

Efficiently Reproducing Distributed Workflows in Notebook-based Systems

Talha Azaz*, Raza Ahmad[†], Md Saiful Islam[‡], Douglas Thain[‡], Tanu Malik*

*University of Missouri, Columbia, MO, USA; [†]DePaul University, Chicago, IL, USA;

[‡]University of Notre Dame, Notre Dame, IN, USA;

tay2f, tanu@missouri.edu, raza.ahmad@depaul.edu, mislam5, dthain@nd.edu

Abstract—Notebooks provide an author-friendly environment for iterative development, modular execution, and easy sharing. Distributed workflows are increasingly being authored and executed in notebooks, yet sharing and reproducing them remains challenging. Even small code or parameter changes often force complete re-execution of the distributed workflow, limiting iterative development for such workloads. Current methods for improving notebook execution largely operate on single-node workflows, while optimization techniques for distributed workflows typically sacrifice reproducibility. We introduce NBReplay, an end-to-end system for efficient, reproducible execution of distributed workflows in notebooks. NBReplay consists of two kernels—audit and repeat—which are based on data-flow analysis across cells. The audit kernel avoids redundant computations through task level caching and incremental, cell-level checkpointing. The repeat kernel replays the cached tasks and reconstructs the checkpoints to enable partial re-execution of notebook cells that manage distributed workflow. Using real-world case studies, we demonstrate that NBReplay significantly improves notebook re-execution, and enables portable, cross-site reproducibility of notebook-based distributed workflows on HPC systems, while adding minimal overhead in terms of execution time and storage space.

Index Terms—Notebooks, Incremental Computation, Distributed Workflows, Workflow Systems, Reproducibility

I. INTRODUCTION

Computational notebooks [1] have seen widespread adoption due to their ability to enable interactive workflow development, cell-level execution, and easy dissemination of computational analyses across scientific disciplines [2]–[5]. Increasingly, distributed scientific workflows are prototyped and executed within notebook interfaces. For example, users may develop a notebook using libraries like Dask [6] (as shown in Figure 1), Parsl [7] or TaskVine [8], which enable specification of the distributed workflow from within notebooks. While such notebooks are often helpful for iterative development, they are difficult to share and reproduce across different cluster environments—often essential in collaborative scientific settings [9].

A core strength of notebooks is the ability to efficiently explore variations in parameters or code while preserving consistency of results. Consider the distributed workflow in Figure 1, where Cell 1 performs an expensive data-loading operation, Cell 2 defines a distributed map, group by, and aggregation, and a `risk_score` function to be computed over aggregated rows. Cell 3 triggers the execution of the

distributed computation on multiple nodes using `compute`, producing a `parquet` file output. In a typical scenario, a scientist executes all cells, inspects the results, and then varies downstream parameters such as the `risk_score` threshold, for example, changing from 10.0 to 8.0. If the kernel remains alive and Dask has retained intermediate data in memory, the scientist may only re-run Cell 3. However, the in-memory state of Dask is lost when the notebook is moved to another HPC cluster due to kernel restart, or when the notebook is shared with a collaborator. The collaborator inherits the notebook but none of its intermediate workflow state, forcing complete re-execution from raw data.

Although some systems offer coarse-grained checkpointing, existing notebook *savepoints* are manual and not designed for distributed workflows [10], [11]. They typically incur significant storage overhead by storing complete program states, and are not able to determine the redundant distributed computations which can be avoided when notebook cells are modified. In our running example, a checkpoint may allow skipping the expensive data-loading in Cell 1, but if the user modifies the `risk_score` formula in Cell 2, both Cells 2 and 3 must be re-run. Rerunning entire Cell 3 is wasteful, since only part of Cell 3’s distributed computation is logically affected by the edit — the distributed group-by aggregation remains valid, and only the `risk_score` computation should change. Current notebook systems do not have an efficient mechanism to checkpoint distributed computation within a cell or to reuse previous results for selective re-execution of the modified code [12].

In this paper, we address the dual challenge of repeating entire notebooks and entire distributed computation within a notebook by (i) checkpointing cell-level kernel state—including intermediate distributed workflow results, and (ii) selectively replaying only those tasks whose inputs or code have changed. The checkpoints of cells and the list of distributed tasks are created incrementally to save on checkpoint size. To our knowledge, no prior system exists which enables efficient re-execution through cell-level checkpointing and task-level caching, and end-to-end reproducibility of distributed workflows in notebook-based systems.

We present NBReplay, a system consisting of two notebook kernels: audit and repeat. The audit kernel performs automatic, *incremental checkpointing*: as users execute each

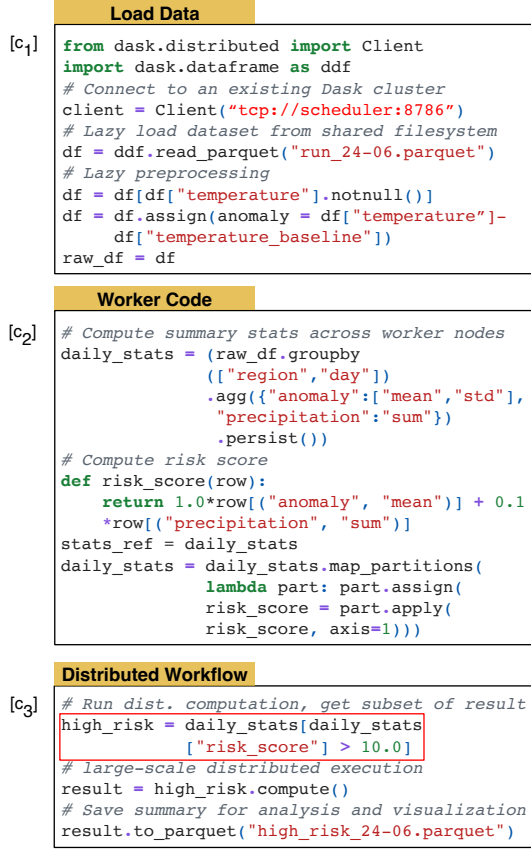


Fig. 1. Graphical illustration of a distributed workflow in a notebook where a manager creates worker threads to compute summary statistics. Execution takes place when `compute()` is called, resulting in a graph of tasks running in parallel, as shown in Figure 7. The red box highlights the part of code which is modified by the user when re-executing the notebook.

cell during iterative development, the kernel detects changes in the notebook’s kernel state and checkpoints only the variables and objects whose values have been updated relative to prior cells. This change detection is implemented by analyzing the variables in code and comparing them with other shared variables using a reverse index mapping of memory to variable names, efficiently determining which objects differ across cell boundaries. Incremental checkpointing is essential for controlling storage cost, as naively checkpointing full kernel state after every cell leads to substantial space consumption. We show that without incremental checkpoints, users must provision prohibitively large storage volumes to support reproducibility.

For cells that contain distributed workflows, the audit kernel interacts with the underlying distributed workflow engine to maintain a cache of distributed tasks and their intermediate outputs. It creates a hash-based structure so that the system can determine which tasks produce identical outputs and which must be invalidated when their code or input changes. When a notebook with distributed workflow is shared and executed in a new cluster environment, the repeat kernel is used to

reconstruct the full state of a notebook cell by composing the previously saved incremental checkpoints. It then selectively replays distributed computation within that cell by determining which tasks are invalidated due to code or dataflow changes in the notebook. Only the invalidated tasks are re-executed—all other tasks reuse memoized intermediate results stored in the cache as captured by the audit kernel. This enables partial re-execution of distributed workflows, ensuring that logically unaffected portions of a workflow are not recomputed.

The split-kernel design of NBReplay builds on our prior work on containerizing dependencies for single-node notebooks [13]. This design pattern has proven effective because it requires minimal changes to the notebook interface, preserves the interactivity of the notebook UI, and avoids intrusive modifications to user code. It is also compatible with emerging standardization efforts such as “backpack”-style artifacts that encapsulate notebook dependencies, data, and software environments [9]. The audit and repeat kernels automatically provide the metadata and state snapshots required to construct such portable bundles.

Our primary contributions in this paper are as follows:

- 1) Introducing partial re-execution through task caching and incremental checkpointing for notebook cells that contain both single-node and distributed workflows.
- 2) Designing a hash-based log and cache structure for efficiently determining the minimal set of tasks that must be re-executed.
- 3) Constructing a functional prototype of the audit and repeat kernels that implements and partial re-execution across notebook cells using distributed task memoization and automatic incremental checkpointing.

The rest of the paper is organized as follows. Section II explains the mechanics of distributed computation in notebook workflows. Section III gives an overview of the NBReplay system and its architecture. Section IV explains the checkpointing mechanism for saving the local notebook state. Section V details the process and methodology of partial re-execution in notebooks for distributed workflows. Section VI highlights our experiments and evaluates the effectiveness of NBReplay on our dataset of notebooks. We conclude in Section VIII with a summary of our work and future directions.

II. DISTRIBUTED WORKFLOWS IN NOTEBOOKS

We describe relevant processes that maintain execution state of distributed workflows within a notebook.

In general, notebooks are clients which users access and edit through a web browser. The code within each cell is executed on the server as part of a programming language-specific interactive *kernel* process, as shown in Figure 2. Because the kernel persists across cells, the state produced by one cell (variables, objects, functions, configurations) becomes the starting point for subsequent cells. The output of the cell-level code evaluations from the kernel process is relayed to the notebook client. Since the kernel executes the user code, the kernel is also aware of the computational environment and

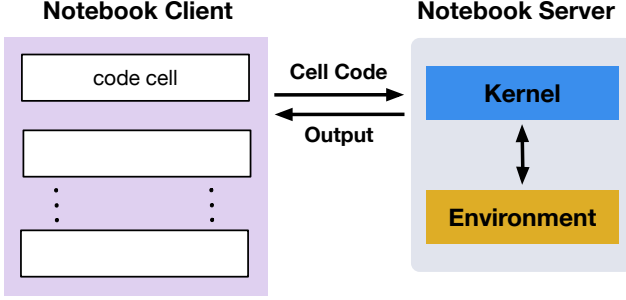


Fig. 2. In a notebook environment, the web-based notebook client interacts with the kernel process in the notebook server. The kernel executes the notebook code using its configured environment and returns the output.

the set of dependencies needed to run those code. Thus the notebook sends the user code one cell at a time, executes it using its configured environment, and returns the response of the executed code in the form of text and/or visualization, which is then displayed to the user through a web browser.

When distributed workflows are developed in notebooks, one or more cells of the notebook may run some of their code either in a process spawned by the kernel or a remote node, giving rise to a distributed workflow within the execution life cycle of that cell. Typically a distributed workflow comprises of a manager and a collection of distributed workers. The manager instructs workers to run some tasks, which are essentially processes running in parallel. The tasks read and write files from a shared file system as shown in Figure 3. Depending on the distributed workflow system, the manager may organize tasks into a directed acyclic graph (DAG) structure to manage inputs and outputs and determine if a task is in progress or waiting for other tasks to finish. DAG-based management of tasks also helps in efficient resource utilization. To enable the distributed workflow within the notebook cell, users may use popular libraries such as TaskVine [8], Dask [6], Parsl [7], which have Python extensions for use within notebooks.

The distributed workflow in notebooks may span multiple cells. The kernel maintains local Python objects representing the workflow (e.g., TaskVine task handles), while the manager process of the distributed system simultaneously maintains the remote state including partially completed tasks and cached intermediate results. Thus, the complete workflow state at any moment consists of both the kernel's local state and the distributed execution's state.

III. NBREPLAY: NOTEBOOK CHECKPOINTING AND PARTIAL RE-EXECUTION

NBReplay is a system consisting of two notebook kernels: *audit* and *repeat*. Both kernels are wrappers around base kernels e.g., the Python or Xeus-SQL kernel. The user uses the audit kernel to execute the notebook using the language interpreter. As the audit kernel executes the notebook, it automatically checkpoints after every cell. This checkpointing allows a user to go back in time and use the state of the notebook at time t instead of executing the other prerequisite

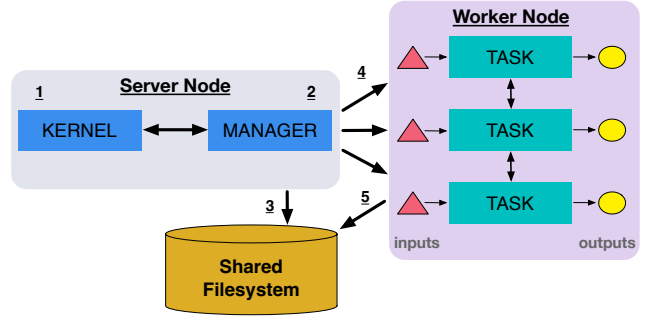


Fig. 3. The execution of a distributed notebook workflow is shown in numbered order. The kernel launches the manager which spawns several worker threads that run in parallel on the worker node.

cells from time $< t$. The audit kernel also intercepts the execution to determine all data and environment dependencies used by the notebook. In our work, we assume that the notebook workflows are deterministic in nature.

When the notebook is shared along with the checkpoints, data and environment dependencies, the user re-runs the notebook using the repeat kernel. The repeat kernel determines if there is a change in the notebook code cell. If there is no change, the repeat kernel bypasses execution of the cell and returns the stored output. If the repeat kernel determines that the cell's code has changed, this cell must be re-executed. For a changed cell, the repeat kernel reconstructs the checkpoint to execute the changed cell. Section IV describes how to efficiently create and reconstruct checkpoints for each of the kernels respectively.

When a distributed workflow is run within the notebook, the user still uses NBReplay's audit and repeat kernels as described above. However, during distributed execution, the audit kernel also determines if a cell issues any tasks. If so, the corresponding manager issuing the tasks is audited for task submissions. The audit kernel creates a log of all tasks that were submitted during execution. This log is available for sharing similar to checkpoints, and other data and environment dependencies. The repeat kernel uses the task log to optimize on re-execution when re-running the notebooks. Specifically, the repeat kernel bypasses execution of any task whose code or inputs have not changed. Section V describes how to intercept the distributed workflow manager and audit tasks for partial re-execution of distributed notebook workflows.

The combined checkpointing of notebook cells and auditing of distributed tasks allows NBReplay to improve scalability and reproducibility of notebooks, especially when users modify the notebook code and attempt to re-execute a shared notebook on a target cluster. Figure 4 shows the notebook's use of NBReplay's audit and repeat kernels to create checkpoints, task logs, data files, and environment, which need to be containerized to make it shareable and reproducible across platforms. We use application virtualization to capture all dependencies of the notebook by observing its execution [13]. In its audit phase, it generates a notebook container that includes

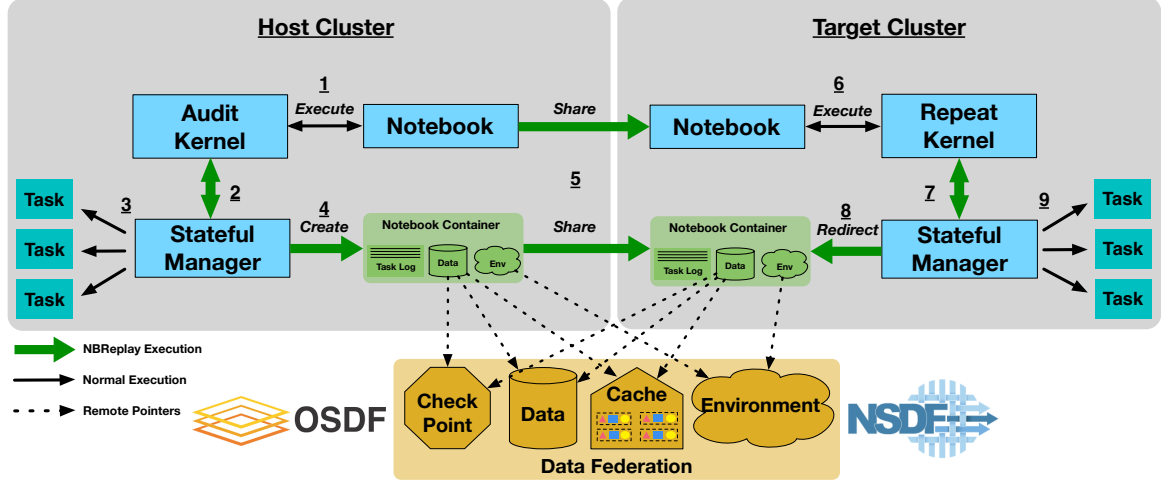


Fig. 4. An illustration of NBReplay architecture with its execution workflow numbered in-order. The audit kernel on host cluster executes a notebook containing distributed workflow through the stateful manager and creates a notebook container. The notebook and its container are then shared with a collaborator to be executed in a target cluster using the repeat kernel. The container contents may be quite large in size including checkpoints, data, cache and environment, and may be fetched dynamically from a remote data federation like OSDF or NSDF.

all necessary dependency files to reproduce the notebook. In the repeat phase, the notebook and its container are shared with a target cluster. The container includes (i) checkpoints to load the program state of each notebook cell, (ii) log and cache of tasks to selectively replay any modified cell, and (iii) environment to execute the cell code. Hence, the notebook is able to execute seamlessly in the target cluster, enabling reproducibility, as long as both host and target clusters are provisioned with similar hardware.

The size of these notebook containers can significantly increase especially for large-running workflows. To facilitate this, contents of the container can be hosted either locally within the cluster or on a remote host like a data federation, such as Open Science Data Federation [14] or the National Science Data Fabric [15]. Notebook containers can include remote pointers to these data sources and fetch them upon request, as shown in Figure 4. Integration with such data hosts is outside the scope of this work.

IV. CHECKPOINTING IN NOTEBOOKS

The audit kernel creates a checkpoint after each cell execution. To create a checkpoint, the kernel could, in principle, capture its own state using an OS-level checkpointing mechanism such as Checkpoint/Restart in Userspace (CRIU), which snapshots the entire process, including memory, file descriptors, threads, and other OS-managed resources. While OS-level checkpointing is programming-language agnostic, it captures a substantial low-level state that is unnecessary for reproducing notebook execution and is expensive to store and restore.

Instead, the NBReplay audit kernel performs application-level checkpointing of the Python session executing within the kernel process. Python maintains execution state in a global dictionary that maps names—such as variables, functions, and

imported modules—to runtime objects. Naively persisting this dictionary after each cell execution is impractical due to its size and the high frequency of updates. To reduce checkpoint overhead, NBReplay performs incremental checkpointing, persisting only those variables whose values may have changed as a result of executing a cell. As a first step, NBReplay applies static notebook slicing similar to NBSlicer [16], by parsing the cell’s abstract syntax tree (AST) to conservatively identify variables that are read or written in the cell. This static analysis bounds the set of candidate variables that may affect the notebook state.

Static slicing alone, however, is insufficient to capture runtime dependencies arising from mutation and shared references. Therefore, NBReplay augments static slicing with dynamic dependency tracking by observing object identities and shared memory references during execution. NBReplay tracks shared variables, variables that reference the same underlying runtime object, and uses this information to determine additional variables that must be checkpointed transitively.

To support this process, NBReplay maintains a reverse memory index, which maps memory references to all variables that alias them. Incremental checkpoints are constructed by combining statically identified variables with dynamically discovered shared dependencies. When a rollback is requested, the kernel reconstructs the notebook state by restoring the relevant subset of the globals dictionary from the corresponding incremental checkpoints.

To illustrate incremental checkpointing and shared-variable inference, Figure 5 presents a running example with two notebook cells. The second row of the figure shows the variables identified for each cell via abstract syntax tree (AST) parsing. This static analysis conservatively captures all variables declared or referenced within a cell. For example, in Cell 1 the parsed variables are `[Client, client, ddf,`

	Cell 1	Cell 2
Notebook Cells	<pre> 1 from dask.distributed import Client 2 import dask.dataframe as ddf 3 client = Client("tcp://scheduler:8786") 4 df = ddf.read_parquet("run_24-06.parquet") 5 df = df[df["temperature"].notnull()] 6 df = df.assign(anomaly = df["temperature"] - 7 df["temperature_baseline"]) raw_df = df </pre>	<pre> 1 daily_stats = (raw_df.groupby(["region", "day"]) .agg({"anomaly": ["mean", "std"], "precipitation": "sum"}).persist()) 2 def risk_score(row): 3 return 1.0*row[("anomaly", "mean")] + 0.1* 4 row[("precipitation", "sum")] stats_ref = daily_stats daily_stats = daily_stats.map_partitions(lambda part: part.assign(risk_score = part.apply(risk_score, axis=1))) </pre>
AST Variables	Client, client, ddf, df, raw_df	daily_stats, raw_df, risk_score, stats_ref
Shared Variables	df, raw_df	None
Variables to Checkpoint	Client, client, ddf, df, raw_df	daily_stats, risk_score, raw_df, df, stats_ref
Global State Space	0x7f8alc001a40 -> Client 0x7f8alc002c60 -> client 0x7f8alc001b50 -> ddf 0x7f8alc003d70 -> df, raw_df 0x7f8alc0060a0 -> daily_stats 0x7f8alc005f90 -> risk_score 0x7f8alc004e80 -> stats_ref	

Fig. 5. The internal states maintained by NBReplay to create checkpoints using data flow analysis, as illustrated by the first two cells of our example notebook in Figure 1. The AST of each cell is parsed to generate candidate variables, followed by dynamic analysis and reverse index lookups to determine the minimal set of variables which have changed since the last cell execution.

df], while Cell 2 yields a different set of variables.

Using the reverse index of memory locations (shown in the last row of Figure 5 for Cell 1), NBReplay identifies shared variables, i.e., variables that reference the same underlying runtime object. For instance, the variable *raw_df* shares a memory location with *df* due to the assignment in Line 7 of Cell 1 (shown in the third row of Figure 5), and is therefore marked as a shared variable.

In contrast, in Cell 2, although *state_ref* is initially assigned to *daily_stats*, they are not marked as shared variables. This is because *daily_stats* is subsequently reassigned in Line 4 of Cell 2, breaking the shared memory relationship. Since shared-variable determination is performed at cell-level granularity, the position of declarative and assignment statements affects the inferred sharing. For example, if Line 4 of Cell 2 were moved to a subsequent cell, then within Cell 2, NBReplay would infer *state_ref* and *daily_stats* as shared variables. However, NBReplay updates shared-variable information immediately upon reassignment, ensuring correctness of incremental checkpoints.

Checkpointing decisions are based on the union of variables identified via AST parsing and those inferred through shared-variable tracking. As a result, in Cell 2, even though *raw_df* is the only variable parsed from the AST, NBReplay also checkpoints *df*, since *raw_df* is shared with *df* through prior assignments.

Figure 6 further shows the metadata checkpointed for each variable at each cell. This metadata includes: (i) the serialized Python object when possible, (ii) the code that produced the object, and (iii) any environment or execution dependencies required for reconstruction. For example, *Client* is a Python class that can be serialized directly and has no prior code dependencies. In contrast, the *client* object represents a live Dask connection and cannot be serialized; such objects are not

```

'Client': {
  'obj': <class 'distributed.client.Client'>,
  # Serializable (class from module)
  'code': 'from dask.distributed import Client',
  'deps': [] },
'client': {
  'obj': None, # Non-serializable (Client
instance with network connections)
  'code': 'client = Client("tcp://
scheduler:8786")',
  'deps': ['Client'] },
'daily_stats': {
  'obj': <dask.dataframe.DataFrame: 80
partitions>, # Serializable (task graph)
  'code': 'daily_stats = (raw_df.groupby(...)',
  'deps': ['raw_df', 'risk_score']}

```

Fig. 6. A cross-section of the checkpoint schema generated by NBReplay includes the name, code, and the dependencies of each persisted object.

checkpointed. Instead, NBReplay reconstructs them during replay by re-executing the relevant initialization code. The variable *daily_stats*, a Dask DataFrame, is serializable, and its checkpoint includes both the object and the code expression that produced it, along with its transitive dependencies.

Finally, NBReplay performs checkpoint deduplication across cells. As shown in Figure 5, several variables (e.g., *df* and *raw_df*) appear in multiple checkpoints. Although these variables are included in successive checkpoints due to the possibility of change, their serialized representations may be identical. NBReplay detects and eliminates duplicate checkpoint entries across cells, significantly reducing storage overhead, as demonstrated in our experimental evaluation.

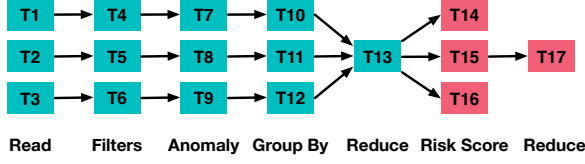


Fig. 7. Partial execution of task graph from example shown in Figure 1. Update in ‘risk_score’ re-executes tasks shown in red and returns cached results for unchanged tasks shown in green.

V. PARTIAL RE-EXECUTION

Checkpointing in notebooks by NBReplay enables the user to save the program state on the local system at any given notebook cell, and subsequently load the checkpoint during re-run to avoid redundant execution. However, it is unaware of any distributed computation launched on another machine by the notebook program. These distributed workloads are orchestrated by a distributed workflow manager which is typically stateless.

A. Stateless Distributed Workflows

Consider the Dask-based climate analysis workflow shown in Figure 1. Figure 7 illustrates an abstracted version of its task dependency graph, where Dask partitions the dataset into 3 chunks for parallel processing. The workflow spawns tasks that (1) load temperature data from a Parquet file (T1–T3) [Cell 1, Line 4], followed by tasks to (2) filter nulls values (T4–T6) [Cell 1, Line 5], and (3) computes anomalies (T7–T9) [Cell 1, Line 6]. Then it performs (4) parallel aggregation by region and day (T10–T12) [Cell 2, Line 1] and (6) a reduce operation to persist the intermediate aggregated result (T13) [Cell 2, Line 1]. It then runs tasks to (7) map partitions that apply the risk score function (T14–T16) [Cell 2, Line 4], and finally (8) a reduce operation that persists the results (T17) [Cell 2, Line 4].

During iterative development, users frequently modify such workflows. Adding a new weather station file creates a new parallel branch while leaving T1 through T12 unchanged. Modifying the risk score function affects only T14, T15, T16, T17 while leaving T1 through T13 unmodified. However, standard execution with Dask and DaskVine forces complete re-execution because neither framework maintains persistent state across runs. DaskVine acts as a stateless executor that submits tasks to TaskVine without awareness of previous executions, and TaskVine does not track task history or cache outputs between workflow invocations. When modifying the risk score function, only 3 of 16 tasks require re-execution, yet the stateless model re-executes all 16 tasks.

B. Stateful Execution with ReplayManager

We introduce a persistence layer that transforms TaskVine into a stateful workflow manager. The architecture adds a ReplayManager, which wraps the standard TaskVine manager and interposes on task submission and completion. The execution flow becomes: Dask → DaskVine → ReplayManager

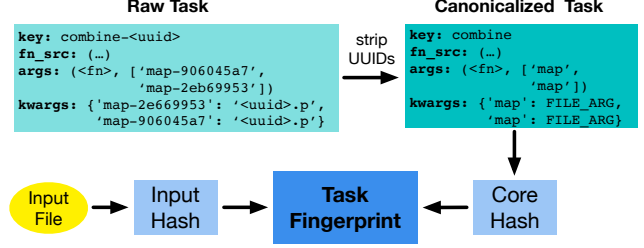


Fig. 8. Fingerprint of a Python function task is computed using its input file hash and a core hash of its function code and arguments. Canonicalization of tasks is done beforehand to remove non-deterministic identifiers. Non-deterministic identifiers are either removed or replaced with a uniform string like FILE_ARG.

→ TaskVine. ReplayManager maintains a transaction log containing task fingerprints and execution metadata, and a task cache containing the output files. When a task is submitted, ReplayManager computes a fingerprint of the task’s code and inputs, and uses it to query the transaction log. In the event of a cache hit, it retrieves and returns the cached output files, thereby avoiding redundant task execution. In the event of a cache miss, the task is sent to TaskVine for execution as usual and upon completion, its fingerprint and outputs are added to the cache. This is illustrated in Figure 7. In case of the risk score function modification, tasks T1 through T13 produce identical fingerprints resulting in a cache hit, while fingerprints for tasks T14 through T17 get changed and are hence executed with the modified risk score function.

Task fingerprinting captures the computational logic and input dependencies influencing task output. The strategy differs between command-based and Python function tasks. For command-based tasks, the fingerprint hashes the command string concatenated with SHA-256 content hashes of all declared input files. For each input, the system reads file contents and computes a hash. The complete representation is serialized to JSON with sorted keys and hashed with SHA-256. This provides strong guarantees when all inputs are declared.

For Python function tasks, fingerprinting proceeds in two stages. First, a core hash captures the function logic and arguments independent of input files. Function arguments are canonicalized to remove non-deterministic UUID suffixes that Dask injects (e.g., subgraphcallable-a1b2c3d4 normalizes to subgraphcallable). The system identifies user-defined functions in arguments, extracts their source code via introspection, and includes source code hashes. In Figure 7, tasks T14 through T16 include the risk_score function as an argument; modifying this function changes the source code hash, invalidating cached results. Similarly, function arguments are serialized and hashed as well. The core hash combines hashed arguments and user function source hashes. Finally, the core hash is combined with input file hashes. The two-staged fingerprinting is illustrated in Figure 8. Similar to command based tasks, for each user-declared input file, the system computes a SHA-256 hash. Input file hashes are sorted and appended to the core hash to compute the final fingerprint.

We encounter several challenges with the fingerprinting scheme. The first challenge is *undeclared dependencies* in which shell commands may reference files not declared as inputs assuming a shared file system (e.g., `bash process.sh data/*.csv` depends on both CSV files and `process.sh`). If `process.sh` is not declared, its modifications are invisible to fingerprinting, causing incorrect cache hits. The other challenge is *non-determinism in function arguments*. The system assumes deterministic functions where identical inputs produce identical outputs. However, functions depending on external state (e.g., timestamps and random numbers) as part of the closure violate this assumption, causing cached outputs to diverge from fresh execution. For example, Dask generates unique identifiers for intermediate results in each run (`finalize-abc123`, `subgraphcallable-xyz789`) that appear in task keys and arguments. When a Dask generated task function is deserialized, its byte representation differs in each run even though the task has not changed because it may reference a non-deterministic identifier. We canonicalize the tasks by removing the known non-deterministic formats through pattern matching. For intermediate files represented by a random number, we replace them by a uniform identifier, like `FILE_ARG` as shown in Figure 8. We note that these patterns are specific to each distributed framework and unrecognized formats may result in unnecessary cache misses.

VI. EXPERIMENTS AND EVALUATION

We evaluate NBReplay by applying it to real-world scientific notebook workflows that contain distributed computations and benefit from re-execution optimization. Our evaluation demonstrates that NBReplay can significantly reduce notebook re-execution time through incremental checkpointing and partial re-execution of distributed tasks, while maintaining reproducibility guarantees. We assess NBReplay’s effectiveness across three key dimensions: (1) checkpoint creation overhead during audit mode, (2) time savings during repeat execution with unchanged notebooks, and (3) selective task re-execution efficiency when notebooks are modified.

Computing Infrastructure: All experiments are conducted on a SLURM cluster with 4 compute nodes connected through a high-speed network. Each node runs a dual Intel Xeon Platinum 8259CL CPU at 2.50GHz and 4GB of memory. The cluster used NFS4 for shared file system access and XFS for local storage. The worker processes are configured with a minimum of 1 and maximum of 2 processes per node, allowing TaskVine to dynamically allocate resources based on workload demands.

Benchmark Description: Our evaluation dataset comprises 5 notebooks from diverse scientific domains, each demonstrating different computational characteristics, data dependencies, and distributed execution patterns. Table I provides a detailed characteristics of each notebook.

A. Methodology

Our experimental methodology¹ is designed to evaluate NBReplay across realistic notebook development and sharing scenarios. We compare notebook execution under five distinct conditions organized into three categories:

Baseline Execution (Without NBReplay):

- 1) **Initial Run:** Execute original notebook from top to bottom using standard Python kernel without any checkpointing or task caching. This is the first run of the notebook to establish a performance baseline.
- 2) **Modified Run:** Execute a modified version of the notebook using standard Python kernel. Modifications include either adding new input data files or changing the workflow code. This simulates iterative development where users make incremental changes to the code and must re-execute the entire workflow as a result.

Audit Phase (With NBReplay Checkpointing):

- 3) **Audit:** Execute original notebook using audit kernel with incremental checkpointing. The kernel creates checkpoints after each cell execution, persisting only those variables and objects that have changed since the previous cell, or that were used in this cell. The manager logs all completed tasks with their input fingerprints and caches all task outputs.

Repeat Phase (With NBReplay Restoration):

- 4) **Repeat Without Change:** Execute original notebook using repeat kernel. The kernel restores cell outputs from their checkpoints and replays cached task results for all tasks. This simulates sharing a notebook with a collaborator who wants to verify results without changes.
- 5) **Repeat With Change:** Execute the modified notebook as used in Baseline using repeat kernel. The kernel identifies which cells and tasks are affected by the modifications and selectively re-executes only the invalidated tasks. Unchanged cells are restored in-order from checkpoints and unchanged tasks are replayed from the cache. This simulates the iterative development process.

For each experimental condition above, we measure:

- **Execution time:** Total time from notebook execution start to completion, including all distributed execution.
- **Cache statistics:** Number of tasks submitted, cached, and re-executed during repeat phase with modifications.
- **Storage overhead:** Size of checkpoints before and after content-based deduplication, and task cache (transaction log and intermediate files).

To expect the workflow to be executed partially, we add additional input data files or modify the task code. For each notebook, we also report the expected tasks to be replayed from the cache and re-executed. We apply the following modifications to each notebook for partial re-execution evaluation.

- **DV5:** Added 1 additional ROOT file containing events.
- **MapReduce:** Modified `reduce` function to compute group counts in addition to total counts.

¹Our code is available at <https://github.com/radiant-systems-lab/NBReplay>

TABLE I
CHARACTERISTICS AND DESCRIPTION OF THE FIVE NOTEBOOKS IN THE DATASET.

Notebook	Cells	Unique Variables	Shared Variables	Dataset Size	Distributed Tasks	Description
DV5	11	60	7	952 KB	4	High energy physics analysis performing distributed event processing on CMS detector data using Dask and Coffea frameworks.
MapReduce	8	70	10	768 KB	13	Distributed word count pipeline creating 12 parallel map and combine tasks for word counting and 1 reduce task for aggregation.
DConv	7	27	5	16.4 MB	1024	Tile based image convolution applying 2 kernel functions to partition image tiles using Python Imaging Library. Processes 1024 tiles.
RAG	14	54	15	6.2 MB	4	Document chunking and indexing pipeline for BM25 based similarity search for books.
CTrend	12	80	7	27.44 MB	120	Climate trend analysis using temperature data. Performs data cleaning, anomaly calculation, and trend aggregation with statistical analysis.

- **DConv:** Added 1 new kernel matrix (Gaussian filter).
- **RAG:** Added 1 additional book (~500KB text).
- **CTrend:** Added 10 new weather station CSV files.

We assume that input data files do not change between successive runs, though new files may be added. Additionally, input data file names remain consistent between runs to enable accurate task fingerprinting and cache lookup. These assumptions align with common scientific workflow practices where input datasets are treated as immutable and new data is added as separate files. To obtain a stable fingerprint of the task for effective caching, we assume that a task function does not contain any non-deterministic value in its closure.

B. Results and Analysis

Figure 9 presents the execution times for each notebook under all experimental conditions. We observe that NBReplay reduces re-execution times significantly across all notebooks regardless of the size of notebook or number of cells in it. The magnitude of time saved depends on the notebook’s workflow complexity and patterns of modification.

Audit Overhead: Audit mode incurs an average of 18% overhead across notebooks ranging from 7-42% : DV5 (25%), MapReduce (1%), DConv (7%), RAG (16%), and CTrend (42%). Figure 9 shows these execution times. This increase in execution time includes the time to audit kernel process execution, maintaining the task cache, and checkpointing the notebook state after execution of each cell. We note that the audit overhead becomes negligible when compared to the performance gains achieved when repeating the notebook.

Repeat Execution Without Modifications: Repeat mode consists of (1) local kernel state reconstruction from incremental checkpoints, and (2) distributed task execution by replaying cached task results. This simulates sharing a notebook with a collaborator who wants to verify results without changes. Comparing repeat mode against baseline normal runs reveals significant speedup from eliminating redundant computation. NBReplay achieves an average performance gain of 18.4x ranging from 8x to 46.2x: DV5 (118.29s → 15.69s), MapReduce (38.79s → 2.78s), DConv (416s → 9s), RAG (140s → 12s), and CTrend (139s → 11s) by restoring kernel checkpoints and replaying cached task outputs. Since DConv has the most time spent in distributed computation, it achieves

the highest performance gains. These results reveal that using NBReplay we can improve the speed of reproducing distributed workflow by order of magnitude.

Repeat Execution With Modifications: Comparing repeat execution with modifications against baseline modified runs demonstrates NBReplay’s effectiveness in executing partially updated workflows. The average performance gain is 2.08x, ranging from 1.04x to 4.41x depending on the modification scope: DV5 (136s → 130.50s), MapReduce (41.23s → 27.67s), DConv (791s → 516s), RAG (145s → 76s), and CTrend (181s → 41s). Since adding 10 new weather stations in CTrend requires processing only 20 new tasks while caching 100 existing tasks we gain the most time savings.

Task Caching Effectiveness: The observed cache hit rates as shown in Figure 10 range from 50% to 92.3% across notebooks: DV5 (75%, 3 of 4 tasks cached), MapReduce (92.3%, 12 of 13 tasks cached), DConv (50%, 1024 of 2048 tasks cached), RAG (75%, 3 of 4 tasks cached), and CTrend (83.3%, 100 of 120 tasks cached). The average hit rate for all notebooks is 75.1%. This reveals that NBReplay’s effectiveness is determined by modification patterns and workflow structure. We identify two primary factors that govern caching behavior: adding new data versus modifying existing code, and the workflow DAG contains independent parallel tasks versus connected dependent tasks.

For workflows with group of tasks that are processed independent of other tasks (DV5, RAG, CTrend), adding new input files creates a separate branch of tasks that leaves existing tasks unaffected. Adding 10 CSV files to CTrend, for instance, creates 20 new tasks while successfully caching the 100 existing tasks, achieving an 83.3% hit rate. This pattern generalizes across workflows where each input file spawns independent processing pipelines.

Code modifications exhibit more complex behavior depending on modification location within the DAG. For example, modifying the reduce function in MapReduce re-executes only 1 of 13 tasks (92.3% hit rate) because the change occurs at the workflow’s terminal stage, leaving all upstream map tasks cacheable. In contrast, modifying early stage functions would cascade through all downstream tasks, eliminating caching benefits entirely. For instance, in DConv, updating the kernel matrix in convolution triggers a complete re-execution of

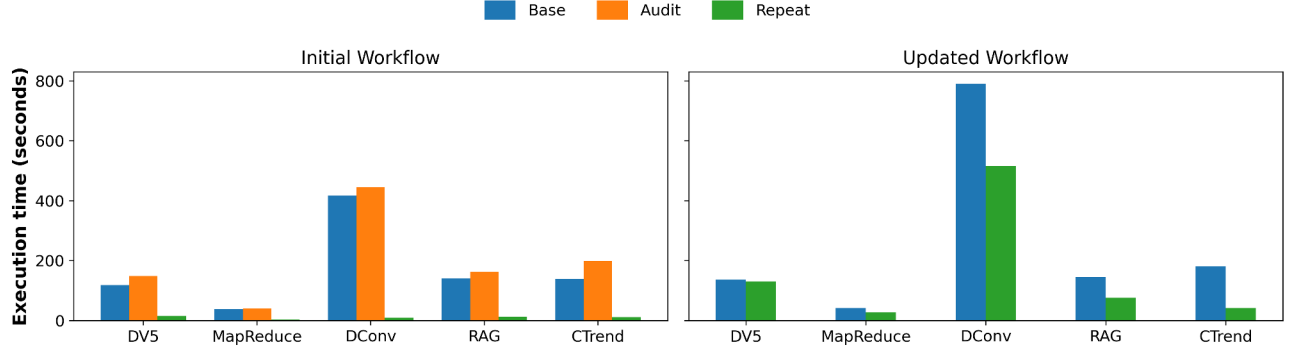


Fig. 9. Execution time in seconds for notebooks for both audit and repeat phases compared with the baseline.

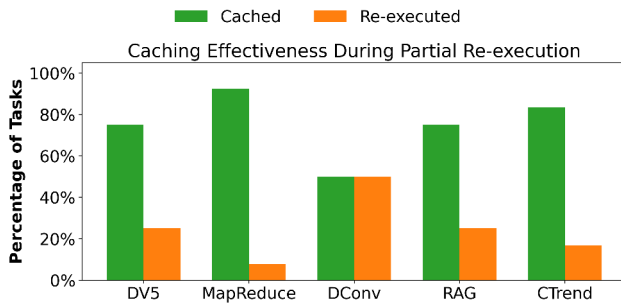


Fig. 10. Percentage of Cached and Re-executed tasks in the updated workflow.

workflow whereas extending the workflow with another kernel will achieve a 50% cache hit.

Storage Overhead: Table II reveals that NBReplay’s storage costs decomposes into two distinct components with different scaling behaviors: Python kernel state checkpoints and distributed task intermediate files. Incremental checkpointing with content-based deduplication keeps kernel state storage minimal, ranging from 0% to 57% reduction in checkpoint size with an average reduction of 27.3%. DConv achieves the highest deduplication savings at 57% (250 MB reduced to 108 MB) due to repeated references to large image arrays across multiple cells. The deduplication mechanism effectively eliminates redundant data referenced by multiple variables and stores it only once. As a result, an increase in the number of cells in a notebook does not linearly increase the additional cost of storing their corresponding checkpoints. In contrast, intermediate file storage scales with the DAG structure. Each of DConv’s 1024 convolution tasks produce an output image tile, generating substantial intermediate storage that persists for task cache replay. These intermediate files, while not directly visible to end users in the final notebook output, enable NBReplay’s selective re-execution capability by preserving task-level outputs.

TABLE II
STORAGE OVERHEAD BREAKDOWN FOR INCREMENTAL CHECKPOINTING: PRE-DEDUP(BEFORE DEDUPLICATION), POST-DEDUP (AFTER DEDUPLICATION), SAVING(%AGE SIZE REDUCTION), SIZE OF INTERMEDIATE FILES, %AGE OF TASKS CACHED.

Notebook	Checkpoint Size			Inter- mediate Files	Tasks Cached
	Pre- Dedup	Post- Dedup	Saving		
DV5	420 KB	264 KB	37%	112 KB	75%
MapReduce	380 KB	332 KB	13%	300 KB	92.3%
DConv	250 MB	108 MB	57%	159 MB	50%
RAG	116 MB	81 MB	30%	8.4 MB	75%
CTrend	1.9 MB	1.9 MB	0%	400 KB	83.3%
Average	74 MB	38 MB	27%	34 MB	75.12%

VII. RELATED WORK

Distributed workflows in Jupyter notebooks have become quite popular in various scientific disciplines. They can also be used to efficiently execute large-scale multi-user deployments where notebooks launch distributed jobs in HPC environments [17], [18], [19]. Similarly, Colonnelli et al. [20] extend the Jupyter model so each cell maps to a workflow step that can run in cloud or on HPC resources. This has become faster and convenient, especially with the rise of systems like Floability [9] which provide rapid, portable, and reproducible deployment of complex scientific workflows across a wide range of cyberinfrastructures.

Intermediate state preservation has been shown to reduce redundant re-execution and enhance interactive performance in traditional HPC environments [21] and interactive notebook system [12]. Kishu [10] is a system for versioning notebook session states through incremental checkpointing. ForkIt [22] creates a system to pursuing multiple exploration paths in a notebook using forking and backtracking. Li et al. propose ElasticNotebook [23] which enables live migration of notebook sessions across machines via lightweight state monitoring and graph-based optimization. Sato et al. [11] developed a computational notebook engine called Multiverse Notebook which allows user to time travel to any past state

through cell-wise checkpointing using the POSIX fork to keep track of the state of cells, besides developing their own garbage collection strategy. While these efforts have made advances in capturing dependencies and guiding users through execution lineage, they typically either address portability, reproducibility, or provenance without explicitly targeting the optimization of notebook re-execution performance. Additionally, they do not monitor any distributed computation taking place in a notebook cell. NBReplay distinguishes itself by unifying incremental checkpointing, content-based deduplication, and application virtualization to explicitly optimize the performance of notebook re-execution for distributed workflows.

For containerization, we use FLINC [24] which creates reproducible notebook containers by virtualizing the kernel process to create reusable research objects that can be repeated and shared for reproducibility across platforms. We choose FLINC over tools like NBSLICER [16] because of its flexibility to work in interactive notebook environments and its integration with NBReplay's split-kernel design for audit and repeat. To our knowledge, there is no prior end-to-end system which combines cell-level checkpointing and task-level caching for efficient re-execution and reproducibility of distributed workflows in notebook-based systems.

VIII. CONCLUSION

We introduced NBReplay which improves the efficiency and reproducibility of distributed workflows in notebook environments. To improve we employed the two mechanisms of checkpointing for individual cells, and maintaining a cache to store distributed tasks and their outputs. We learned that creating such an efficient system requires a developer to restrict to application-level checkpointing and using canonical task descriptions for fingerprinting. In future, we plan to investigate how such a system can be useful for data-intensive workflows that integrate with global data federations.

ACKNOWLEDGMENTS

This work is supported in part by the U.S. National Science Foundation under grant NSF 251657 and 2411436.

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] K. M. Mendez, L. Pritchard, S. N. Reinke, and D. I. Broadhurst, "Toward collaborative open data science in metabolomics using jupyter notebooks and cloud computing," *Metabolomics*, vol. 15, no. 10, p. 125, 2019.
- [3] J. Bascuñana, S. León, M. González-Miquel, E. González, and J. Ramírez, "Impact of jupyter notebook as a tool to enhance the learning process in chemical engineering modules," *Education for Chemical Engineers*, vol. 44, pp. 155–163, 2023.
- [4] R. Castilla and M. Peña, "Jupyter notebooks for the study of advanced topics in fluid mechanics," *Computer Applications in Engineering Education*, vol. 31, no. 4, pp. 1001–1013, 2023.
- [5] Y.-D. Choi, B. Roy, J. Nguyen, R. Ahmad, I. Maghami, A. Nassar, Z. Li, A. M. Castronova, T. Malik, S. Wang *et al.*, "Comparing containerization-based approaches for reproducible computational modeling of environmental systems," *Environmental Modelling & Software*, vol. 167, p. 105760, 2023.
- [6] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *SciPy*, 2015.
- [7] 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.
- [8] 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.
- [9] M. S. Islam, T. Azaz, R. Ahmad, A. S. M. S. Hossain, F. Baig, S. Wang, K. Lannon, T. Malik, and D. Thain, "Backpacks for notebooks: Enabling containerized notebook workflows in distributed environments," in *Proceedings of the 21st IEEE International Conference on e-Science (eScience '25)*, 2025, pp. 169–177.
- [10] Z. Li, S. Chockchowwat, R. Sahu, A. Sheth, and Y. Park, "Kishu: Time-traveling for computational notebooks," *arXiv preprint arXiv:2406.13856*, 2024.
- [11] S. Sato and T. Nakamaru, "Multiverse notebook: Shifting data scientists to time travelers," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 754–783, 2024.
- [12] H. Fang, S. Chockchowwat, H. Sundaram, and Y. Park, "Enhancing computational notebooks with code+data space versioning," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, ser. CHI '25. New York, NY, USA: Association for Computing Machinery, 2025.
- [13] R. Ahmad, N. N. Manne, and T. Malik, "Improving reproducibility of interactive notebooks using application virtualization," *Future Generation Computer Systems*, p. 108043, 2025.
- [14] 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*, ser. PEARC '24. New York, NY, USA: Association for Computing Machinery, 2024.
- [15] J. Luettgau, H. Martinez, P. Olaya, G. Scorzelli, G. Tarcea, J. Lofstead, C. Kirkpatrick, V. Pascucci, and M. Taufer, "Nsdf-services: Integrating networking, storage, and computing services into a testbed for democratization of data delivery," in *Proceedings of the 16th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC '23. New York, NY, USA: Association for Computing Machinery, 2024.
- [16] S. Shankar, S. Macke, S. Chasins, A. Head, and A. Parameswaran, "Bolt-on, compact, and rapid program slicing for notebooks," *Proceedings of the VLDB Endowment*, vol. 15, no. 13, pp. 4038–4047, 2022.
- [17] A. Zonca and R. S. Sinkovits, "Deploying jupyter notebooks at scale on xsede resources for science gateways and workshops," in *Proceedings of the Practice and Experience in Advanced Research Computing: Seamless Creativity*, 2018, pp. 1–7.
- [18] M. Milligan, "Interactive hpc gateways with jupyter and jupyterhub," in *Practice and Experience in Advanced Research Computing 2017: Sustainability, Success and Impact*, ser. PEARC '17. New York, NY, USA: Association for Computing Machinery, 2017.
- [19] R. Thomas and S. Cholia, "Interactive supercomputing with jupyter," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 93–98, 2021.
- [20] I. Colonnelli, M. Aldinucci, B. Cantalupo, L. Padovani, S. Rabellino, C. Spampinato, R. Morelli, R. Di Carlo, N. Magini, and C. Cavazzoni, "Distributed workflows with jupyter," *Future Generation Computer Systems*, vol. 128, pp. 282–298, 2022.
- [21] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman, "Design and implementation for checkpointing of distributed resources using process-level virtualization," in *2016 IEEE International Conference on Cluster Computing (cluster)*. IEEE, 2016, pp. 402–412.
- [22] N. Weinman, S. M. Drucker, T. Barik, and R. DeLine, "Fork it: Supporting stateful alternatives in computational notebooks," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21. New York, NY, USA: Association for Computing Machinery, 2021.
- [23] Z. Li, S. Chockchowwat, H. Fang, R. Sahu, S. Thakurdesai, K. Pridaphatrakun, and Y. Park, "Demonstration of elasticnotebook: Migrating live computational notebook states," in *Companion of the 2024 International Conference on Management of Data*, 2024, pp. 540–543.
- [24] 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.