Declare, Resolve, Reuse: A Portable Data Layer for Notebook Workflows Across HPC Systems

Md Saiful Islam*, Reyer Band*, Jin Zhou*, Tanu Malik†, Kevin Lannon*, Douglas Thain*

*University of Notre Dame, Notre Dame, IN, USA; †University of Missouri, Columbia, MO, USA

mislam5@nd.edu, rband3@nd.edu, jzhou24@nd.edu, tanu@missouri.edu, klannon@nd.edu, dthain@nd.edu

***Abstract*—Notebook workflows are widely used for rapid prototyping and exploratory scientific analysis, but they remain difficult to scale and rerun on new HPC systems. Scaling such workflows requires workflow managers, resource provisioning, and compatible runtime environments. However, for data-intensive notebooks, the primary barrier to portability remains how data dependencies are referenced, staged, and validated. These choices also determine whether previously staged data can be reused. Site-specific paths, download commands, and staging assumptions are commonly embedded directly in notebook code, so the same notebook often requires manual edits to run on another system or to reuse previously staged data. To address these challenges, this paper presents a portable data layer for notebook workflows across heterogeneous HPC systems. Building on the backpack abstraction, which packages notebook execution context, we extend the model with a declarative data specification that separates data dependencies from code. The specification unifies access to heterogeneous sources, supports profile-based dataset selection and multi-source fallback, records validation metadata, and exposes data to notebooks at stable runtime paths. We implement this in the Floability runtime with an HPC-local cache that combines source metadata with the specification to make data identity explicit and enable safe reuse across runs. We evaluate the layer with synthetic cache benchmarks and three real workflows using datasets from 7.5 GB to 108 GB. The same backpacks run unchanged across three HPC systems spanning two scheduler families and different storage architectures. The caching mechanism adds little overhead and enables efficient reruns without manual code changes.**

***Index Terms*—notebook workflows, data portability, heterogeneous HPC systems**

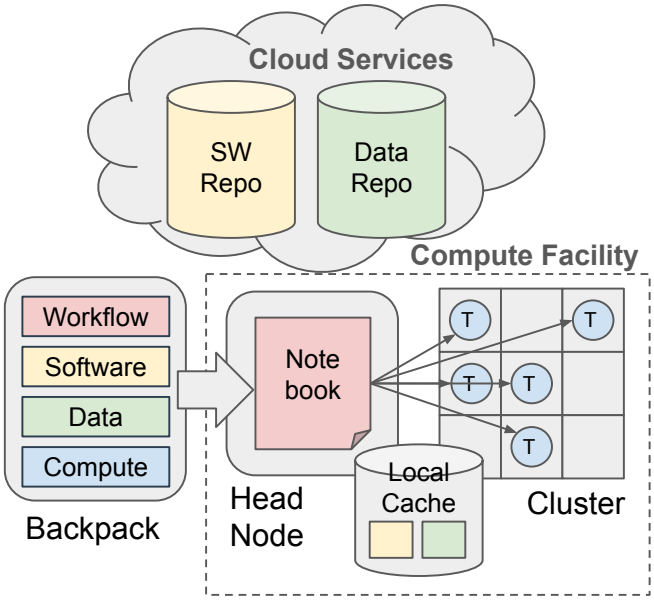


Fig. 1: Portable execution of a data-intensive notebook workflow using a Floability backpack. The backpack specifies workflow, software, data, and compute requirements. Floability resolves software and data from external services, caches data on site-local storage, and deploys the notebook and distributed tasks (`T`) across HPC head and compute nodes.

I. Introduction

Interactive notebooks combine code, results, and narrative in a single shareable artifact, and from the inception of Jupyter they were promoted as a publishing format for reproducible computational science [1]. In practice they rarely meet that promise once they leave the machine on which they were written. A study of 1.4 million notebooks on GitHub found that only about a quarter could be executed end-to-end without error, with failures primarily attributed to missing dependencies and data-path errors [2]. A more recent study of 27,271 notebooks linked from PubMed Central publications found that only 1,203 ran without error and 879 produced outputs identical to those originally reported [3]. Part of this fragility is internal to the notebook execution model. Unlike scripts, notebooks permit out-of-order cell execution and carry hidden kernel state, so they can fail even in place. However, both studies attribute most failures to factors outside the notebook itself, such as unavailable data and environment differences.

The problem intensifies for data-intensive workflows that scale beyond a single node on distributed frameworks such as Dask [4], Parsl [5], Pegasus [6], and TaskVine [7], and that run on heterogeneous HPC systems with different schedulers and storage. The *software* side of this portability problem is increasingly well served: container runtimes [8], [9] and relocatable environment packagers [10], [11] make software environments portable across sites. For data-intensive notebooks, the part that still breaks is *data*. Federated data infrastructures such as XRootD [12], Pelican [13], and Globus [14] move bytes efficiently at the infrastructure level, but an unmodified notebook cannot benefit from them without a workflow-facing abstraction for how its data is named, staged, and validated.

The same gap that prevents first-time reproduction on a new system also affects efficient reruns on the same one.

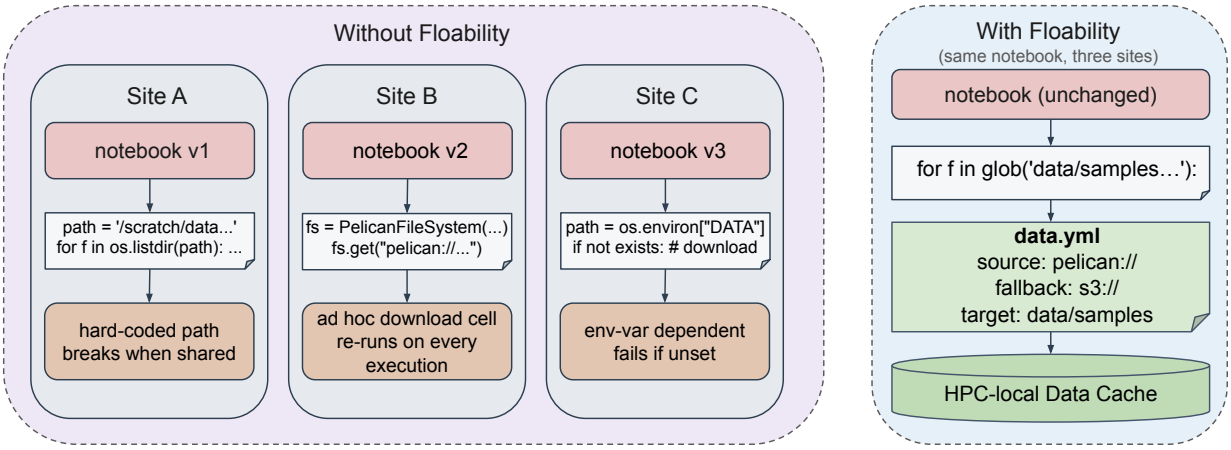


Fig. 2: The data portability gap in notebook workflows. Without explicit data abstractions (left), running the same workflow on three HPC systems with different storage architectures and access mechanisms requires three notebook variants, each fragile in different ways (failures in brown). With Floability (right), a single `data.yml` declares sources and fallbacks; the HPC-local cache resolves them and materializes datasets at stable paths, so the same notebook runs unchanged across all sites.

Users often reuse a workflow by modifying its analysis logic while the software environment and most input data stay the same. Without an explicit notion of what data the workflow consumed, the system has no reliable way to recognize that prior staged data still match the current request. Users are then left to manage reuse manually, by tweaking notebook code, which is error-prone and rarely scales beyond a single experiment. Scientific workflow standards and FAIR principles call for reusable computational artifacts [15], [16], and prior work has shown the value of reusing nearby data instead of re-transferring it [17]. Portable execution therefore requires not only capturing the execution context, but also identifying and reusing unchanged data and software artifacts across runs.

In prior work, we introduced the *backpack* abstraction [18], which packages a notebook with software, data, and resource specifications for execution across heterogeneous HPC systems. Backpacks showed that externalizing execution context substantially improves portability, but focused on pre-packaged datasets. Many scientific workflows depend on large datasets hosted in external storage systems, object stores, or federated data services. For portable execution and efficient reruns, data items need to be uniquely referenced, verifiable through explicit identity, and accessible at stable runtime paths regardless of the underlying source or storage architecture.

This paper addresses these gaps with a *portable data layer* for notebook workflows, where a workflow *declares* its data once, the runtime *resolves* each item across heterogeneous sources, and an HPC-local cache lets unchanged data be *reused* across runs. A declarative data specification, kept separate from notebook code, names required datasets, unifies access across storage backends, supports profiles and multi-source fallback, and records validation metadata. Floability resolves each item, materializes it at the stable path expected by the notebook, and caches it on site-local storage. The cache combines the specification with observed source metadata, enabling unchanged data to be reused while changes trigger re-staging. Figure 1 shows the resulting abstraction, and Figure 2 contrasts it with site-specific notebooks containing hard-coded paths, ad hoc download cells, and environment-dependent staging.

We evaluate the data layer with synthetic cache microbenchmarks and three real data-intensive notebook workflows whose datasets range from 7.5 GB to 108 GB, run with the same backpacks across three HPC systems that span two scheduler families and different storage architectures. Because schedulers, storage, and queueing differ across these sites, we report runtime composition rather than ranking sites by raw performance. Across all three, metadata-based cache lookup adds little overhead, repeat runs drop data materialization and environment setup from the critical path, and the notebook code itself never changes, because data sources, staging policies, cache reuse, and runtime paths all live outside it.

II. Dimensions of Data Handling in Portable Notebook Workflows

Data handling is a major source of fragility when notebook workflows move across HPC systems. Notebooks often embed assumptions about how data are referenced, moved, and identified across runs, making them difficult to preserve across different storage architectures and access policies. We organize these concerns into three dimensions, reference, movement, and identity, which motivate declaring, resolving, and reusing data through a portable data layer.

A. How Data Are Referenced

Notebooks commonly bind data to code in three ways (Figure 3). In the *embedded* model, data are packaged with the notebook or repository. This works for small examples, but it does not scale to multi-gigabyte datasets and complicates versioning when data evolve independently of code.

In the *location-based* model, notebooks reference explicit paths, URLs, and object-store locations. This is convenient

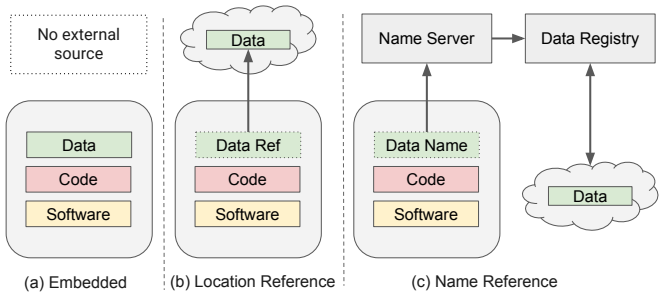


Fig. 3: Three ways notebook workflows bind data to code: embedded, location-based, and name-based. In all three, assumptions about where the data live and how they are fetched remain inside the notebook.

during development, but it embeds site-specific assumptions in notebook cells. Paths, credentials, network access, and storage layouts often differ across HPC systems, and download cells may re-fetch unchanged data on every run.

In the *name-based* model, datasets are referenced through logical identifiers resolved by federated data services such as Pelican [13] and XRootD [12]. Logical names can be more portable than physical paths, but notebooks still often contain the resolution, retry, and fallback logic needed to use them.

Across all three, the problem is not the reference mechanism but that assumptions about data location, access, and fallback are embedded in the notebook rather than *declared* as part of the workflow.

B. When and Where Data Are Accessed

Portable execution also depends on when and where data movement occurs. Data may be pre-staged manually, fetched and distributed by a workflow manager, or pulled directly by workers during execution, and each pattern suits a different setting. But notebooks rarely declare which pattern they assume, so a workflow may expect shared storage where the target site requires worker-local staging, or worker internet access where compute nodes are isolated. Left for the notebook rather than a runtime to *resolve*, implicit movement also causes late failures, silent drift when remote data change between runs, and redundant transfers when data are re-fetched without checking for a valid staged copy.

C. How Data Are Identified

Portable execution also requires knowing which data were used, not just where they are. Notebooks rarely record sizes, checksums, or versions, so the runtime cannot tell whether two executions used equivalent inputs, safely reuse cached data, or detect silent changes. Explicit identity therefore connects validation, provenance, and reuse: it catches unexpected inputs before execution, records what a workflow consumed, and lets unchanged datasets be *reused* across runs.

D. Toward a Portable Data Layer

Across these three dimensions, the common problem is the same: data handling stays embedded in notebook code rather than declared and externalized. A portable data layer instead lets a workflow *declare* its datasets separately, *resolve* each to a concrete source at a stable, site-independent path, and *reuse* unchanged data through recorded identity metadata. Sections III and IV present the specification and runtime that implement this model.

III. Declarative Data Specification

We refer to the combination of a declarative data specification and the runtime that acts on it as the *portable data layer*: a workflow *declares* its data, and the runtime *resolves* each item to a concrete source and *reuses* unchanged data across runs, exposing every dataset at a stable notebook path. This section presents the specification; Section IV presents the runtime.

A. Backpacks and Floability

A *backpack* packages one or more notebooks with a declarative specification of their execution context, including software dependencies, data requirements, and resource assumptions, so they can run outside their original environment [18]. *Floability* deploys and executes backpacks across heterogeneous environments: given a backpack, it materializes a concrete execution instance with the configured software environment, resolved workflow directory, and staged data, and can instantiate the same backpack repeatedly across sites without modifying notebook code.

B. Structure of the Data Specification

Data requirements are expressed in a YAML document, conventionally `data.yml`, containing a schema version and one or more *profiles*. A profile is a named collection of data items for a particular context, such as a test, production, or site-specific dataset, and may include a policy block that controls resolution-time behavior such as retries and timeouts.

A *data item* is the basic unit: it declares a logical name, a source type, a source location, and a target path relative to the workflow directory, with optional integrity metadata such as expected size and checksum. These fields separate notebook-facing paths from physical locations: the notebook uses the declared target path, while the runtime resolves the source and prepares the data before execution. Figure 4a shows two profiles mapping different datasets to the same runtime path, and Figure 4b shows a data item with fallback sources.

C. Profiles for Dataset Selection

Profiles let the same notebook run with different datasets without code changes: when profiles share target paths, selecting a different profile changes the staged data while preserving the filesystem layout the notebook sees, supporting variants such as development versus production inputs or alternate dataset versions. In Figure 4a, `train_set` and `test_set` point to different external datasets but both materialize at `data/samples`, so a notebook reading `data/samples` runs unchanged under either.

```yaml
schema_version: 1.0
profiles:
  train_set:
    policy:
      retry_attempts: 0
      timeout: 30
    data:
      - name: train_data
        source_type: pelican
        source: "pelican://data.example.org/train/"
        target_path: "data/samples"

  test_set:
    data:
      - name: test_data
        source_type: pelican
        source: "pelican://data.example.org/test/"
        expected_size: 41943040
        checksum: sha256:abc123...
        target_path: "data/samples"
```

(a) Multiple profiles map different datasets to the same runtime path.

```yaml
schema_version: 1.0
profiles:
  dataset_profile:
    data:
      # Single file with validation
      - name: config_file
        source_type: http
        source: "https://example.org/config.yaml"
        target_path: "config/config.yaml"

      # Multi-source fallback
      - name: model_weights
        source_type: multi
        sources:
          - source_type: pelican
            source: "pelican://data.example.org/model.bin"
          - source_type: s3
            source: "s3://example-bucket/model.bin"
        target_path: "models/model.bin"
```

(b) A single data item declares fallback sources tried in order.

Fig. 4: Examples of declarative data specifications. (a) Profile-based dataset selection. (b) Source abstraction and multi-source fallback.

D. Source Types and the Policy Block

Data items abstract over heterogeneous storage through a common source interface. Floability supports federated data services, S3-compatible object stores, HTTP endpoints, local filesystem paths, and backpack-relative paths for small packaged inputs; each has a different access mechanism but is presented uniformly as a source. The policy block applies to all items in a profile and describes how the runtime attempts source resolution, including timeout and retry behavior, so different execution contexts can use different resolution behavior without changing the notebook or duplicating each item.

E. Multi-Source Fallback

A data item may declare multiple sources, tried in order until one succeeds, which is useful when the same dataset is available through different systems, such as a federated service at one site and an object store at another. In Figure 4b, `model_weights` prefers a Pelican source and falls back to S3, both materializing at `models/model.bin`: the notebook declares what file it needs and where it should appear, while the runtime selects the source during materialization.

IV. Materializing the Specification

This section describes the runtime half of the data layer: how Floability acts on a declarative specification during execution, realizing its remaining two capabilities. *Resolve* turns each declared item into staged data at the target path, selecting among sources and validating what it stages. *Reuse* recognizes when staged data already satisfy a request and serves it from an HPC-local cache instead of re-fetching. Together they produce site-independent data across heterogeneous HPC systems.

A. Data Operations and Materialization

Floability resolves each declared item through an operations model that separates metadata validation, data transfer, and integrity checking. As shown in Figure 5, a workflow selects one of three modes, `check`, `fetch`, or `verify`, that determines how each dependency is processed before notebook execution.

a) ***Check****:* The `check` operation performs metadata-only validation of declared data sources. Floability verifies that each source exists and, when available, that basic attributes such as size match the specification, without downloading data. This mode is useful for workflows that should not stage large datasets upfront, for data already present on shared filesystems, or for workflows in which workers access data directly from remote services.

b) ***Fetch****:* The `fetch` operation stages data from the specified source into the execution environment. When size metadata are provided, transferred data are validated against expected sizes. Fetch enables workflows to materialize required inputs before execution without requiring full integrity verification.

c) ***Verify****:* The `verify` operation extends `fetch` by performing integrity checks using checksums. For directory datasets, verification uses a composite checksum derived from directory contents. This mode provides stronger correctness guarantees for data-intensive workflows.

As illustrated in Figure 5, `fetch` and `verify` may optionally stage data through a local cache before materializing datasets into the workflow instance. When caching is disabled, data are written directly into the instance view.

B. Cache Lookup and Metadata-Based Keys

a) ***Cache key generation****:* Each cached dataset is identified by a key combining two components (Figure 6). The *spec key*, computed from normalized specification fields such as source type, location, name, expected size, and checksum, identifies the requested data item and needs no contact with the remote source. The *remote key*, computed from source metadata, identifies the currently observed version: for a single file it uses size, modification time, and ETag when available; for a directory the runtime lists entries deterministically and fingerprints sorted per-file paths and metadata. The final cache path combines the two keys, so multiple observed versions of the same requested dataset can coexist.

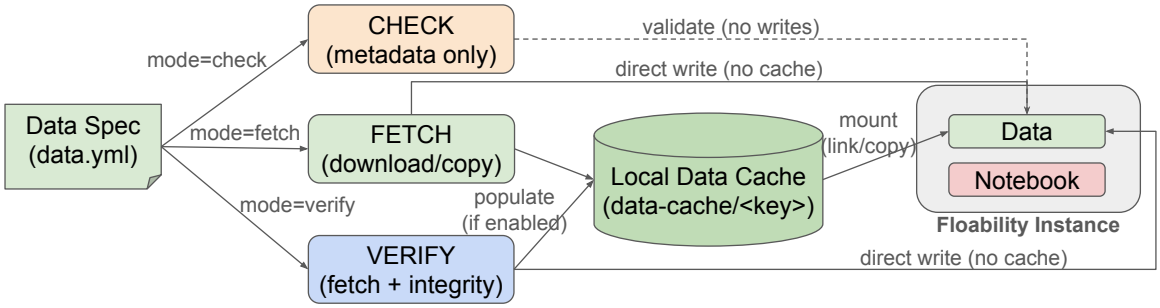


Fig. 5: Floability's data operations model. The specification selects one of three modes (`check`, `fetch`, `verify`) that determines how each dependency is processed. `fetch` and `verify` may optionally stage data through an HPC-local cache before materializing datasets into the workflow instance.

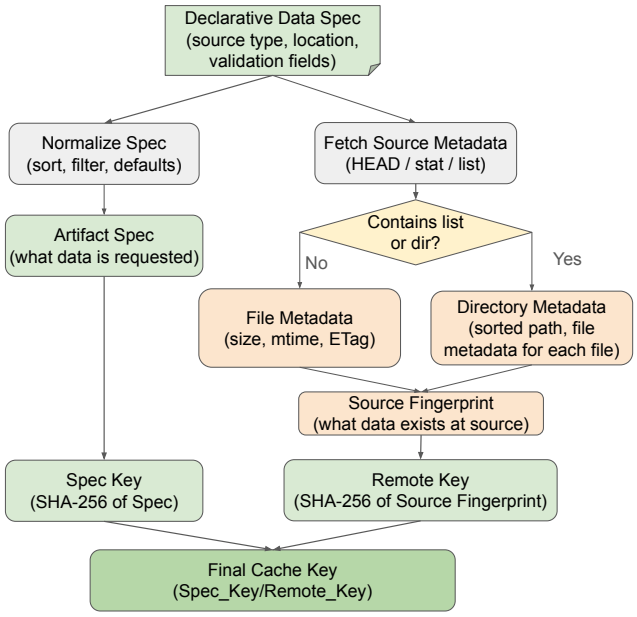


Fig. 6: Cache key generation from declarative specifications and source metadata. The spec key identifies the requested data, while the remote key identifies the observed source state.

b) ***Cache lookup****:* Before fetching from any external source, the data layer checks the HPC-local cache for a matching entry. In strict mode, both the spec key and remote key must match. On a hit, data transfer is skipped and the cached dataset is materialized through the same mechanism used for fresh fetches, so reuse is transparent to the notebook, which sees the same target path whether the data were just transferred or reused. In non-strict mode, if the remote source is unavailable or cannot be validated, Floability may reuse an existing entry with the same spec key, letting a workflow continue with a previously cached version.

c) ***Cache invalidation****:* Invalidation follows from the key construction: a changed specification changes the spec key, and changed source metadata changes the remote key. In strict mode, either change causes a cache miss and triggers re-staging. Storage systems with unusual metadata semantics can produce conservative misses where reuse might have been possible, but changed remote data are not silently reused under strict lookup. To reclaim space, Floability provides explicit, user-invoked reclamation, with an option that retains only the active working set, rather than automatic eviction that risks deleting data an imminent run will reuse.

d) ***Data materialization into workflow instances****:* Once the runtime has decided whether to fetch fresh data or reuse a cached copy, the resolved dataset is exposed at the target path declared in the specification. The mechanism is site-dependent: symbolic or hard links where filesystem semantics allow them, copies otherwise. The choice is invisible to the notebook, which sees the same target path regardless, and materialization completes before notebook execution starts, so all declared inputs are present when the first cell runs.

C. Failure Handling and Assumptions

If a fetch from the preferred source fails, the runtime tries the next declared source; if all sources fail, the workflow does not start, so incomplete data never reach execution.

Three assumptions are worth stating. First, source-side metadata such as size and modification attributes are treated as indicators of source state; strict lookup detects changes only when the backend exposes them through these metadata. Second, the data layer uses but does not manage external delivery infrastructure such as Pelican, XRootD, and S3, including authentication: delivery clients inherit credentials from the ambient environment, so `data.yml` carries no secrets. Third, the HPC-local cache requires writable site storage; without it, the system falls back to direct-to-instance staging.

V. Evaluation

The evaluation asks whether the data layer delivers the capabilities named in its design. First, what does it cost to *resolve* and *reuse* data through the metadata-based cache, i.e., what overhead does cache lookup add for file and directory datasets? Second, do the *declare* and *resolve* mechanisms let the same data-intensive notebook workflows execute unchanged across heterogeneous HPC systems? Third, how does *reuse* change the runtime composition of repeated executions?

TABLE I: Evaluated notebook workflows and datasets.

Workflow	Domain	Dataset Size	# Tasks	Env. Size (MB)
DV5	HEP	∼108 GB	44,010	3,128
LFV	HEP	∼7.5 GB	475	2,433
DConv	Image Processing	∼93 GB	48,528	3,051

A. Experimental Setup

We evaluate the data layer with two experiments. First, we use synthetic datasets to isolate cache lookup and fingerprinting overhead from notebook execution. Second, we execute real data-intensive notebook workflows across three HPC systems using the same backpack and notebook code at each site. The goal is to characterize the layer's overhead, portability, and reuse behavior rather than to compare raw site performance.

a) ***HPC sites****:* End-to-end experiments run on ND CRC, Purdue Anvil, and TACC Stampede3. ND CRC uses HTCondor, while Anvil and Stampede3 use Slurm. The systems differ in scheduler policy, queue behavior, storage configuration, and worker startup latency. These differences can substantially affect notebook execution time, so we report runtime composition within each workflow-site pair rather than treating the sites as directly comparable.

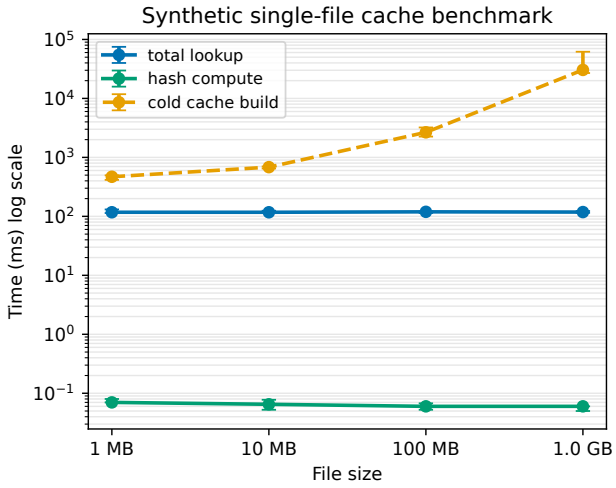
b) ***Workflow applications****:* We evaluate three backpacked workflows: DV5, a distributed high-energy physics analysis over simulated LHC data; LFV, a lepton flavor violation analysis for Higgs decay candidates; and DConv, an image convolution workflow over tiled GOES-16 satellite images. Table I summarizes their dataset and execution characteristics.

c) ***Synthetic cache benchmarks****:* We construct synthetic S3 datasets to measure cache lookup independently of notebook execution. The single-file benchmark varies object size from 1 MB to 1 GB. The directory benchmark varies file count from 10 to 10,000 using mixed file sizes, spanning total directory sizes from roughly 277 MB to 134 GB and thereby bracketing the real workflow datasets. Each benchmark is repeated ten times, and we report medians with interquartile ranges (IQR) for total lookup time, fingerprint computation time, and cold cache-build time.

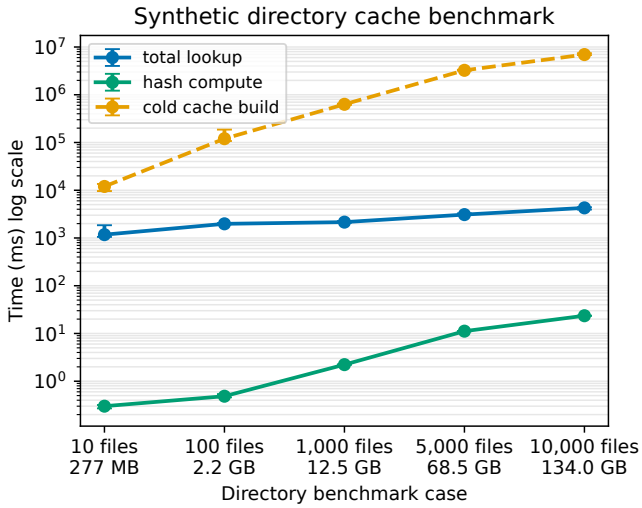
d) ***Run types****:* A cold run starts with an empty data cache and includes cache lookup, data materialization, environment setup, and notebook execution. A repeat run reuses cached data and a prepared unpacked environment, so it includes only cache lookup and notebook execution. We perform three cold and three repeat runs for each workflow-site pair and report mean values; run-to-run variability is summarized in Section V-C.

B. Cache Lookup Microbenchmarks

We first isolate the cost of cache lookup from notebook execution using synthetic S3 datasets. Figure 7 reports median timings over ten runs with interquartile ranges, separating total lookup (source metadata access plus cache-key generation), fingerprint computation, and cold cache build (transferring data and writing the entry).



(a) Single-file cache lookup as object size increases.



(b) Directory cache lookup as file count increases.

Fig. 7: Synthetic cache lookup microbenchmarks (log scale). Points show the median over ten runs and error bars show the IQR. The single-file benchmark varies object size; the directory benchmark varies file count and total directory size. The dashed cold cache-build line is shown for contrast.

Single-file lookup stays nearly constant at about 0.12 s as object size grows from 1 MB to 1 GB (Figure 7a) because the runtime keys on source metadata such as size, modification time, and ETag rather than reading or hashing contents. Fingerprint computation is negligible and lookup stays close to the cost of a metadata query.

Directory lookup grows with file count, from about 1.2 s at 10 files to 4.3 s at 10,000 files (Figure 7b), since the runtime lists entries and fingerprints sorted per-file metadata; even at 10,000 files fingerprint computation stays under 24 ms, and lookup remains two to three orders of magnitude cheaper than a cold build. The design thus avoids full content hashing before reuse.

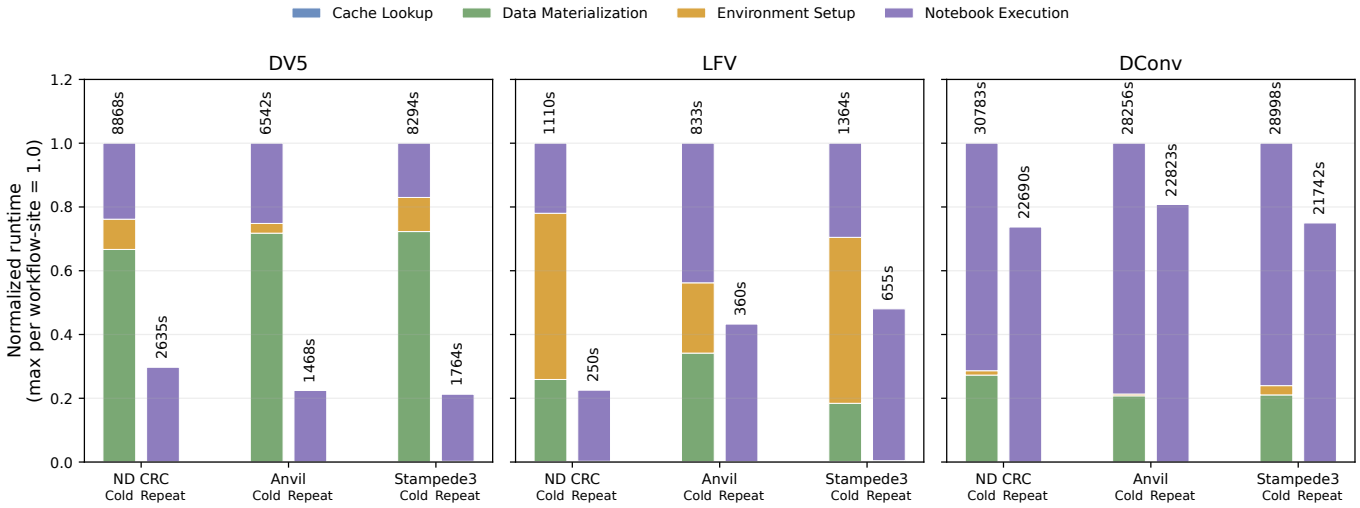


Fig. 8: Normalized runtime breakdown for cold and repeat runs across three HPC systems. For each workflow-site pair, the longer of the cold and repeat runs is normalized to 1.0; the label above each bar gives the absolute mean total. Stacks show the contribution of cache lookup, data materialization, environment setup, and notebook execution. Cache lookup is included but contributes a median of only about 2 s, too small to be visible at this scale; Figure 7 reports its cost in isolation. Each bar is the mean of three runs; run-to-run variability is reported in the text rather than as error bars (see Section V-C).

C. Cross-Site Execution and Cache Reuse

We evaluate whether the same data-intensive notebook workflows execute unchanged across ND CRC, Anvil, and Stampede3, and how cache reuse changes runtime composition. All three workflows execute successfully on all three systems using the same notebook code and backpack specification. Because data sources, staging policy, cache reuse, and runtime paths are declared outside the notebook and resolved by the runtime, moving a workflow to a different site requires no edits to notebook cells; only site-specific scheduler and filesystem options are passed on the command line.

Figure 8 reports the runtime breakdown. The results are not a performance comparison across systems: the platforms use different schedulers, storage, queue policies, and allocation models, which introduce substantial runtime variability. Each workflow-site pair is therefore normalized independently, with the longer of its cold and repeat runs scaled to 1.0, so the figure emphasizes phase composition within a site. We deliberately do not report cross-site speedups, since with these factors uncontrolled such a comparison would measure the platform rather than the data layer.

Repeat runs remove data materialization and environment setup at every site, leaving cache lookup and notebook execution. The removed phases dominate some cold runs, such as DV5, but form a smaller fraction of compute-intensive DConv runs. Cache lookup remains negligible and consistent with the microbenchmarks.

Each reported value is the mean of three runs, with a median coefficient of variation (CV) of 6.6%. Cold runs were stable (1–13%), while repeat runs varied more because, once staging and setup are removed, their remaining runtime is dominated by notebook-execution variability rather than the data layer.

D. Summary and Qualitative Observations

The evaluation supports three findings. First, *metadata-based lookup is cheap and content-independent*: single-file lookup stays near 0.12 s from 1 MB to 1 GB, directory lookup grows only from 1.2 s to 4.3 s between 10 and 10,000 files. Second, *declaring data externally and resolving it at runtime yields portability*: the same notebooks run unchanged across three systems. Third, *reuse behaves consistently across sites*: repeat runs drop data and environment setup time, whether those phases dominate the cold run (DV5) or not (DConv).

A few qualitative observations are worth noting. Anvil and Stampede3 required a few batch options, such as a queue name and expected runtime, but these were passed as CLI arguments with no change to the backpack. We staged data and environments on the scratch directory, whose performance varies across sites and affected environment-creation and cache-build times. In total, the evaluation comprises 54 runs collected over an extended period, with runtimes varying with cluster load.

VI. Related Works

We review representative work across environment portability, reproducibility, dependency tracking, workflow execution, and large-scale data management, positioning ours by its gaps in data handling for portable notebook workflows.

Containerization and Environment Management. Container technologies such as Singularity/Apptainer and Charliecloud provide strong isolation and portability for HPC workloads with near–bare-metal performance [8], [9], [19]. Lighter-weight approaches, including Conda-pack, Poncho, Poetry, and Pipenv, produce relocatable environments without full OS virtualization [11], [20]. HPC-oriented package managers like Spack and EasyBuild support reproducible builds but require

site-specific recipes [10], [21]. Our backpack abstraction occupies a middle ground, capturing sufficient execution context without the overhead of full containers.

Workflow Systems and Distributed Execution. Workflow systems such as Pegasus, Nextflow, and Snakemake support scalable, fault-tolerant execution across clusters [6], [22], [23], but require workflows to be expressed in declarative or domain-specific languages. Within notebooks, Dask and Parsl enable distributed execution via explicit APIs [4], [5], while Papermill and nbconvert execute notebooks as batch jobs [24]. Floability preserves interactive notebook development while enabling scalable execution through external orchestration.

Notebook Reproducibility and Provenance. Prior studies document widespread reproducibility failures in notebooks due to missing dependencies, implicit state, and unavailable data [1]–[3], [25]. Provenance-based systems such as ReproZip, noWorkflow, SciUnit, and FLINC capture execution dependencies to support re-execution and auditing [26]–[30]. These tools focus on provenance and dependency capture, but do not address data staging or reuse across heterogeneous HPC sites.

Federated Data Access and Caching. Federated data systems such as XRootD, Rucio, and Pelican enable scalable access to distributed datasets [12], [13], [31], and capability-based frameworks such as SciTokens add secure, credentialed access to remote scientific data [32]. Caching infrastructures like StashCache and the Open Science Data Federation improve locality and performance [17], [33]. Closer to data identity, data-management systems such as DataLad version datasets and track their provenance alongside code [34]. Stork established data placement as a first-class, independently scheduled activity rather than a side effect of computation [35]. These systems move, secure, or version data effectively, but they operate largely outside notebook execution models and do not provide a workflow-facing abstraction that declares data, resolves it across sources, and reuses it at stable paths across heterogeneous HPC sites.

Positioning of This Work. These lines of work each address part of the portability problem but leave the data side to the notebook. Containers and relocatable environments make software portable [8], [10]; workflow systems scale execution but require declarative or domain-specific languages [6], [15]; provenance tools capture what a run depended on but do not stage or reuse data across sites [26]; and federated data, caching, and versioning systems move and version data but sit outside the notebook execution model [12], [17], [34]. The backpack abstraction in Floability captures the software and execution context [18] but assumed pre-packaged data. This paper completes that picture with a portable data layer that declares data separately from code, resolves it across heterogeneous sources, and reuses unchanged data through a metadata-keyed HPC-local cache, providing the workflow-facing data abstraction needed for portable notebook execution.

VII. Discussion and Future Work

The data layer establishes input-side identity: it materializes the same declared inputs at stable paths and gates reuse on whether they still match. The layer's mechanisms are not inherently notebook-specific, but the abstraction is motivated by notebooks, where data-access and path assumptions are often embedded directly in cells. It does not establish output equivalence, and we do not claim bitwise reproducibility, which depends on software, hardware, and nondeterminism beyond the data layer. Reuse identity rests on source metadata such as size and modification time rather than content hashes, which keeps recognition cheap and works when metadata reflects content changes, as it normally does; `verify` mode adds checksums for cases where it may not.

Future work will broaden the data layer in several areas. *Credential-aware specifications* would extend the ambient-credential model with scoped tokens and site-specific identity mappings for private sources. *Intermediate and derived data* would extend items beyond inputs to outputs, lineage, and retention, so the layer can reuse what a workflow produces, not only what it consumes. *Staged and adaptive materialization* would let items declare when they are needed, whether at startup, in a later phase, or by streaming, and let the runtime choose among direct access, full or partial staging, and cached reuse based on dataset size, reuse, latency, and network. *Specification interoperability* would map data items and runtime records to standards such as CWL.

VIII. Conclusion

We presented a portable data layer that lets notebook workflows *declare* data requirements once and have them *resolved*, materialized, validated, and *reused* across heterogeneous HPC systems without modifying notebook code. The layer separates notebook-facing paths from physical storage locations, supports heterogeneous and fallback sources, and makes data identity explicit for reliable reuse across runs.

Cross-site experiments show that the same notebooks and backpacks execute unchanged across three HPC systems spanning two scheduler families and different storage architectures. By moving data access, staging, validation, and reuse out of notebook cells and into a declarative layer, portable and repeatable execution becomes a property of the workflow rather than a site-specific engineering task.

Availability

Floability is available at https://floability.github.io. The example backpacks and evaluation scripts are available at https://github.com/saifulislampi/floability-data-experiments.

Acknowledgment

This project is supported by the U.S. National Science Foundation under grant OAC-2411436. This work used Anvil (Purdue) and Stampede3 (TACC) through ACCESS allocation CIS250350, which is supported by NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296. The authors used Claude and ChatGPT for grammar corrections and to help draft the analysis code for Figures 7 and 8; all generated material was reviewed and verified by the authors, who take full responsibility for the work.

REFERENCES

[1] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks–a publishing format for reproducible computational workflows," in *Positioning and power in academic publishing: Players, agents and agendas*. IOS press, 2016, pp. 87–90.

[2] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*. IEEE, 2019, pp. 507–517.

[3] S. Samuel and D. Mietchen, "Computational reproducibility of jupyter notebooks from biomedical publications," *GigaScience*, vol. 13, p. giad113, 2024.

[4] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *SciPy*, 2015.

[5] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster *et al.*, "Parsl: Pervasive parallel programming in python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 25–36.

[6] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[7] B. Sly-Delgado, T. S. Phung, C. Thomas, D. Simonetti, A. Hennessee, B. Tovar, and D. Thain, "TaskVine: Managing In-Cluster Storage for High-Throughput Data Intensive Workflows," in *18th Workshop on Workflows in Support of Large-Scale Science*, 2023.

[8] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[9] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2017, pp. 1–10.

[10] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. De Supinski, and S. Futral, "The spack package manager: bringing order to hpc software chaos," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2015, pp. 1–12.

[11] Anaconda, Inc., "conda-pack: Package conda environments for redistribution," 2018. [Online]. Available: https://conda.github.io/conda-pack/

[12] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky, "Xrootd/txnetfile: a highly scalable architecture for data access in the root environment," in *Proceedings of the 4th WSEAS International Conference on Telecommunications and Informatics*, vol. 1, 2005, pp. 156–162.

[13] Pelican Platform, "Pelicanfs: An fsspec implementation that integrates with the pelican platform," https://github.com/PelicanPlatform/pelicanfs, accessed: 2025-05-26.

[14] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett *et al.*, "Software as a service for data scientists," *Communications of the ACM*, vol. 55, no. 2, pp. 81–88, 2012.

[15] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble *et al.*, "Methods included: standardizing computational reuse and portability with the common workflow language," *Communications of the ACM*, vol. 65, no. 6, pp. 54–63, 2022.

[16] C. Goble, S. Cohen-Boulakia, S. Soiland-Reyes, D. Garijo, Y. Gil, M. R. Crusoe, K. Peters, and D. Schober, "Fair computational workflows," *Data Intelligence*, vol. 2, no. 1-2, pp. 108–121, 2020.

[17] D. Weitzel, M. Zvada, I. Vukotic, R. Gardner, B. Bockelman, M. Rynge, E. F. Hernandez, B. Lin, and M. Selmeci, "Stashcache: a distributed caching federation for the open science grid," in *Practice and Experience in Advanced Research Computing 2019: Rise of the Machines (learning)*. Association for Computing Machinery, 2019, pp. 1–7.

[18] M. S. Islam, T. Azaz, R. Ahmad, A. S. M. Shahadat Hossain, F. Baig, S. Wang, K. Lannon, T. Malik, and D. Thain, "Backpacks for notebooks: Enabling containerized notebook workflows in distributed environments," in *2025 IEEE International Conference on eScience (eScience)*, 2025, pp. 169–177.

[19] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 74–81.

[20] S. Eustace, "Poetry: Python packaging and dependency management made easy," 2018. [Online]. Available: https://python-poetry.org/

[21] M. Geimer, K. Hoste, and R. McLay, "Modern scientific software management using easybuild and lmod," in *2014 First International Workshop on HPC User Support Tools*. IEEE, 2014, pp. 41–51.

[22] 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.

[23] J. Köster and S. Rahmann, "Snakemake—a scalable bioinformatics workflow engine," *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012.

[24] nteract contributors, "Papermill: Parameterize and execute jupyter notebooks," 2018. [Online]. Available: https://github.com/nteract/papermill

[25] A. Rule, A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M. H. Nguyen, S. B. Rosenthal, F. Pérez *et al.*, "Ten simple rules for reproducible research in jupyter notebooks," *arXiv preprint arXiv:1810.08055*, 2018.

[26] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "Reprozip: Computational reproducibility with ease," in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 2085–2088.

[27] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire, "noworkflow: capturing and analyzing provenance of scripts," in *Provenance and Annotation of Data and Processes: 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014. Revised Selected Papers 5*. Springer, 2015, pp. 71–83.

[28] D. H. Ton That, G. Fils, Z. Yuan, and T. Malik, "Sciunits: Reusable research objects," in *IEEE eScience*, Auckland, New Zealand, 2017.

[29] R. Ahmad, N. N. Manne, and T. Malik, "Reproducible notebook containers using application virtualization," in *2022 IEEE 18th International Conference on e-Science (e-Science)*. IEEE, 2022, pp. 1–10.

[30] "FLINC," 2022, [Online; accessed 19-May-2025]. [Online]. Available: https://github.com/depaul-dice/Flinc

[31] M. Barisits, T. Beermann, F. Berghaus, B. Bockelman, J. Bogado, D. Cameron, D. Christidis, D. Ciangottini, G. Dimitrov, M. Elsing *et al.*, "Rucio: Scientific data management," *Computing and Software for Big Science*, vol. 3, no. 1, p. 11, 2019.

[32] A. Withers, B. Bockelman, D. Weitzel, D. Brown, J. Gaynor, J. Basney, T. Tannenbaum, and Z. Miller, "SciTokens: Capability-based secure access to remote scientific data," in *Proceedings of the Practice and Experience in Advanced Research Computing 2018 (PEARC '18)*, 2018.

[33] F. Andrijauskas, D. Weitzel, and F. Wuerthwein, "Open science data federation-operation and monitoring," in *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, 2024, pp. 1–5.

[34] Y. O. Halchenko, K. Meyer, B. Poldrack, D. S. Solanky, A. S. Wagner, J. Gors, D. MacFarlane, D. Pustina, V. Sochat, S. S. Ghosh, C. Mönch, C. J. Markiewicz *et al.*, "DataLad: distributed system for joint management of code, data, and their relationship," *Journal of Open Source Software*, vol. 6, no. 63, p. 3262, 2021.

[35] T. Kosar and M. Livny, "Stork: Making data placement a first class citizen in the grid," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE, 2004, pp. 342–349.