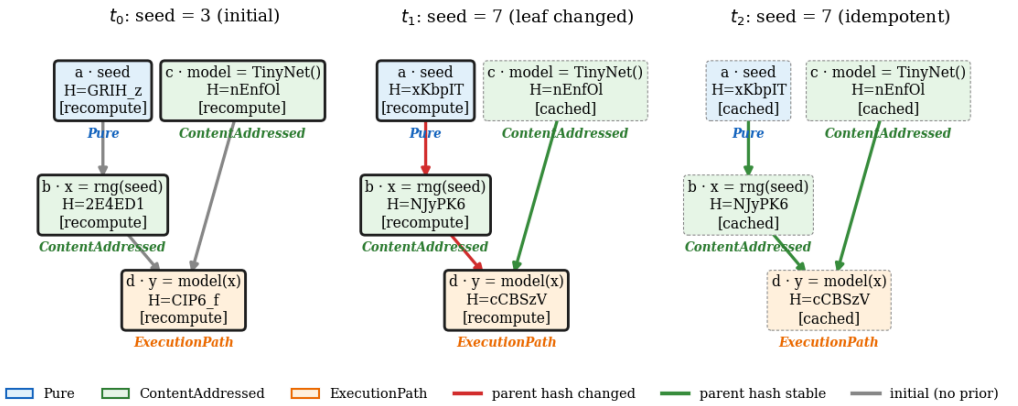


Figure 2. *A worked example of the recurrence on the four-cell graph written out above. Every hash and branch label in the figure is produced by marimo's real hasher on a compiled cell graph at render time. Changing the seed ($t_0 \to t_1$) invalidates* a, b, *and* d *(red edges);* c *stays cached because the seed is not among its references. Re-rendering with the same seed ($t_1 \to t_2$) leaves every hash unchanged (green edges), so the rebuilder reuses every result. The italic label under each box names the dispatch branch that cell exercises.*

4. Storage and Loading

On a cache hit, marimo must restore the variables the cached cell would have defined. Restoring has two parts: *lookup* finds the stored entry whose key matches $H(c)$, and *loading* deserializes the entry's values into memory. The two parts need not happen together, because the notebook does not always need a value's bytes. For example, a downstream cell may take a variable and pass it to a third cell without inspecting it. marimo therefore lets loading lag lookup, and offers two loaders that differ in how far.

The default loader, `PickleLoader`, does not lag at all. It writes the full `Cache` envelope, the record holding every variable the cell defined, as a single blob using pickle, Python's built-in serializer. On lookup, it loads every variable back immediately.

A second loader, `LazyLoader`, writes a JSON (JavaScript Object Notation) manifest listing each variable, alongside one blob file per value. The manifest records, for each variable, either the value itself (small primitives are stored inline) or the name of the blob file holding it. Its `cache_type` field records which strategy of Section 1 produced the key. Abridged, a manifest for a cell that defined a seed and a large array might read:

```
{
  "hash": "9V5v6Cji…",
  "cache_type": "ContentAddressed",
  "defs": {
    "seed": {"primitive": 7},
    "x": {"reference": "blob-3fa9…"}
  },
  "stateful_refs": [],
  "meta": {"version": 4, "blob_hashes": {"blob-3fa9…": "e3b0c4…"}}
}
```

On lookup, `LazyLoader` binds each variable not to its value but to a *stub*: a small placeholder object that records where the value's bytes live and how to deserialize them. All stubs share one interface with a single method, `load()`, and one stub type exists per storage format:

pickle, joblib, NumPy `.npy`, and Apache Arrow. The first time the notebook uses the variable, the stub loads the value. A variable that is never used is never loaded.

4.1. WASM portability

Cached values also work in marimo's static WASM export: a standalone HTML file that runs the notebook in the reader's browser, with no server, on Pyodide, a Python runtime compiled to WebAssembly. Exporting a notebook through marimo's command-line interface (`marimo export html-wasm --execute`) bundles the cache manifests and blobs into that file. When a reader opens it, the browser session derives the same keys the exporting machine derived, so every lookup hits, and each value loads on first use exactly as in an interactive session. Scientific articles (including this one), blog posts, and educational materials can therefore include precomputed results and trained models with the notebook.

The export contains only the cached values and the user code that produced them, not external libraries. A cell may therefore depend on packages the browser cannot import, as long as the values it defines are stored in a portable format. Our demonstration, published at https://dmadisetti.github.io/scipy_proceedings/demo/, exercises exactly this case. On the exporting machine, the notebook trains a PyTorch model, converts it to ONNX (Open Neural Network Exchange) bytes, defines a small wrapper class, and stores a sweep of predictions as NumPy arrays. In the browser, where Pyodide cannot currently import PyTorch, the arrays and the wrapper load through their format stubs, and a custom stub feeds the ONNX bytes to `onnxruntime-web`, an ONNX runtime for the browser, restoring a model the notebook can call.

5. Using marimo's cache

marimo provides users three primary mechanisms to use the cache: in-memory caching for use within a session, persistent caching for reuse across sessions, and a mode that caches every cell automatically. None of the three asks the user to declare dependencies or construct keys. Every key is built as described in Cache Keys, so invalidation follows from the notebook's dataflow graph, including changes to UI elements and `mo.state`.

5.1. In-memory caching

The decorator `@mo.cache` memoizes a function. Each call is keyed, through the dispatch of Section 1, on the function's code, its arguments, and the variables it uses from outside its own body. Results stay in the kernel's memory, so a hit reads nothing from disk, but all results are lost when the session ends. This form suits values that are cheap to hold in memory but wasteful to recompute, such as the result of a function that is called again every time a slider moves. `@mo.cache` keeps every result; the variant `@mo.lru_cache` keeps a bounded number, 128 by default, evicting the least recently used when full.

Comparison to **`functools`.** Python's built-in `functools.cache` is not well-suited to reactive notebooks. A reactive runtime re-runs a function's defining cell whenever its code or inputs change, recreating the function with an empty `functools` cache. Every stored result is therefore thrown away on every re-run of the defining cell, even when the change did not affect the function's behavior.

5.2. Persistent caching

The decorator `@mo.persistent_cache` memoizes a function the same way, with the same keys, but writes results to disk through the loaders of Storage and Loading, so they survive kernel restarts. This is the form that lets a notebook restart without re-running its expensive cells, and the files it writes are what the WASM export of Section 3 bundles. Used as a context

manager, `with mo.persistent_cache(name="training"):` caches a block of code inside a cell. On a hit, the block is not executed at all: its variables are restored from disk, and any side effects the block would have had do not happen. Optional arguments choose the storage directory (`save_path`) and the loader (`method="pickle"` or `method="lazy"`). A third option, `pin_modules`, opts library versions into the key.

5.3. Automatic caching of every cell

The decorators and the context manager cache only what the author wraps. marimo can also cache every cell automatically. In this execution mode, the runtime computes each cell's hash before running it and checks the cache: on a hit it skips the cell's body and restores its variables, and on a miss it runs the cell and saves the results for next time. A stored entry occasionally cannot be used: if one of its variables could not be serialized, the entry holds only a placeholder, and the cell re-runs along with any upstream cells needed to rebuild the missing value. Turning this mode on (the `cache_cells` runtime option) yields the automatic, cell-granular caching that Introduction argued reactive notebooks make possible: the whole notebook is cached, and the author marks nothing.

6. Evaluation

Caching does not always save time. Every hit pays a fixed overhead: deriving the key, then loading the value. When the goal is portability, or a record of where results came from, rather than speed, that overhead does not matter. When the goal is to skip expensive recomputation, the overhead must be smaller than the cell body it avoids.

We validate the implementation by measuring three cost components — key derivation, value load, and value save — both separately and as end-to-end hit and miss paths. Payloads are NumPy `float64` arrays from 1 MB to 500 MB. We compare six strategies: `mo.cache`, `mo.persistent_cache` with each of its two loaders, mandala's decorated-function memoization, and diskcache in two forms, a memoizing decorator and a plain store with a fixed key. The fixed-key form performs no key derivation at all, so it is a lower bound on hit time. Each cost is measured over 10 runs after priming and a warmup, and we report the median. Every persisted measurement is keyed on a host fingerprint (operating system, architecture, and Python version) and a methodology-version string, so no result is reused across hosts or after the harness changes.

6.1. End-to-end cache evaluation

A notebook user experiences the cache as a single delay: the time from editing one cell to the moment a downstream cell's value is available in Python again. Panel (a) of Figure 3 reports that end-to-end time for six strategies. All six cluster together up to roughly 49 MB. The dashed line at 100 ms marks the threshold below which a response reads as instantaneous [28], and every persistent method stays under it across typical exploratory payload sizes.

The dotted curve in panel (a) shows when caching pays off. With hit rate p, caching saves time when the cell body costs more than $T_{\text{hit}} + \frac{1-p}{p}\left(T_{\text{key}} + T_{\text{save}}\right)$. On these payloads the measured miss overhead is comparable to the hit cost, so at $p = 0.9$ the break-even body cost is roughly 10% above the hit curve.

Panel (b) decomposes the largest-payload hit into key derivation and value load, next to the overhead a miss adds (key derivation and value save). mandala derives its key with `joblib.hash`, which serializes the value through pickle before hashing it [22]. marimo hashes the array's contiguous bytes directly through Python's buffer protocol, the content addressing of Section 1, with no serialization step. On the Apple M4 Max used for the camera-ready figures, hashing is fast, and the extra pickle pass costs mandala roughly 3× end to end.

On a Linux x86-64 server, where memory copies are cheap relative to hashing, the pickle pass costs little, value load dominates instead, and the penalty shrinks to roughly 1.2×. marimo's value-load time matches that of the fixed-key diskcache form, which performs no key derivation, confirming that the gap comes from key derivation rather than from storage. Panel (c) exposes per-call variance; it also shows `diskcache.memoize` failing once a payload exceeds its SQLite blob-size ceiling, so its largest sizes drop out of the sweep.

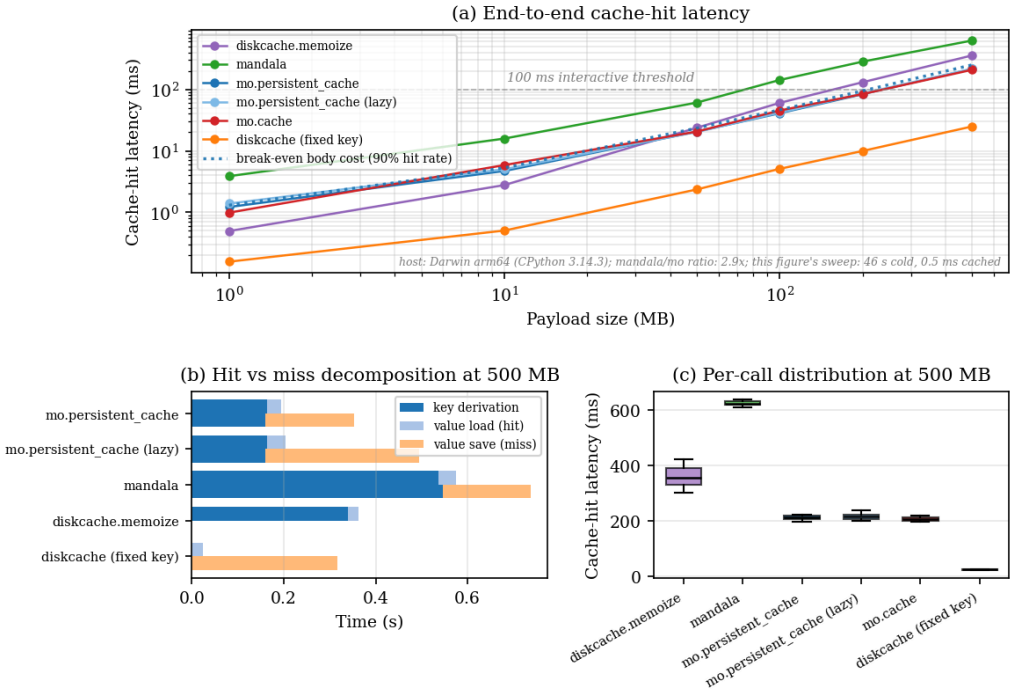


Figure 3. *End-to-end cache evaluation on NumPy `float64` payloads. (a) Cache-hit latency versus payload size, on log-log axes. The dashed line at 100 ms marks the interactive threshold [28], and the dotted curve is the break-even body cost at a 90% hit rate. (b) Decomposition of a hit (key derivation plus value load) and a miss (key derivation plus value save) at the largest sweep size, measured on the real disk-backed paths. (c) Per-method distribution of cache-hit samples at the largest size; `diskcache.memoize` drops out past its blob ceiling. The host label reports the build host, the mandala-to-marimo ratio, and the cold-versus-cached cost of the figure's own sweep.*

The camera-ready version of this paper was evaluated on a MacBook Pro with an Apple M4 Max. The measurements are produced by the paper's own build, and the figure's host label reports the measured mandala-to-marimo ratio, so a reproduction on different hardware shows its own number. The stage-decomposition measurement times the real disk-backed paths on both sides: `PickleLoader.load_cache` and `LazyLoader.load_cache` for marimo, and `joblib.load` on a temp-file blob for mandala. The load comparison therefore includes the cost of reading through the operating system's file cache and of reconstructing the cache envelope, both of which a bare `pickle.loads` would skip.

7. Limitations and Discussion

The most salient limitations follow.

Library versions. By default, the key does not take library versions into account; the user must explicitly opt in (Using marimo's cache). Upgrading a package can therefore serve stale hits. The portable WASM export must accept this gap, because pinning would bind keys to the export host.

Python versions. Because code is hashed as bytecode, keys are specific to the Python version (Section 1). Upgrading the interpreter therefore invalidates every cached result. Unlike a library upgrade, this failure is safe: the keys miss and the cells recompute. It is also why the browser export requires the exporting interpreter to match the browser's Python version (Section 3).

Mutation outside the graph. Mutations that bypass the dataflow graph — aliased mutation through a closure, or attribute writes on an object that cannot be content-addressed — can change a value without changing any key, so downstream cells can be served stale results. A suite of checks that runs on every build of this paper measures exactly this class of undetectable false positive, and Rex [29] probes the same boundary.

Cache tampering. Unpickling can execute arbitrary code, so loading from a cache that an attacker has modified could run a malicious payload. For the lazy loader, marimo mitigates this by signing caches: each manifest carries an Ed25519 signature that also covers the SHA-256 hash of every blob, and signatures are checked before any bytes are deserialized. The pickle loader has no such protection, and its caches should be loaded only from trusted sources.

Side effects. The cache is mostly blind to side effects such as file reads, network calls, and wall-clock queries. marimo does provide a mechanism for tracking a side effect explicitly: the side effect is wrapped in a handle whose value is folded into the cell's hash as if it were an external reference. Two such handles exist today, `mo.watch.file` and `mo.watch.directory`, keyed on file content and directory listing respectively. Randomness and wall-clock time could bind to similar lifetime-managed handles.

None of these limitations is fundamental to the approach. Future work can address each one by adding branches to the key dispatch.

8. Future work

Our evaluation and limitations point to four directions.

Cost-aware policy. Section 7 measured what a cache hit costs and what a cache miss adds. With those numbers available at runtime, marimo could apply the break-even rule automatically: decline to cache a cell whose body runs faster than the cache's own overhead, and report the time each cached cell actually saved.

Expanded side effects. Limitations and Discussion described handles that fold a side effect's value into a cell's key. Handles for randomness (`mo.random`), network requests (`mo.request`), and time (`mo.clock`) would be straightforward additions. Whether tracking more side effects justifies the added interface surface remains an open question.

Richer storage formats. The lazy loader stores each value with a format-specific stub (Storage and Loading). A format for PyTorch tensors (`.pt`) and one for Apache Arrow tables (`pyarrow.Table`) are natural next steps. Cached cell results could also be reused by agentic workflows that drive long-running notebook sessions [30].

Chain of trust for exports. The signing described in Limitations and Discussion protects a cache against tampering, but a shared or exported cache also requires the reader to know which public key to trust. Establishing that chain of trust, from the machine that produced an export to the reader's browser session, remains open.

Two further questions remain open: a storage policy (eviction, per-codec footprint), and cross-session memoization aligned with recorded provenance [31].

9. Conclusion

We have presented a caching mechanism for marimo, a reactive notebook for Python. The cache key for a cell is built from the cell's compiled body and the content-addressed values of its references, falling back to the producing cell's hash when a reference cannot be addressed by content. This construction preserves reactive determinism, survives superficial edits, and supports portable exports. Our evaluation shows that a marimo cache hit performs comparably to existing scientific-Python memoizers while requiring no annotations from the user. Finally, the cache participates in marimo's static HTML/Pyodide WASM export, so users can share notebooks with precomputed results and models, as the export published at https://dmadisetti.github.io/scipy_proceedings/demo/ demonstrates.

9.1. Acknowledgments

Portions of this work were assisted by a generative AI tool (Claude, Anthropic). Claude was used to help develop and run the benchmark harness reported in the Evaluation. All benchmark code and results were reviewed, verified, and revised by the authors, who take full responsibility for the accuracy and integrity of the final content.

References

[1] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks," in *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*, 2019, pp. 507–517. doi: 10.1109/MSR.2019.00077.

[2] F. van der Plas and Pluto.jl contributors, "Pluto.jl: Simple Reactive Notebooks for Julia." [Online]. Available: https://github.com/fonsp/Pluto.jl

[3] M. Bostock, "A Better Way to Code." [Online]. Available: https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0

[4] J. Valim and Livebook Team, "Livebook: Interactive and Collaborative Code Notebooks for Elixir." [Online]. Available: https://livebook.dev/

[5] B. Victor, "Inventing on Principle." [Online]. Available: https://worrydream.com/InventingOnPrinciple/

[6] A. Agrawal and M. Scolnick, "marimo: An Open-Source Reactive Notebook for Python." [Online]. Available: https://marimo.io/

[7] Z. Li, S. Chockchowwat, R. Sahu, A. Sheth, and Y. Park, "Kishu: Time-Traveling for Computational Notebooks," *Proceedings of the VLDB Endowment*, vol. 18, no. 4, pp. 970–985, 2025, doi: 10.14778/3717755.3717759.

[8] D. Michie, "``Memo'' Functions and Machine Learning," *Nature*, vol. 218, no. 5136, pp. 19–22, 1968, doi: 10.1038/218019a0.

[9] D. Koop and J. Patel, "Dataflow Notebooks: Encoding and Tracking Dependencies of Cells," in *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, Seattle, WA, 2017. [Online]. Available: https://www.usenix.org/conference/tapp17/workshop-program/presentation/koop

[10] S. Macke, H. Gong, D. J.-L. Lee, A. Head, D. Xin, and A. Parameswaran, "Fine-Grained Lineage for Safer Notebook Interactions," *Proceedings of the VLDB Endowment*, vol. 14, no. 6, pp. 1093–1101, 2021, doi: 10.14778/3447689.3447712.

[11] S. Macke, "IPyflow: A Next-Generation, Dataflow-Aware IPython Kernel." [Online]. Available: https://github.com/ipyflow/ipyflow

[12] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build Systems à la Carte," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 1–29, 2018, doi: 10.1145/3236774.

[13] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A Safe and Policy-Free System for Software Deployment," in *Proceedings of the 18th USENIX Conference on System Administration (LISA '04)*, Atlanta, GA, 2004, pp. 79–92. [Online]. Available: https://www.usenix.org/legacy/event/lisa04/tech/dolstra.html

[14] E. Dolstra, "The Purely Functional Software Deployment Model." [Online]. Available: https://edolstra.github.io/pubs/phd-thesis.pdf

[15] C. Greco and J. Tagliabue, "Reproducible Data Science over Data Lakes: Replayable Data Pipelines with Bauplan and Nessie," in *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning (DEEM@SIGMOD '24)*, 2024. doi: 10.1145/3650203.3663336.

[16] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, 2017, doi: 10.1038/nbt.3820.

[17] F. Mölder *et al.*, "Sustainable data analysis with Snakemake," *F1000Research*, vol. 10, p. 33, 2021, doi: 10.12688/f1000research.29032.2.

[18] P. J. Guo and D. Engler, "Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*, Toronto, ON, Canada, 2011, pp. 287–297. doi: 10.1145/2001420.2001455.

[19] Y. Xie, *Dynamic Documents with R and knitr*, 2nd ed. Boca Raton, FL: Chapman & Hall/CRC, 2015. doi: 10.1201/9781315382487.

[20] Executable Books Project, "jupyter-cache: A defined interface for working with a cache of executed Jupyter notebooks." [Online]. Available: https://github.com/executablebooks/jupyter-cache

[21] Streamlit Team, "Introducing Two New Caching Commands to Replace st.cache." [Online]. Available: https://blog.streamlit.io/introducing-two-new-caching-commands-to-replace-st-cache/

[22] A. Makelov, "Mandala: Compositional Memoization for Simple & Powerful Scientific Data Management," in *Proceedings of the 23rd Python in Science Conference (SciPy 2024)*, 2024. doi: 10.25080/JHPV7385.

[23] G. Jenks, "DiskCache: Disk and File Backed Cache for Python." [Online]. Available: https://github.com/grantjenks/python-diskcache

[24] Joblib Developers, "Joblib: Running Python Functions as Pipeline Jobs." [Online]. Available: https://joblib.readthedocs.io/

[25] Z. Li, P. Gor, R. Prabhu, H. Yu, Y. Mao, and Y. Park, "ElasticNotebook: Enabling Live Migration for Computational Notebooks," *Proceedings of the VLDB Endowment*, vol. 17, no. 2, pp. 119–133, 2024, doi: 10.14778/3626292.3626296.

[26] R. J. M. Hughes, "Lazy Memo-Functions," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '85)*, in LNCS, vol. 201. 1985, pp. 129–146. doi: 10.1007/3-540-15975-4_34.

[27] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology — CRYPTO '87*, in Lecture Notes in Computer Science, vol. 293. 1988, pp. 369–378. doi: 10.1007/3-540-48184-2_32.

[28] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The Information Visualizer, an Information Workspace," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*, New Orleans, LA, USA, 1991, pp. 181–186. doi: 10.1145/108844.108874.

[29] M. Zheng, W. Crichton, A. Narayan, D. Raghavan, and N. Vasilakis, "When Are Reactive Notebooks Not Reactive?." [Online]. Available: https://arxiv.org/abs/2511.21994

[30] T. Manz, M. Scolnick, and A. Agrawal, "Beyond the Shell: Extending Agents with Reactive Python Notebooks," in *Proceedings of the SAO Workshop on AI Agents and Data Systems (CAIS 2026)*, San Jose, CA, USA, 2026. [Online]. Available: https://bauplanlabs.github.io/SAO-workshop/papers/45.pdf

[31] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "noWorkflow: A Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1841–1844, 2017, doi: 10.14778/3137765.3137789.