Backpacks for Notebooks: Enabling Containerized Notebook Workflows in Distributed Environments

Md Saiful Islam*, Talha Azaz†, Raza Ahmad†, A S M Shahadat Hossain‡, Furqan Baig§, Shaowen Wang§, Kevin Lannon*, Tanu Malik†‡, Douglas Thain*

*University of Notre Dame, Notre Dame, IN, USA; †DePaul University, Chicago, IL, USA;
‡University of Missouri, Columbia, MO, USA; §University of Illinois Urbana–Champaign, Champaign, IL, USA

mislam5@nd.edu, tazaz, raza.ahmad@depaul.edu, ahr8v@missouri.edu,
fbaig, shaowen@illinois.edu, klannon@nd.edu, tanu@missouri.edu, dthain@nd.edu

***Abstract*—Notebooks have become widely adopted in the scientific community due to their interactive interface and ease of sharing. However, using notebooks to execute large-scale scientific workflows remains challenging. Scientific workflows are typically distributed and require resource provisioning and data management prior to execution. Because notebooks do not natively embed workflow specifications, users often resort to inserting custom configuration steps directly within notebook cells to enable provisioning. This practice undermines reproducibility, as the same notebook may not run consistently across different cluster environments. In this paper, we introduce the concept of a notebook backpack—a companion specification that captures the embedded workflow along with all relevant configuration elements. We describe how notebook tracing can be leveraged to automatically populate the backpack. We then describe an integrated tool that provisions a backpack on distributed resources. Using real-world case studies, we demonstrate that the backpack abstraction enables minimal modification of the notebook, portable execution, and cross-site reproducibility of notebook-based workflows on HPC clusters without significantly increasing notebook execution time.**

Fig. 1: Portability challenges in notebook workflows

A notebook workflow alone cannot execute successfully without its supporting environment. Simply sharing the notebook or script is not enough—we must also recreate the same runtime environment, ensure access to the necessary data, and provision appropriate computing resources to achieve reproducible execution.

I. Introduction

Notebooks [1] have become a hallmark of interactive computing, increasingly adopted as the primary user interface for scientific programming. While it is relatively easy to initiate and develop a local analysis in a notebook, scaling this analysis beyond a single machine presents significant challenges. When the need for scaling arises, users may turn to classical workflow systems—such as Pegasus [2], Nextflow [3], or TaskVine [4]—which are designed to orchestrate complex workflows across distributed environments. However, these systems are not always designed for seamless integration with the interactive notebook paradigm, often compelling users to transition their work outside the notebook environment. Libraries like Dask [5] and Parsl [6] enable distributed execution from within notebooks, but require code modifications and site-specific configuration. This tweaking undermines true portability and consistent reproducibility, often leading to subtle errors, underperformance, and a considerable burden on the user to manage operational complexities [7].

Despite their popularity, reproducibility remains a major challenge for notebooks, largely due to missing dependencies and data availability [8], [9]. While notebook files unify code and metadata, they typically omit environment specifications, reducing portability across heterogeneous systems [10]. As shown in Figure 1, code alone is insufficient—environment, data, and resource specifications are essential for reproducibility. This issue is exacerbated in distributed workflows. For instance, the notebook in Figure 2 performs batch matrix multiplication—a common task in machine learning. In this example, only the matrix multiplication performed by `multiply_pair` is computationally expensive—at $O(n^3)$—and embarrassingly parallel, as each input can be processed independently. To run such computations in parallel, a workflow management system is required. Figure 3 illustrates how this can be achieved with TaskVine, dispatching each matrix operation to remote workers. However, such implementations require replicating all dependencies—at minimum, the correct version of `numpy`—on every worker node [11], and users must provision compatible workers to the cluster. This setup is non-trivial, as each cluster typically has its own approach to launching and managing workers through site-specific schedulers, containers, or protocols.

Furthermore, any data required by the workflow must be staged appropriately, with valid and accessible paths. Hardcod-

```python
def multiply_pair(A, B):
    import numpy as np
    A_np = np.array(A, dtype=float)
    B_np = np.array(B, dtype=float)
    C_np = A_np @ B_np
    return C_np.tolist()
```

```python
def read_matrices_from_csv(filename):
    pairs = []
    with open(filename, 'r') as f:
        for row in csv.reader(f):
            vals = [float(v) for v in row]
            i = 0
            r1, c1 = int(vals[i]), int(vals[i+1])
            i += 2
            A = []
            for x in range(r1):
                A.append(vals[i+x*c1:i+(x+1)*c1])
            i += r1*c1
            r2, c2 = int(vals[i]), int(vals[i+1])
            i += 2
            B = []
            for x in range(r2):
                B.append(vals[i+x*c2:i+(x+1)*c2])

            pairs.append((A, B))
    return pairs
```

```python
import csv
matrix_pairs = read_matrices_from_csv('matrices.csv')
results = []

for i, (A, B) in enumerate(matrix_pairs):
    C = multiply_pair(A, B)
    results.append(C)
    print(f"Pair {i}: result size {len(C)}x{len(C[0])}")
```

Fig. 2: Simple Batch Matrix Multiplication Workflow
This notebook performs batch matrix multiplication using input pairs read from a CSV file. Each row in the file encodes two matrices, which are multiplied using a NumPy-backed function `multiply_pair`*. The workload is embarrassingly parallel, as each matrix pair is processed independently.*

```python
def multiply_pair(A, B):
    # same as before
```

```python
def read_matrices_from_csv(filename):
    # same as before
```

```python
import csv
import ndcctools.taskvine as vine

matrix_pairs = read_matrices_from_csv('matrices.csv')

m = vine.Manager([9123, 9150], name="matrix-tv")

tasks_map = {}
results = [None] * len(matrix_pairs)

# Submit tasks to TaskVine
for i, (A, B) in enumerate(matrix_pairs):
    task = vine.PythonTask(multiply_pair, A, B)
    t_id = m.submit(task)
    tasks_map[t_id] = i
    print(f"[manager] Submitted task for pair {i}")

# Wait for results
print("[manager] Waiting for tasks to complete...")
while not m.empty():
    done_task = m.wait(5)
    if done_task:
        idx = tasks_map[done_task.id]
        if done_task.successful():
            results[idx] = done_task.output
            print(f"[manager] Pair {idx} done")
        else:
            # Handle Failure
```

Fig. 3: Distributed Matrix Multiplication Using TaskVine
This notebook transforms a simple call to `multiply_pair` *into a distributed Python task, which is submitted to the TaskVine manager (outlined in red). Each matrix pair is independently dispatched to remote workers, allowing parallel execution and efficient resource use across HPC nodes.*

ing data paths in the notebook creates site-specific assumptions that hinder portability and reproducibility. Instead, these details should be captured separately in an external specification that accompanies the notebook. Such a specification can define the software environment, data sources, and resource requirements in a portable, declarative way, enabling the same notebook to run unmodified across diverse systems.

In this paper, we introduce the concept of a backpack for notebooks—a structured companion specification that captures the software, data, and resource dependencies needed for executing notebook workflows on distributed systems. Since the notebook file alone is incomplete [10], the backpack acts as a lightweight, portable container with all necessary context explicitly described. We demonstrate that backpacks address critical challenges such as environment replication, data staging, and resource provisioning. To support this, we present *Floability*, a tool that implements the backpack specification and automates notebook execution on clusters, enabling consistent and reproducible workflows across diverse systems.

II. Reproducibility Challenges in Distributed Notebook Workflows

Users typically develop workflows incrementally, installing packages and adding data on the fly. While this approach works locally, moving to HPC or cloud environments reveals hidden portability and reproducibility issues.

Consider the matrix multiplication notebooks in Figures 2 and 3. One runs locally, while the other uses TaskVine to distribute it across remote workers. The TaskVine version improves scalability but exposes key challenges:

1) **Managing Software Dependencies Across Distributed Nodes:** Notebook workflows often install packages incrementally using commands like `pip install`, which only affect the local environment. In Figure 2, matrix multiplication runs locally with `numpy`; in Figure 3, the same function is offloaded to TaskVine workers, which also require the correct version of `numpy`. Installing it in the notebook has no effect on worker nodes. Instead of duplicating the manager's entire environment—which

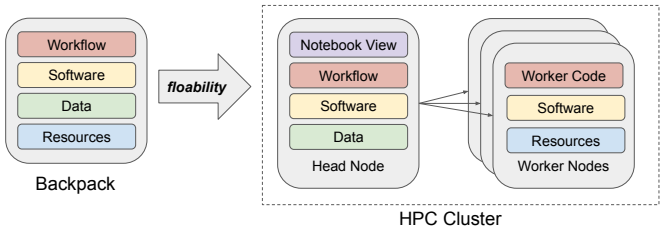

Fig. 4: Deploying a backpack with Floability
Floability consumes a backpack and launches a manager on the HPC head node and spawns workers on compute nodes, scaling dynamically as needed.

may contain large, unused libraries—workers should be provisioned with minimal, task-specific environments for efficiency and portability.

2) **Proactive Data Management and Verification:** Distributed workflows often process substantial amounts of data, requiring alignment of computational resources and data availability. If datasets are missing, inaccessible, or corrupted mid-execution, it can waste computation and cause workflow failure. For example, in Figure 3, if the required CSV is missing or unreadable when needed hours into execution, the workflow may stall. To prevent this, datasets should be proactively discovered, verified, and documented—including their source, access method, and integrity—before execution.
3) **Accurate Resource Specification for Distributed Execution:** Distributed workflows require explicit provisioning of resources (e.g., CPU, RAM, GPUs) before execution. Under-provisioning causes failures, while over-provisioning wastes resources. Accurately documenting and specifying each worker's needs is critical for efficient execution.

These challenges show how incremental notebook development complicates portability, reproducibility, and scalability in distributed settings. The next section introduces *backpacks*—a structured way to address these issues and enable seamless execution across systems.

III. Backpacks

A backpack encapsulates a notebook workflow with structured specifications of all its dependencies—data, software, and computational resources—needed for seamless execution across diverse environments. Traditionally, workflows are shared with minimal documentation, often just the notebook. In contrast, well-documented workflows specify all required software, datasets, and resources. Backpacks formalize this best practice, providing documentation that is both human-readable and machine-parseable, enabling Floability to reproduce and deploy workflows across heterogeneous HPC clusters with minimal manual intervention.

The backpack design directly addresses the three main challenges discussed in Section II: (1) software dependencies are explicitly separated for manager and worker nodes; (2) data requirements are proactively specified and verified; and (3) resource needs are precisely documented. Each backpack consists of four primary components:

A. Workflow Specification

The workflow component defines the computational logic, typically as a Jupyter notebook or script. To ensure portability across different HPC environments, workflows must adhere to specific guidelines:

- Use environment variables or relative paths instead of absolute file paths
- Separate computation logic from resource allocation logic
- Avoid site-specific configurations or resource identifiers

B. Software Dependencies

The software specification explicitly defines all dependencies required by the workflow. This component uses YAML format similar to Conda's `environment.yml` and includes:

- **Manager Environment** (`environment.yml`): Specifies the complete software stack required on the manager node for workflow orchestration.
- **Worker Environment** (`worker_environment.yml`): Defines minimal, task-specific dependencies for worker nodes. If omitted, workers default to the manager's environment. This separation enables efficient resource utilization, as demonstrated in the matrix multiplication example (Figure 3), where workers only need `numpy` rather than the manager's full environment.
- **Custom Installation Scripts**: Optional bash scripts for software that cannot be installed through standard package managers, such as custom-compiled binaries or specialized libraries.

C. Data Specification

The data component (`data.yml`) defines how datasets should be retrieved, stored, and verified, ensuring proactive data management. Each data entry specifies:

- **Name**: Descriptive identifier (e.g., `training_set`)
- **Source**: URL or filesystem path with retrieval method
- **Target Location**: Destination within the execution environment
- **Checksum**: MD5 hash for integrity verification
- **Post-Fetch Operations**: Additional processing (e.g., decompression)

This specification can reference large repositories such as OSDF [12] or XRootD [13]. It helps prevent failures where missing data stalls workflows by recording identifiers and access methods so workers can retrieve data without duplication.

D. Resource Requirements

The resource specification (`resource.yml`) provides a blueprint for resources needed for successful execution:

- **Worker Count and Type**: Number of parallel workers (e.g., TaskVine, Dask)

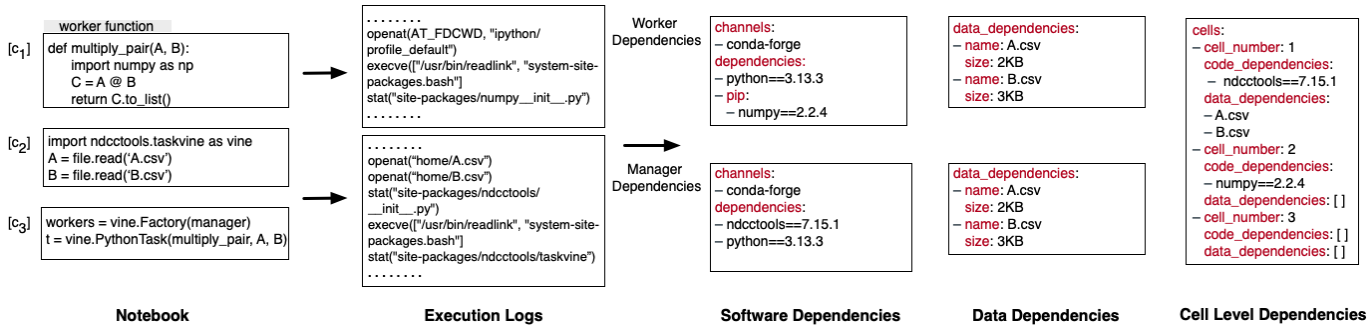

Fig. 5: End-to-End Process for Capturing Notebook Dependencies in Floability

The figure illustrates how Floability captures software and data dependencies from a notebook. It uses system call tracing to extract manager and worker software packages, which are then validated to produce an environment file with precise versions. Data dependencies are extracted from the logs and presented for manual verification.

- **Worker Specifications**: CPUs, memory, and disk space per worker
- **Access Credentials**: SSH keys or authentication tokens for resource access

As illustrated in Figure 4, the backpack abstraction transforms the incomplete notebook workflow shown in Figure 1 into a complete, portable package that can execute consistently across different HPC clusters with minimal user intervention.

IV. Generating Backpack Dependencies

A Floability backpack comprises code, data, workflow, and compute requirements, all of which may already exist in the user's environment alongside the notebook. This section explains how we can automate software and data dependency extraction. Figure 5 shows the end-to-end process.

A. Generating Software Dependencies

Generating notebook execution log: We generate provenance logs for notebook and worker processes using `strace` [14], which records every system call—including all files accessed during execution. Separate logs are collected for the manager and each worker. By tracing the entire workflow execution, we capture all libraries, binaries, and data files referenced during execution.

Extracting software dependencies: Whenever the notebook imports a Python package, the log records the full path of the accessed files. We parse this log to extract the package set, capturing each name for validation in the next step.

Validating Software Details: Once dependencies are identified, Floability validates them to create a precise, reproducible environment specification. It detects the environment type (e.g., Conda or virtualenv) using markers like `sys.prefix` and `CONDA_PREFIX`. Active environments are queried with `conda list` or `pip list` to retrieve accurate package versions. System call logs are filtered and cross-referenced with installed packages, retaining only valid, installable ones. Conda and pip packages are separated, with channel info recorded as needed.

This process outputs two complete `environment.yml` files—one for the manager and one for the workers—listing the Python version, required packages, and channels. Unlike `pip freeze` or `conda env export`, which dump every installed package, Floability's audit records only packages actually used during execution.

B. Generating Data Dependencies

To generate data dependencies of a notebook program, we modify Python's *open* function to log all data files accessed during execution. These include files used by both manager and workers, distinguished by comparing their paths with execution logs from `strace`. The logged list may also contain configuration or settings files from certain packages and libraries, which are removed through post-processing based on path analysis. For example, files in *proc*, *sys*, or *site-packages* are non-data files and excluded. This does not produce a complete list of data dependencies—remote files accessed via protocols such as HTTP or XRootD are not captured—and instead provides a starting point for the user to complete manually.

C. Generating Dependencies at Cell Level

To provide fine-grained reproducibility and environment capture, we also generate software and data dependencies at the level of individual notebook cells. This is achieved by registering and invoking functions which are triggered before and after the execution of every cell. Through these trigger functions, we write special instructions to the execution log which mark the beginning and end of the cell execution.

By analyzing intervals between these markers in the provenance logs, we identify which files and Python packages are accessed by each cell. The resulting dependency data is compiled into a YAML specification for selective re-execution, debugging, or modular environment provisioning. Figure 5 shows this format on the right.

Another approach is to create a specialized kernel that performs static analysis of each cell's code. This has limitations: in distributed workflows, manager and worker code cannot

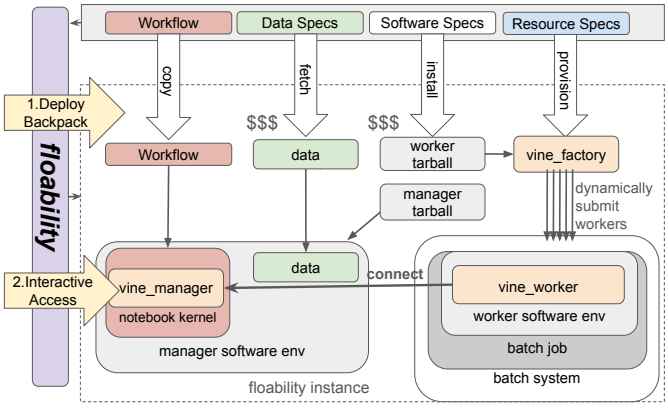

Fig. 6: Floability Architecture

Floability architecture consists of a manager process running on the head node and multiple worker processes running on compute nodes. The `vine_factory`, `vine_worker`, and `vine_manager` components are inherited from TaskVine. The factory process is responsible for launching and dynamically scaling workers in the cluster.

be distinguished by code analysis, and specialized kernels restrict users from running customized kernels. Similarly, data dependencies can be inferred by analyzing notebook execution logs with heuristics to separate data from code/configuration files. However, this is error-prone and less accurate than our approach, which dynamically modifies the behavior of the *open* function call.

V. Distributed Deployment with Floability

Floability implements the backpack specification and automates the execution of notebook-based scientific workflows on high-performance computing (HPC) clusters. It launches a Jupyter server on the cluster head node and dynamically spawns workers on compute nodes. Floability handles environment setup, resource allocation, and workflow orchestration, allowing users to focus on science rather than infrastructure. By leveraging standardized backpacks, Floability ensures that workflows remain portable, reproducible, and scalable across diverse HPC sites.

A. Architecture and Workflow

As shown in Figure 6, Floability uses a distributed architecture to execute notebook workflows on HPC clusters. It consumes a backpack and creates a Floability instance, which includes a manager process, a factory process, multiple worker processes, a Jupyter notebook server, and a lifecycle process. It first fetches the data specified in the backpack's data specification. If any required file is missing or its checksum fails verification, `floability verify` aborts the run immediately and prints a concise remediation message, preventing wasted cluster time. Then, it creates a Conda environment using the backpack's software specification. Based on the resource specification, the factory process launches the workers in the given runtime environment.

The manager coordinates workflow execution, while the factory manages the worker lifecycle. The workers execute the tasks defined in the notebook, and their results are sent back to the manager for aggregation and final output. The lifecycle process monitors execution and cleans up resources once the workflow completes.

B. Underlying Infrastructure: TaskVine Integration

Floability is built on top of TaskVine, a distributed execution engine designed to coordinate large-scale, dynamic workloads across clusters. TaskVine provides the core infrastructure for managing remote task execution, enabling Floability to decouple workflow logic from the complexities of resource management and distribution.

A key component is `vine_factory`, which launches and manages worker processes on clusters. By default, it provisions standard Vine workers that connect back to a TaskVine manager in the notebook kernel, but it can also launch arbitrary worker processes, supporting heterogeneous workloads.

While Floability uses TaskVine as a reference backend, the core concept is a general manager-and-worker model for distributed execution. In this architecture, tasks are submitted directly from the notebook, with the manager process residing inside the notebook kernel and worker processes running on distributed nodes. This approach is agnostic to the underlying execution engine and can be extended to support a variety of distributed frameworks beyond TaskVine, providing flexibility while maintaining a consistent, portable workflow interface.

C. Floability CLI

Floability provides a command-line interface (CLI) for managing the full lifecycle of distributed notebook workflows, including packaging, auditing, data fetching, and execution.

Floability offers two execution modes. `floability run` starts a live Jupyter session for interactive, exploratory work. `floability execute` runs the notebook headlessly in batch mode, then returns the fully populated `.ipynb`. This lets users automate a workflow while still benefiting from the notebook's rich, shareable output.

Other key subcommands include:

- `floability audit` — audits notebook and worker execution locally to generate software and data specifications for the backpack
- `floability fetch` — retrieves input data as specified
- `floability verify` — checks the completeness and validity of a backpack

These CLI tools enable seamless transition from interactive development to reproducible, distributed execution across heterogeneous environments.

VI. Evaluation and Use Cases

We evaluate Floability by applying it to real-world scientific workflows that were originally developed on local workstations but intended for large-scale HPC clusters. Our

goal is to assess Floability's ability to provide portability, reproducibility, and ease of execution across heterogeneous computing environments and workflow systems.

We selected three representative applications from different scientific domains. These applications are available in the Floability Examples repository [15]. The workflows are:

1) **Distributed Image Convolution** (**DConv**): An image processing pipeline that performs tiled filtering using the `im2col + GEMM` approach. Implemented with native TaskVine.
2) **Climate Trend Analysis (CTrend):** A climate science workflow that analyzes long-term temperature trends using NOAA Global Summary of the Month data with Parsl and TaskVine.
3) **CMS DV5 Analysis** (**DV5**): A high-energy physics analysis pipeline that processes collider data using Dask and the Coffea framework.

Each application differs in its task count, dataset size, software stack, and resource needs. Table II summarizes these static workload properties, including environment sizes and worker provisioning configurations. To reduce setup time and enable rapid experimentation, we use lightweight datasets and simplified configurations suitable for proof-of-concept deployment. This lets us focus on Floability's core capabilities: automatic environment capture, cross-site portability, and reproducible execution—rather than raw performance benchmarking.

We begin by auditing each application on a local machine to capture software and data dependencies. These backpacks are then manually refined with domain-specific adjustments before being executed on three different HPC clusters. The following sections present our evaluation in three parts: (1) local audit, (2) cross-site deployment, and (3) qualitative observations.

A. Auditing on Local Run

To generate backpack dependencies, we ran each workflow locally using a reduced number of tasks and a single local worker. The notebook code was executed with both the manager and worker processes running on the same machine and audited using *strace*, as described in Section IV. Table I reports the average of three runs per application, comparing plain notebook execution, execution while generating the backpack, and execution under an interactive application-virtualization (AV) method like FLINC [16] which runs the notebooks and audits its execution into a container-like package. The results demonstrate that time taken to generate the backpack by auditing the notebook execution is comparable to the interactive AV method. The slight overhead is attributed to the code executed to extract software and data dependencies from execution logs.

B. Running on HPC Clusters

After auditing and refining the application backpacks locally, we deployed each workflow to five HPC clusters—Notre Dame CRC (HTCondor) [17], Purdue Anvil (Slurm) [18], UT Stampede3 (Slurm) [19], and OSG [20] OSPool (HTCondor) [21] — as well as an AWS cluster (Slurm), without altering their backpack specifications. This cross-site evaluation tests Floability's ability to execute complex workflows portably under different resource managers and system policies.

TABLE I: Execution time (seconds) for plain notebook execution (NB Only), interactive application virtualization (Int. AV), and backpack generation (Backpack Gen.). Final columns show captured dependencies.

App.	NB Only	Int. AV	Backpack Gen.	# Mgr. Deps.	# Wkr. Deps.
DV5	19.59	22.45	23.31	77	61
DConv	183.58	183.13	212.61	93	4
CTrend	29.00	38.72	43.32	88	54

Table III presents the runtime decomposition for each application, showing the cost of environment setup and notebook execution on each site. "First" runs include the one-time cost of building manager and worker environments, whereas "Repeat" runs reuse those cached environments.

Each application was executed using the `floability execute` command without modifying the backpack content across sites. While the runtimes varied between clusters due to infrastructure differences, all workflows completed successfully using the same specifications.

TABLE II: Static Properties of Applications Used in Cross-Site Evaluation (sizes in MB).

App.	Backpack Size	#Tasks	Mgr. Env Size	Wkr. Env Size	#Wkr
DConv	2.7	1024	564	82	5–10
CTrend	30	335	631	150	5–10
DV5	2.1	16	610	566	2–5

C. Qualitative Observations

During deployment across four HPC systems, we observed several site-specific differences that highlight both the challenges of cross-site portability and Floability's flexibility in addressing them.

Site-Specific Configurations: Different sites required varying batch job configurations. Stampede3 required explicit queue names and time limits in job submissions, while other sites had different defaults. OSPool differed more significantly, as we needed special permission from the OSPool administrators to open sockets on the access point node for manager–worker communication. Floability accommodated these variations through flexible command-line arguments and resource specifications. In addition, some clusters—such as Purdue Anvil—had limited access to `/tmp`, requiring us to redirect the scratch directory to a shared filesystem path, which can be configured through a Floability command-line option.

Storage System Variations: We observed significant performance differences based on storage architectures, partic-

TABLE III: Decomposition of application runtimes (seconds) across five HPC systems. "First" runs include cold-start overheads for environment build + data fetch, while "Repeat" runs reuse cached artifacts. Values are reported as mean ± SD over three executions of each run type.

	Metric	ND CRC	Purdue Anvil	UT Stampede3	AWS Cluster	OSPool
First Run (cold start)						
DConv	Total Runtime	292 ± 5	744 ± 54	684 ± 11	216 ± 33	897 ± 37
	Env Creation (M+W)	132 ± 1	516 ± 37	307 ± 16	107 ± 7	71 ± 6
	Env Extraction	26 ± 1	36 ± 3	116 ± 3	26 ± 1	319 ± 23
	Notebook Execution	116 ± 9	174 ± 13	229 ± 15	66 ± 20	496 ± 33
CTrend	Total Runtime	265 ± 2	1052 ± 53	405 ± 9	302 ± 35	1185±203
	Env Creation (M+W)	159 ± 16	763 ± 38	217 ± 15	133 ± 8	85 ± 13
	Env Extraction	30 ± 1	44 ± 2	97 ± 8	30 ± 1	352 ± 13
	Notebook Execution	57 ± 2	225 ± 11	75 ± 5	127 ± 37	694±213
DV5	Total Runtime	370 ± 6	1212 ± 15	827 ± 9	500 ± 5	455 ± 41
	Env Creation (M+W)	303 ± 3	973 ± 15	598 ± 25	242 ± 6	165 ± 11
	Env Extraction	28 ± 4	43 ± 2	105 ± 13	29 ± 1	72 ± 10
	Notebook Execution	25 ± 1	172 ± 2	103 ± 4	215 ± 4	205 ± 19
Repeat Run (cached software environment and data)						
DConv	Total Runtime	136 ± 21	255 ± 27	330 ± 12	88 ± 6	791 ± 35
	Env Creation (M+W)	0	0	0	0	0
	Env Extraction	29 ± 5	39 ± 1	116 ± 6	26 ± 1	303 ± 31
	Notebook Execution	98 ± 20	198 ± 26	171 ± 6	51 ± 6	476 ± 5
CTrend	Total Runtime	107 ± 3	304 ± 26	201 ± 6	216 ± 10	1146±333
	Env Creation (M+W)	0	0	0	0	0
	Env Extraction	29 ± 1	44 ± 1	90 ± 4	32 ± 3	335 ± 19
	Notebook Execution	64 ± 4	240 ± 27	101 ± 13	172 ± 12	694±345
DV5	Total Runtime	70 ± 2	281 ± 16	218 ± 6	89 ± 5	354 ± 53
	Env Creation (M+W)	0	0	0	0	0
	Env Extraction	29 ± 2	44 ± 2	83 ± 2	37 ± 6	69 ± 15
	Notebook Execution	28 ± 1	217 ± 18	115 ± 9	40 ± 8	273 ± 58

ularly during conda environment creation. At Notre Dame CRC, we utilized local storage on the login node for environment creation, resulting in substantially faster setup times. In contrast, Purdue Anvil required the use of shared filesystems for all operations, leading to the longest environment creation times across all sites. Stampede3 allowed local environment creation but required extraction to shared storage for worker access,causing higher extraction times. These differences underscore the value of understanding site-specific storage and leveraging hybrid local/shared setups when possible.

Data Availability: In our evaluation, all workflows either downloaded data from the internet or included small datasets directly in the backpack. However, for production workflows with large datasets, downloading data from the archival location is inefficient and often impractical. Real-world deployments would typically access data through shared storage systems like the Open Science Data Federation (OSDF) [12], [22] or require data to be pre-staged on the cluster's shared filesystem. Regardless of the source—backpack, shared storage, or remote URL—Floability verifies that all required data are present and correctly staged before execution begins.

Performance Considerations: The runtime variations and standard deviations in Table III reflect infrastructure differences rather than Floability overhead. ND CRC had the lowest variability, indicating predictable performance, while OSPool showed the highest due to its opportunistic resource model and lack of a shared file system, with workers retrieving data directly from remote sources. AWS was consistently stable with competitive setup times, whereas Purdue Anvil and Stampede3 showed moderate variation, likely from later ACCESS allocations. Across sites, queue policies, network configurations, and storage systems differed. Despite these differences, all workflows ran successfully using identical backpack specifications, with only minor command-line adjustments, demonstrating Floability's portability while allowing site-specific adaptations.

VII. Related Work

Research on notebook-based distributed workflows spans workflow portability, reproducibility, dependency tracking, and distributed execution. We examine existing approaches and identify gaps our backpack abstraction addresses.

Containerization and Environment Management: Container technologies provide different levels of isolation and portability. Full container solutions like Singularity/Apptainer [23] and Charliecloud [24] package entire operating systems,

achieving HPC portability with near bare-metal performance [25]. HPCCM-based approaches [26] and scalability studies [27] demonstrate container viability for HPC but focus on traditional applications. At the other end of the spectrum, environment-only tools like Conda-pack [28] and Poncho [11] create lightweight, relocatable environments without full OS isolation. Poetry [29] and Pipenv provide deterministic builds through lock files, while Spack [30] and EasyBuild [31] offer HPC-specific package management but require explicit build recipes. Our backpack approach occupies a middle ground—more comprehensive than environment definitions yet lighter than full containers.

Workflow Systems and Distributed Execution: Traditional workflow systems like Pegasus [2], Nextflow [3], Makeflow [32], and TaskVine [4] provide robust fault-tolerant execution across clusters but require expressing workflows in domain-specific languages. CWL [33] standardizes workflow descriptions for portability but loses notebook interactivity. From within notebooks, Dask [5] and Parsl [6] enable parallel execution through explicit API usage, requiring code modifications. Papermill [34] and nbconvert execute notebooks as monolithic units without distributed task support.

Notebook Reproducibility and Provenance: Several tools capture execution dependencies to enable reproducibility. ReproZip [35], noWorkflow [36], [37], FLINC [16], and Sciunit [38], [39] use system call tracing, AST analysis, and profiling to capture environments and provenance. While these tools excel at tracking execution and dependent files, they focus on provenance rather than distributed execution. Floability extends dependency tracking through `floability audit`, which uniquely separates manager and worker dependencies. Binder [40] and Repo2Docker [41] create shareable environments but do not address distributed workflow execution or the separation of manager and worker dependencies required for HPC deployment.

Our Contributions: The backpack abstraction bridges the gap between heavyweight containers and lightweight environment definitions. By automatically extracting dependencies with manager/worker separation, we create optimized, portable notebook workflows without code modification. This approach combines the benefits of containerization (isolation, reproducibility) with the efficiency of targeted dependency management, enabling seamless notebook execution from laptops to HPC clusters while preserving the interactive development paradigm.

VIII. Discussion and Future Work

In our evaluation, backpacks successfully captured and reproduced notebook workflows across heterogeneous HPC environments, demonstrating portability without code modification. However, several important challenges remain for achieving complete computational reproducibility.

Data Reproducibility and Privacy: While backpacks enable workflow portability, true reproducibility requires identical datasets. For large datasets, sharing is often infeasible due to size, ownership, or privacy constraints. Future work could explore sealed computations—where backpacks execute analyses on protected datasets without exposing raw data, requiring integration with secure computing frameworks.

Dynamic Resource Adaptation: Our current implementation requires manual resource specification. Since Floability already traces dependencies, extending this to automatically profile resource usage (CPU, memory, I/O) would enable intelligent resource allocation based on workflow characteristics and historical data.

Automated Concurrency Detection: Notebooks often contain implicit parallelism—cells that could execute concurrently but are serialized by the linear execution model. Developing analysis techniques to automatically identify independent cells and transform them into distributed tasks would improve performance without user intervention.

Enhanced Tooling: Floability could benefit from real-time monitoring dashboards within Jupyter, intelligent data staging that automatically selects between local caching and data federation services, and native support for multiple execution frameworks beyond TaskVine.

IX. Conclusion

We introduced the concept of backpacks for notebooks—a structured and portable specification that captures all dependencies for executing notebook workflows across distributed systems. By explicitly defining software environments, data, and resource requirements, backpacks enable scalable, portable, and reproducible notebook execution beyond a single machine.

Our implementation, Floability, demonstrates that notebook workflows can achieve portability and reproducibility across heterogeneous HPC environments without code modification. Evaluation on real scientific applications across four HPC systems showed that backpacks successfully abstract infrastructure differences while maintaining execution consistency.

The backpack abstraction represents a significant step toward making notebooks first-class citizens in HPC environments. By decoupling the interactive development experience from the complexities of distributed execution, we enable scientists to focus on their research while ensuring their computational methods remain portable, reproducible, and scalable.

Availability

Floability is open-source software, available at https://floability.github.io. The example backpacks used in our evaluation are hosted at https://github.com/floability/floability-examples.

Acknowledgment

We thank Ben Tovar (Research Software Engineer, University of Notre Dame) for his help with the Floability conda release and for providing the DV5 application.

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] 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.

[3] 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.

[4] R. F. da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, and M. Livny, "Toward Fine Grained Online Task Characteristics Estimation in Scientific Workflows," in *Workshop on Workflows in Support of Large Scale Science (WORKS)*, 2013, pp. 58–67, doi: 10.1145/2534248.2534254.

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

[6] 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.

[7] 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 writing and sharing computational analyses in jupyter notebooks," p. e1007007, 2019.

[8] 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.

[9] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's wrong with computational notebooks? pain points, needs, and design opportunities," in *Proceedings of the 2020 CHI conference on human factors in computing systems*, 2020, pp. 1–12.

[10] 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.

[11] B. Sly-Delgado, N. Locascio, D. Simonetti, B. Wiseman, B. Tovar, and D. Thain, "PONCHO: Dynamic Package Synthesis for Distributed and Serverless Python Applications," in *Workshop on High Performance Serverless Computing*, 2022, doi: 10.1145/3526060.3535459.

[12] OSG, "Open science data federation," 2015, https://doi.org/10.21231/0KVZ-VE57.

[13] A. Hanushevsky, A. Dorigo *et al.*, "Xrootd: Scalable architecture for data access," 2025, https://xrootd.slac.stanford.edu/.

[14] strace developers, "strace: Diagnostic, debugging and instructional userspace tracer for linux," https://strace.io, version 6.7, Accessed: 2024-05-25.

[15] Floability Team, "Floability examples," https://github.com/floability/floability-examples, 2025, accessed: 2025-05-26.

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

[17] U. of Notre Dame Center for Research Computing, "Center for research computing," https://crc.nd.edu, accessed: 2025-05-26.

[18] X. C. Song, P. Smith, R. Kalyanam, X. Zhu, E. Adams, K. Colby, P. Finnegan, E. Gough, E. Hillery, R. Irvine *et al.*, "Anvil-system architecture and experiences from deployment and early user operations," pp. 1–9, 2022.

[19] T. A. C. Center, "Stampede3 supercomputer at tacc," https://www.tacc.utexas.edu/systems/stampede3, accessed: 2025-05-26.

[20] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, I. Foster, R. Gardner, M. Wilde, A. Blatecky, J. McGee, and R. Quick, "The open science grid," *Journal of Physics: Conference Series*, vol. 78, p. 012057, 2007.

[21] OSG, "Ospool," 2006, https://doi.org/10.21231/906P-4D78.

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

[23] 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.

[24] 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.

[25] 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.

[26] S. Hoque, M. S. De Brito, A. Willner, O. Keil, and T. Magedanz, "Towards container orchestration in fog computing infrastructures," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2017, pp. 294–299.

[27] S. Abdulah, J. Ejarque, O. Marzouk, H. Ltaief, Y. Sun, M. G. Genton, R. M. Badia, and D. E. Keyes, "Portability and scalability evaluation of large-scale statistical modeling and prediction software through hpc-ready containers," *Future Generation Computer Systems*, vol. 161, pp. 248–258, 2024.

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

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

[30] 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.

[31] 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.

[32] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids," in *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012, doi: 10.1145/2443416.2443417.

[33] 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.

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

[35] 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.

[36] 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.

[37] J. F. N. Pimentel, V. Braganholo, L. Murta, and J. Freire, "Collecting and analyzing provenance on interactive notebooks: When {IPython} meets {noWorkflow}," in *7th USENIX workshop on the theory and practice of provenance (TaPP 15)*, 2015.

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

[39] T. Malik and et. al., "Sciunit," https://sciunit.run/, 2017, [Online; accessed 20-July-2021].

[40] B. Ragan-Kelley, C. Willing, F. Akici, D. Lippa, D. Niederhut, and M. Pacer, "Binder 2.0-reproducible, interactive, sharable environments for science at scale," in *Proceedings of the 17th python in science conference*. F. Akici, D. Lippa, D. Niederhut, and M. Pacer, eds., 2018, pp. 113–120.

[41] Project Jupyter, "jupyter/repo2docker: Turn repositories into jupyter-enabled docker images," 2017, [Online]. Available: https://github.com/jupyter/repo2docker. [Online]. Available: https://github.com/jupyter/repo2docker