Open Access

Hash all the things: Caching for fast notebook restarts

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

[1]marimo

SciPy 2025

July 7 - July 13, 2025

Proceedings of the 24th Python in Science Conference
ISSN: 2575-9752

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. Input values whose bytes are accessible are hashed directly, while other inputs are represented by the key of the cell that produced them, computed the same way. Because each cell's key folds in the keys of the cells it depends on, editing one cell can invalidate the cached results of 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. These values are also bundled into marimo's static export, a standalone web page written in HyperText Markup Language (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 in place. 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 needing little user setup.

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 read-evaluate-print loops (REPLs) in which running each block of source code, or *cell*, mutates shared global state. As a result, these notebooks accumulate hidden state, and the outputs saved in the notebook file can differ from those of a fresh top-to-bottom run.

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, reactive notebooks offer runtime configuration, such as lazy executors that mark cells stale instead of running them; however, these primitives require user intervention. We exploit their deterministic execution order to build a caching mechanism that automatically skips recomputation of an expensive cell whose code and inputs are unchanged.

Caching for notebooks and Python is not new; we review prior systems in Background and Related Work. Each prior method asks the user to opt in at a boundary, such as a decorated function, document chunk,

Published Jul 13, 2026

Correspondence to
Dylan Madisetti
dylan@marimo.io

or session. In contrast, reactive notebooks already draw a boundary around every cell, so the author never has to choose where caching applies.

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 notebook sessions that follow marimo's reactive principles. The static browser export additionally makes cached results available to readers running the notebook through WebAssembly.

2. Background and Related Work

A memoized function, or *memo* function [8], stores the result of a call and returns that result when it receives the same inputs again. To find the stored result, the function associates each set of inputs with a *cache key* as an identifier used to index the cache. marimo constructs this key with a *hash*, a short, fixed-size fingerprint derived from data. Matching hashes can stand in for matching data because cryptographic hash functions make it improbable that two different inputs have the same fingerprint. When a system identifies a value by hashing the value's own bytes, the value is said to be *content-addressed*. To reuse a cached cell result, marimo must derive the same key whenever its inputs are unchanged.

A cell's inputs are the variables it reads, which may be defined in other cells or in the global environment. In a reactive notebook like marimo, these variables are statically known prior to execution, as the cell's "refs" derive a dependency graph from source. At run time, marimo additionally knows the values currently bound in memory. It can therefore derive a cell's key from the graph, the cell's code, and the current reference values, rather than from the history of which cells happened to run. Cell-level dataflow tracking has an earlier antecedent in [9].

IPyflow and nbsafety [10], [11] take a different route to reactivity. They retrofit reactivity onto Jupyter by tracing execution to record which variables each cell reads and writes, then flag or re-run cells whose inputs have become stale. Because the resulting graph contains only dependencies observed during execution, different execution histories can produce different graphs for the same notebook. marimo instead derives the graph from source, so the same source produces the same dependencies before any cell runs.

The dependency graph identifies what contributes to a cell, but the cache must still reduce that information to a stable key. For this step, marimo borrows from build systems. Build Systems à la Carte [12] separates a *scheduler*, which decides the order in which tasks run, from a *rebuilder*, which decides whether each task must run again. marimo's reactive dataflow engine supplies the scheduler, while its cache acts as the rebuilder by comparing keys computed from a cell's code and inputs (Cache Keys).

The Nix package manager provides the model for extending a key through the dependency graph [13], [14]. Nix identifies a package with a hash of its inputs, including the hashes of the packages from which it was built. This *recursive hash* makes the package's identity cover its dependency tree. marimo applies the same construction to a reactive notebook: a cell's key incorporates the keys of upstream cells when their values cannot be content-addressed directly. Data-engineering systems use related constructions at coarser granularity. Bauplan and Nessie hash pipeline stages [15], while the workflow engines Nextflow (`-resume`) and Snakemake (`--cache`) key task results on code, parameters, and input hashes [16], [17].

Existing caching systems for Python and notebooks differ in both their unit of reuse and the information they require from the author. IncPy modifies CPython, the reference implementation of Python, to memoize function calls automatically [18]. knitr caches chunks of literate documents whose dependencies the author declares by hand [19]. jupyter-cache re-executes a notebook as a whole when any code cell changes [20]. Streamlit asks the author to choose between `cache_data`, for values identified by content,

and `cache_resource`, for values identified by what produced them [21]. marimo instead uses its key construction (Section 1) to make that choice for each reference, while its reactive graph supplies a reuse boundary around every cell.

There are a few existing scientific-Python memoizers (for comparison to this work see Evaluation). The most similar, mandala [22], provides the end-to-end comparison by memoizing execution inside a `with storage:` context, computing content addresses with `joblib.hash`, and recording which calls produced which values. Alternatively, diskcache [23] provides the storage control by storing values under byte keys supplied by the caller, allowing the evaluation to hold key construction constant while measuring storage cost. Other notable work includes Kishu [7] and ElasticNotebook [24], which checkpoint and migrate notebook state, but are not directly comparable to marimo's mechanism.

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) wastes time but does not return an incorrect result. The key derivation must therefore change whenever a value the cell reads changes, but remain stable under superficial edits such as reformatting or comment changes.

Consider two obvious ways to derive a key. The first derivation hashes the value of every reference the cell reads. A marimo notebook can attempt this because, at runtime, it exposes both the dataflow graph and the reference values bound in memory. This derivation fails for Python values that expose nothing stable to hash. An object's memory address is neither stable nor meaningful as an identity. A weak reference identifies an object rather than its contents. An opaque C-extension object exposes neither its underlying bytes nor a stable text representation. The second derivation hashes the cell's source bytes alone. This key is computable, but it does not change when the cell's reference values change and can therefore produce false positives.

marimo combines the two derivations: it hashes a reference value when possible and otherwise uses the key of the cell that produced the reference. Because the producing cell's key is derived by the same construction, this substitution extends recursively through the dataflow graph. This recursive use of producer keys borrows from build systems such as Nix [13], where a package's identity includes the identities of its dependencies. Section 1 describes how marimo constructs keys over the graph. Section 2 then explains invalidation: editing a cell invalidates only downstream cached results. Neither derivation observes untracked external state such as filesystem contents, network responses, wall-clock time, or randomness. Limitations and Discussion describes the mechanisms that cover some of these cases and the remaining gaps.

3.1. Constructing the key

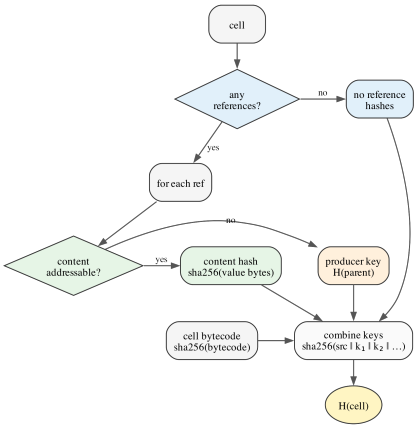
The following will describe computing cache keys from source code and cell references and will outline the mechanism for combining them. Throughout, H denotes the hash function, and $H(c)$ denotes the hash of cell c.

3.1.1. *Hashing source code*

The cells `y = x + 1` and `y = x - 1` may have exactly the same "inputs", but they compute different outputs. As such, it would be incorrect to strictly use a cell's inputs to determine its cache key. Under the assumption that code is deterministic (see Limitations and Discussion for a discussion of side effects), it follows that a cell's code must contribute to its cache key. By using cell code in key construction, basic invalidation is achieved (editing a cell changes its key, so its stale result is never reused).

Source-code formatting and comments should not change a code hash, but details such as the Python version potentially should. marimo therefore hashes the compiled bytecode rather than the source text. Compilation discards edits like comments and formatting, making bytecode stable within a specific Python version. A caveat is that cached results may therefore not transfer across interpreter upgrades. As a result, marimo's browser export (Section 3) requires a Python version compatible with the browser's Pyodide 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. *Figure 1 shows the cascading hashing mechanism for references. The left panel traces how references contribute to a cell key; the right shows the derivation over the full cell, abridged from* `BlockHasher.__init__` *(*`marimo/_save/hash.py`*).*

Hashing compiled code accounts for changes to the body of a cached cell, but not for changes to the values that the cell uses. If the cell has no references, its code hash is the only contribution to its key. Otherwise, the hash key must account for values defined outside of the cached cell. Each reference contributes either a hash of its value or the key of the cell that produced it. We call this choice the *key dispatch*. The dispatch algorithm is illustrated in Figure 1 and outlines three resulting cell-key categories: code only, content-addressed, and producer substitution, labeled in the figure as `Pure`, `ContentAddressed`, and `ExecutionPath`, respectively. For blocks and functions cached within a cell, marimo also incorporates the surrounding cell's code, a special case labeled `ContextExecutionPath` in the figure and described in Section 2.4.

A reference is **content-addressed** when marimo derives a hash from the reference's value. Immutable values, such as numbers, strings, and frozen collections, are hashed this way. Before hashing a reference to a user interface (UI) element such as a slider, marimo replaces the element with its current value, so moving the slider changes the hash. A value that exposes its underlying bytes through Python's buffer protocol is hashed from those bytes. This case covers NumPy ndarrays and other objects advertising NumPy's array interface, making it important for scientific computing. Here, the hash is computed from the contiguous buffer without serialization, an approach borrowed from joblib [25] and mandala [22].

A reference is hashed via "execution path" or **producer substitution** when marimo cannot content-address the reference but knows which upstream cell initialized the value. In this case, it substitutes the producing cell's key for the value'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.

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 collisions between different inputs to SHA-256 are improbable (Background and Related Work), matching keys can safely stand in for matching code and reference contributions.

3.2. Invalidation

Since the construction of Section 1 is recursive, a cell's key may be derived from its producer's key, which in turn may be derived from that producer's key. In practice, derivations do not have to explicitly walk the whole ancestry of a value to compute a key. 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 and the new key no longer matches any stored entry. When this happens, the next lookup misses and the cell recomputes. Invalidation is not an action that marimo performs, since nothing is deleted; the old entry simply stops being addressable.

To keep false negatives rare, invalidation should also be limited, and an edit should not invalidate results it cannot affect. Additionally, updating keys should be cheap. The two subsections below establish each property in turn.

3.2.1. *Invalidation never misses a change*

When the bytes of a content-addressed reference change, the hash changes, and so does every key built from it. However, the ``producer substitution'' case requires an argument tied directly to the notebook's execution. Because marimo is a *reactive* notebook, a value can change only if the cell that produced it re-runs, and a cell re-runs only when its own code or inputs change, which are the ingredients of its key. An unchanged producer key therefore implies an unchanged value, provided cell bodies are deterministic; Limitations and Discussion discusses side effects and mutations that bypass the dataflow graph.

3.2.2. *Invalidation is limited and cheap*

Wherever producer substitution occurs, a cell's key contains the keys of cells upstream of it. By this definition, the notebook's keys form a Merkle directed acyclic graph (DAG) [26], since each node's fingerprint depends on the fingerprints of the nodes on which it depends. 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 key construction on the four-cell PyTorch notebook composed of a seed, a random tensor generated from the seed, a small neural network, and a forward pass that applies the network to the tensor. The italic label under each cell in the figure classifies that cell's own key, as computed by marimo's hasher. By cell label, `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)
```

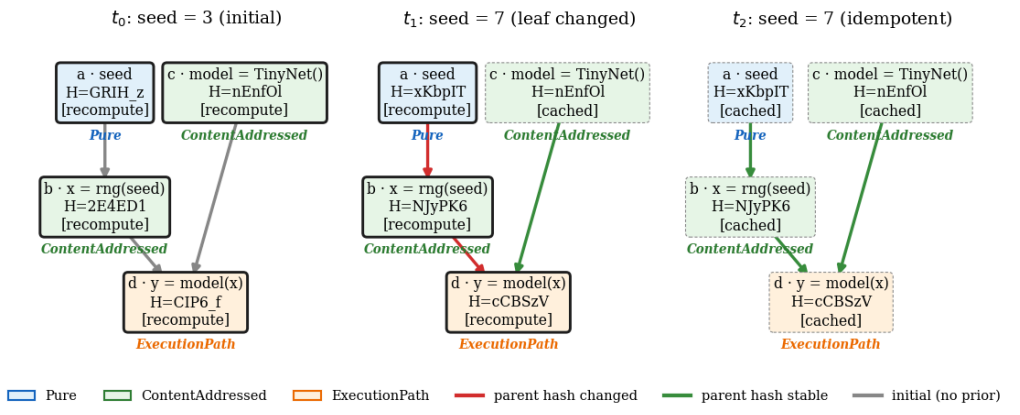


Figure 2. *A worked example of the recurrence on the four-cell graph written out above. Every hash and branch label in the figure is produced by marimo's real hasher on a compiled cell graph at render time. The seed is hashed as an immutable value, the tensor is hashed through its buffer, and the network (`TinyNet`), which exposes no bytes to hash, is represented by the key of the cell that constructed it, covering the `Pure`, `ContentAddressed`, and `ExecutionPath` cases discussed in*

Section 1. Changing the seed ($t_0 \to t_1$) invalidates `a`, `b`, and `d` (red edges), while `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.

3.2.4. *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.

4. Storage and Loading

On a cache hit, marimo must restore the variables the cached cell would have defined. marimo provides a few ``loaders'' that differ in how they store and load values. However, restoring always has two parts: *lookup* finds the stored entry whose key matches $H(c)$, and *loading* deserializes the entry's values into memory. These two parts do not have to happen at the same time, because a notebook does not always need a value's bytes when it finds the corresponding cache entry. For example, a downstream cell may take a variable and pass it to a third cell without inspecting it.

The loader, `LazyLoader`, demonstrates the lookup / loading separation. On write, the loader writes a JSON (JavaScript Object Notation) manifest listing each variable. For each variable, the manifest records either the value itself (small primitives are stored inline) or the path to an external file with the value. Its `cache_type` field records which strategy of Section 1 produced the key. For example, an abridged manifest for a cache named `training` whose cell defined a value, `seed`, and a large array, `x`, might read:

```
{
  "hash": "9V5v6Cji…",
  "cache_type": "ContentAddressed",
  "defs": {
    "seed": {"primitive": 7},
    "x": {
      "reference": "training/9V5v6Cji…/x.npy",
      "type_hint": "numpy.ndarray"
    }
  },
  "stateful_refs": [],
  "meta": {
    "version": 5,
    "blob_hashes": {
      "training/9V5v6Cji…/x.npy": "e3b0c4…"
    },
    "signer_public_key": "-----BEGIN PUBLIC KEY-----\n…",
    "signature": "…"
  }
}
```

When signing support is available, the blob hashes and signature let marimo verify the external files before deserializing them; Limitations and Discussion discusses this protection and its limits.

On lookup, `LazyLoader` binds each variable to a *stub* rather than immediately loading its value. A stub is 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, NumPy `.npy`, Apache Arrow, PyTorch `.pt`, and binary media. The first time the notebook uses the variable, its stub loads the value. A variable that is never used is never loaded.

By contrast, the loader, `PickleLoader`, loads values during lookup. It writes the full `Cache` envelope, holding every variable the cell defined, as a single blob using pickle (Python's built-in serializer). When it finds a matching entry, it immediately loads every variable back into memory.

4.1. WASM portability

Separating stored values into a manifest and individual blobs also lets marimo bundle them with a static notebook. marimo's static WebAssembly (WASM) export is a standalone HTML (Hypertext Markup Language) file that runs the notebook in the reader's browser, with no server, on Pyodide, a Python runtime compiled to WebAssembly. When automatic cell caching is enabled, 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 the export, the browser session derives the same keys and loads matching cached values on first use. A missing or unverifiable entry is treated as a cache miss. Scientific articles (including this one), blog posts, and educational materials can therefore include precomputed results and trained models with the notebook.

The export includes the cached values and the user code, but not the external libraries used to produce those values. A cell may therefore depend on a package 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/, exercises 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. 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 exposes caching with in-memory caching within a session, persistent caching across sessions, and automatic caching of every cell. All three use the key construction described in Cache Keys, so users do not need to declare dependencies or construct keys. Changes to user interface elements and values created with marimo's state primitive, `mo.state`, are included in this construction.

5.1. In-memory caching

The decorator `@mo.cache` memoizes a function in memory. The key for each call combines the function's code, its arguments, and variables it uses from outside its body, following the construction in Section 1. Results stay in the kernel's memory, so a hit reads nothing from disk, but all results are lost when the session ends.

In comparison to Python's built-in **`functools.cache`**, marimo's `@mo.cache` is better suited to reactive notebooks. Since a reactive runtime re-runs the cell that defines the function, it creates a new function with an empty `functools` cache. This discards every stored result, even when the change that caused the cell to run did not affect the function's behavior.

5.2. Persistent caching

The decorator `@mo.persistent_cache` uses the same key construction but writes results to disk through the loaders described in Storage and Loading, allowing the results to survive kernel restarts. This lets a notebook restart without re-running its expensive cells. These files are also bundled with the WebAssembly export described in Section 3.

Additionally, `mo.persistent_cache` can be used as a context manager. A block of code under `with mo.persistent_cache(name="training")` is cached, and on a hit, marimo restores the block's variables without executing the block. Optional arguments select the storage directory (`save_path`) and loader (`method="pickle"` or `method="lazy"`). The `pin_modules` option adds selected library versions to the key.

5.3. Automatic caching of every cell

The decorators and context manager cache only the code the author marks. The `cache_cells` runtime option instead applies caching to every cell. Before running a cell, the runtime computes its key and checks the cache. On a cache hit, the runtime skips the cell's body and restores its variables; on a miss, it runs the cell and saves the results. Since the cache attempts to account for all variables, stored entries may

sometimes contain only a placeholder for variables that cannot be serialized. When a later cell needs that variable, marimo re-runs its defining cell and any upstream cells needed to rebuild it. With `cache_cells` enabled, the author does not need to annotate individual functions, blocks, or cells.

6. Evaluation

Caching does not always save time. A hit pays the cost of deriving a key and loading a value; a miss derives a key and saves a value after running the cell body. When portability or provenance is the goal, this overhead may be worthwhile even if caching is not faster. When the goal is to skip recomputation, the work avoided must repay the overhead.

We measure key derivation, value load, and value save, both separately and as end-to-end hit and miss paths. The payloads are NumPy `float64` arrays ranging from 1 MB to 1 GB. We compare six strategies: `mo.cache`, `mo.persistent_cache` with each of its two loaders, mandala's decorated-function memoization, and diskcache as both a memoizing decorator and a plain store with a fixed key. The fixed-key form performs no key derivation and provides a lower bound on cache-hit time. We measure each cost over 10 runs after priming each cache and one warmup, and report the median. Persisted measurements are keyed on the operating system, architecture, Python implementation and version, and a methodology-version string. A change to any of these properties causes the benchmark to recompute the measurement.

6.1. End-to-end cache evaluation

A notebook user experiences a cache hit as the time needed to derive the key and restore the stored value. Panel (a) of Figure 3 reports this delay for the six strategies. On the build host, every strategy stays below 100 ms for payloads through 50 MB. At 100 MB, both persistent marimo variants remain below this threshold while mandala exceeds it. The dashed line marks the 100 ms threshold below which a response reads as instantaneous [27].

The dotted curve in panel (a) shows how expensive the cell body must be before caching saves time. Let T_{hit} be the cost of serving a hit, and let $T_{\text{key}} + T_{\text{save}}$ be the overhead a miss adds to running the cell body. If a fraction p of lookups are hits, caching saves time when the cell body costs more than $T_{\text{hit}} + \frac{1-p}{p}\left(T_{\text{key}} + T_{\text{save}}\right)$. For these payloads, the measured miss overhead is comparable to the hit cost, so at a hit rate of $p = 0.9$, 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, alongside the key derivation and value save that a miss adds. This decomposition explains the separation between implementations at larger payload sizes. mandala derives its key with `joblib.hash`, which serializes the value through pickle before hashing it [22]. marimo instead hashes the array's contiguous bytes directly through Python's buffer protocol, avoiding the serialization step. On the Apple M4 Max used for the camera-ready figure, mandala is roughly three times slower end to end at the largest measured payload. However, on a Linux x86-64 server, value loading dominates and the ratio shrinks to roughly 1.2 times. The difference therefore appears to be hardware-dependent. marimo's value-load time closely matches the fixed-key diskcache control, indicating that the gap to mandala comes from key derivation rather than storage.

Panel (c) reports the variation across individual cache hits at the largest payload size. `diskcache.memoize` does not appear at this size because the payload exceeds the blob-size limit of its underlying SQLite database.

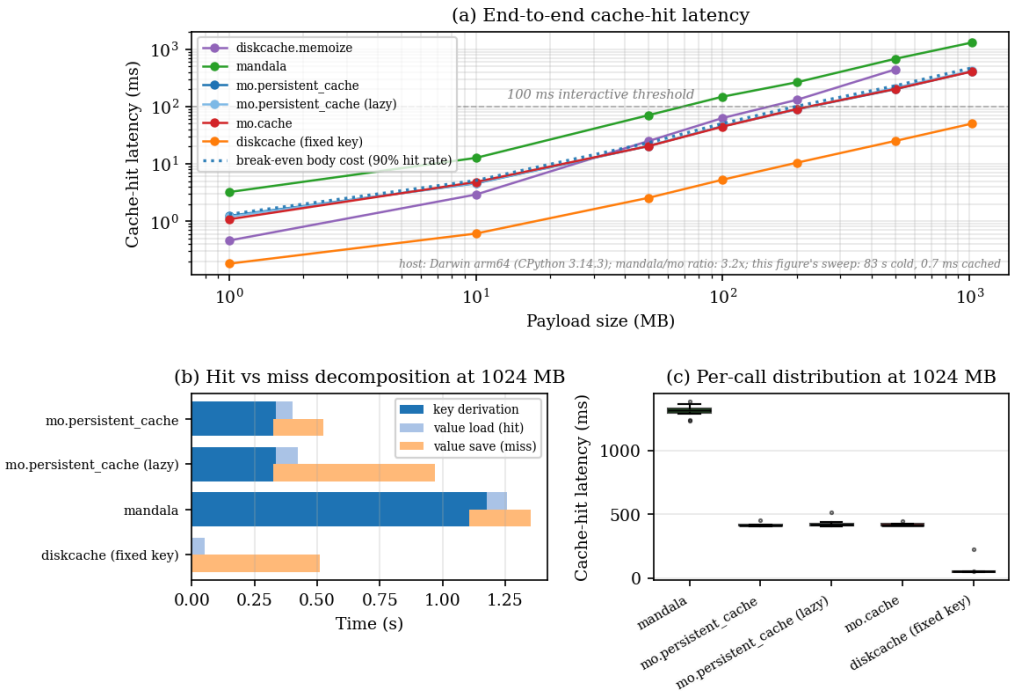


Figure 3. *End-to-end cache evaluation on NumPy `float64` payloads. (a) Cache-hit latency versus payload size on logarithmic axes. The dashed line at 100 ms marks the interactive threshold [27], and the dotted curve gives the break-even body cost at a 90% hit rate. (b) The cost of a hit (key derivation and value load) and the overhead added by a miss (key derivation and value save) at the largest payload size. (c) The distribution of cache-hit measurements at the largest payload size. `diskcache.memoize` does not appear at this size because the payload exceeds the blob-size limit of its underlying SQLite database. The host label reports the build host, the measured mandala-to-marimo ratio, and the cold and cached costs of the figure's own benchmark sweep.*

The camera-ready figure was produced on a MacBook Pro with an Apple M4 Max. Because the relative costs of hashing, serialization, and loading depend on the hardware, the figure reports its build host and measured mandala-to-marimo ratio; a build on different hardware reports its own result. The stage decomposition measures the actual disk-backed loading paths for both marimo loaders and for mandala, including the operating system's file cache and the cost of reconstructing the cache envelope. An in-memory deserialization benchmark would omit both costs.

7. Limitations and Discussion

The current implementation has five limitations.

Library versions. The decorator and context-manager APIs do not include library versions in the key by default; the user can include selected versions with `pin_modules` (Using marimo's cache). Without module pinning, upgrading a package can therefore produce a stale cache hit. Automatic cell caching includes library versions by default, so a version mismatch produces a cache miss instead.

Python versions. Because code is hashed as bytecode, keys may change across Python versions (Section 1). Upgrading the interpreter may therefore invalidate cached results. When it does, the failure is safe: the keys miss and the cells recompute. It is also why the browser export requires the exporting interpreter to be compatible with the browser's Python version (Section 3).

Mutation outside the graph. Mutations that bypass the dataflow graph can change a value without changing any cache key. Examples include aliased mutation through a closure and attribute writes on an object that cannot be content-addressed. Downstream cells can then receive a stale cache hit that marimo cannot detect. A suite of checks that runs on every build of this paper exercises this case, and Rex [28] probes the same boundary.

Side effects. External state does not enter a cache key unless the notebook represents it as a dependency. File reads, network calls, randomness, and wall-clock queries can therefore change a cell's result without changing its key. marimo can represent two such dependencies explicitly: `mo.watch.file` includes a file's contents in the key, and `mo.watch.directory` includes a directory listing. Other sources of external state and nondeterminism are not tracked automatically.

Cache integrity and trust. Unpickling can execute arbitrary code, so loading a cache that an attacker has modified could run a malicious payload. When cryptographic signing support is available, the lazy loader signs caches by default. The signature covers the manifest and the SHA-256 hash of every stored blob, and marimo verifies them before deserializing any value. The pickle loader has no such protection and should load caches only from trusted sources. Signatures show that a cache was produced by a trusted key and has not been modified; they do not establish which keys a reader should trust or whether the signed result is correct. Sharing caches therefore requires a way to establish trust in the signing key.

8. Future work

The evaluation and limitations suggest four directions for future work.

Performance. Section 7 measures the costs of cache hits and misses. marimo could measure these costs while a notebook runs and apply the break-even rule automatically, declining to cache a cell when the cache costs more than the computation it would avoid. The same measurements could report how much execution time each cached cell saved.

Additional external dependencies. The explicit-dependency mechanism used for files and directories could be extended to sources such as randomness, network responses, and time. The open design question is which sources justify dedicated interfaces and how each source should contribute a stable value to the key.

Persistent-cache management. The persistent loaders do not impose a disk-capacity or eviction policy. Future work could add size limits, cleanup policies, and storage accounting that reflects the different formats used for cached values.

Provenance-aware reuse. Persistent caches already reuse results across sessions. They could also record and expose which computation produced each reused value, connecting cache reuse with systems for tracking computational provenance [29].

9. Conclusion

We have presented a caching mechanism for marimo, a reactive notebook for Python. A cell's cache key combines a hash of its bytecode with hashes derived from its references; when a reference cannot be hashed by content, the key of the cell that produced it stands in. This construction invalidates cached results when their code or tracked inputs change, while preserving them across formatting and comment changes. On the Apple M4 Max used to build this paper, both persistent marimo loaders served 100 MB arrays in under 100 ms, and mandala took roughly three times as long as marimo's persistent cache at the largest measured payload. At the same payload on Linux x86-64, mandala took roughly 1.2 times as long. marimo can apply the mechanism automatically to every cell without per-cell annotations and include the resulting caches in its static WebAssembly export. The export published at https://dmadisetti.github.io/scipy_proceedings/ demonstrates how readers can use precomputed results and models in a browser-only Python runtime.

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/b15166.

[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] 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.

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

[26] 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.

[27] 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.

[28] 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

[29] 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.