

Record and Replay in Interactive and Distributed Computing

Raza Ahmad
DePaul University
Chicago, Illinois, USA
raza.ahmad@depaul.edu

Abstract

Computational notebooks increasingly drive distributed workflows on HPC clusters. Yet, such workflows are hard to reproduce: the notebook kernel accumulates hidden state that the notebook file does not record, capturing a workflow that spans many workers and has no natural termination point is costly, and a shared notebook fails on a new cluster because its software and data dependencies do not accompany it. Record and replay offers a systematic way to reproduce these workflows, but existing techniques largely assume a single batch process on a local machine. This paper addresses this gap by extending record and replay to interactive and distributed environments. Our approach captures provenance at points the system already controls, namely the kernel, the workflow manager, and each worker, and identifies cells and tasks by lineage and stable fingerprints that remain valid across runs and clusters. We develop audit and repeat kernels that bind a notebook to the environment it ran in, and backpacks that capture the software, data, and resource needs of the manager and workers for portable execution. We show that incremental checkpoints and a stateful task cache let a modified notebook re-execute only the affected work, and that auditing groups of similar tasks only once reduces audit time by over 90% compared to naively tracing each task.

ACM Reference Format:

Raza Ahmad. 2026. Record and Replay in Interactive and Distributed Computing. In . ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Traditional computing is batch oriented: a script runs from top to bottom on one machine without user intervention and yields a result only when it finishes. Interactive computing departs from this model in three ways. Execution is out of order because cells can be run in any sequence, a human remains in the loop to steer a long-lived process, and feedback appears immediately next to the code that produced it. Computational notebooks such as Jupyter [22] and Apache Zeppelin [3] are its most popular forms. In a notebook, the user edits cells in a browser client, which sends their code to a kernel process that holds the live in-memory state, executes the code, and returns the results.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

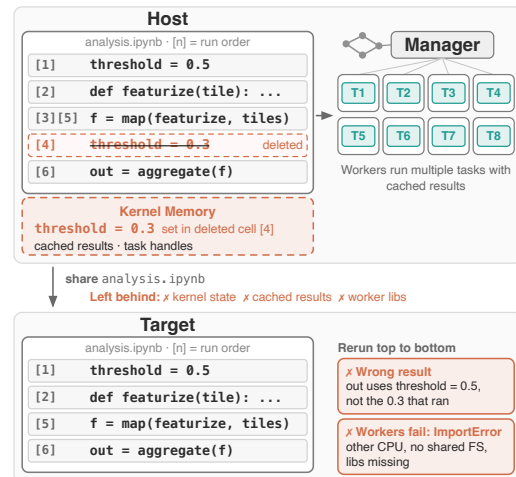


Figure 1: A notebook workflow on a host cluster. When shared with a target cluster, a rerun either fails or computes a wrong result.

Distributed computing departs from batch computing along a different axis: it changes where a program runs. Its design space ranges from tightly coupled programs such as MPI simulations, whose processes exchange fine-grained messages in lockstep, to loosely coupled workloads of a large number of tasks that communicate through files. We study the latter, specifically *many-task computing* [37], in which a manager holds a directed acyclic graph (DAG) of tasks and dispatches them to workers that share neither memory nor, often, a filesystem. Engines such as TaskVine [41], Dask [39], and Parsl [6] let scientists build these graphs inside notebook cells—a pattern common in image processing, genomics, and machine learning pipelines.

Increasingly, the two paradigms are converging into interactive HPC, because scientific workloads quickly outgrow a workstation [36]. Machine learning is a prominent driver of this convergence, from data preparation and evaluation steered from notebooks to the interactive prototyping and fine-tuning around LLM training [9, 18, 25, 31, 33]. On an interactive HPC system [13], a user obtains an allocation through Slurm [48] or a gateway such as Open OnDemand [19], and the notebook kernel runs on a compute node where it drives a distributed engine, so a single cell can dispatch a task graph across hundreds of workers. The workflow's state is therefore split: the kernel holds the DAG, task handles, and cached data, while each worker holds the state of its own tasks.

We study reproducibility in these environments through *record and replay* techniques, also called *audit and repeat*. *Record* audits the workflow as it runs and packages the notebook together with the software and data used by its kernel, manager, and workers, as well as the provenance that identifies each cell and task. *Replay* uses this package to reconstruct the workflow elsewhere so that

Table 1: Properties of record-and-replay systems. The middle group lists systems designed for notebooks.

System	Purpose	Records	Distributed?	Interactive?	Portable?
Staggered checkpointing [46]	fault tolerance	process state	yes	no	no
DMTCP [2]	fault tolerance	process state	yes	no	no
liblog, Friday [14, 15]	debugging	library calls, messages	yes	no	no
rr [32]	debugging	system calls, signals	no	no	no
CDE [16]	reproducibility	files, libraries	no	no	yes
ReproZip [8]	reproducibility	files, libraries	no	no	yes
Sciunit [44]	reproducibility	files, libraries	no	partial	yes
NBSafety [29]	stale-state detection	cell dependencies	no	yes	no
ElasticNotebook [28]	session migration	kernel state	no	yes	partial
CHEX [30]	multiversion replay	per-cell lineage, checkpoints	no	yes	no
Kishu [27]	session time travel	changed variables	no	yes	no
Our Systems [1, 4, 5, 20]	reproducibility	per-cell lineage, per-task provenance	yes	yes	yes

every recorded cell produces an equivalent output. Classical tools such as rr [32] and application virtualization [16] capture one batch execution that ends when its process exits, whereas a notebook never exits and its state spans a kernel and many workers.

Figure 1 shows a typical notebook on a cluster, with each cell labeled by its execution order. The user first sets `threshold = 0.5` in cell [1] and defines a function `featurize` in cell [2] that reads this threshold. In cell [3], the user maps `featurize` over thousands of image tiles; the distributed engine dispatches one task per tile to the workers, which keep the results. Unsatisfied with the output, the user adds a new cell (cell [4]), sets `threshold = 0.3`, and reruns the map cell, which now executes as [5] and recomputes every tile with the new value. The user then aggregates the results in cell [6], deletes the cell that set the threshold to 0.3, and saves the notebook. The saved file now shows only `threshold = 0.5`, while every reported result was computed with 0.3, a value that survives only in the kernel’s memory. When the file is shared without its environment, a rerun on another cluster either fails because the workers lack its dependencies, or silently computes a different answer using the 0.5 that the file records.

This example exposes three challenges for reproducibility. The first is the *hidden state trap*: out-of-order execution accumulates state in the kernel that cannot be reconstructed by merely reading the source from top to bottom. Distribution makes it worse, since part of that state, such as cached results, lives on workers that the file never describes. The second is the *reproducibility overhead*: capturing and re-executing these workflows is expensive. An interactive workflow has no fixed point at which its full state can be captured once [47], so capture must happen continually as cells run. In a distributed setup, tracing every task separately copies the same libraries thousands of times and incurs two privilege-mode transitions on every system call. Re-execution is costly as well: during iterative development, a one-line edit to the `featurize` function forces the whole task graph to rerun. The third challenge is *portability*: a notebook shared without its environment fails on workers that lack its libraries or data. Rebuilding that environment by hand on each cluster is impractical for the domain scientists who increasingly use interactive HPC, most of whom are unfamiliar

with batch scripts and containers [38], so the environment must be captured and restored automatically.

Research Goal: The objective of my Ph.D. research is to extend record and replay techniques to interactive and distributed environments so that such workflows are captured efficiently, including their hidden state, and reproduced with full fidelity on other clusters. We address the following research questions:

- RQ1.** How can record and replay be extended to capture the execution environments and dependencies of interactive and distributed workflows?
- RQ2.** How should a captured interactive workflow be packaged for portable, full-fidelity replay across heterogeneous execution environments?
- RQ3.** During iterative development in a notebook, how can re-execution be accelerated by leaving unaffected prior computation untouched?
- RQ4.** How can redundant auditing in a distributed cluster be eliminated, so that provenance capture and replay scale with workload size?

2 Background

Notebooks trace back to literate programming [23], which interleaves code with prose for human readers. Jupyter [22, 34] carried this idea into an executable medium whose protocol separates the browser front end from the kernel, letting the kernel run on a remote compute node. Supercomputers, built for batch throughput, gained interactive access through JupyterHub deployments [36] and web portals [12, 40], driven by urgent and AI workloads [38]. Condor [43] schedules large numbers of independent jobs, a model that many-task engines extend to dependent tasks and bring into notebooks.

Techniques that record an execution and replay it later have developed along three lines of work, which differ in purpose, in what they record, and in what replay assumes (Table 1). *Checkpoint and rollback* records state so that a distributed computation survives failures [7, 11, 24, 42]. Vaidya showed that the capture cost can be moved off the critical path and paid later [45, 46]. BLCR [17] and DMTCP [2] bring checkpointing to Linux clusters. *Deterministic replay* records only nondeterminism so that a bug can be reproduced at the level of shared objects [26], virtual machines [10], distributed

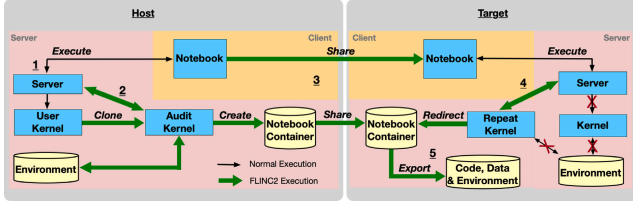


Figure 2: FLINC’s audit and repeat kernels. The audit kernel records the user kernel into a notebook container, which is shared with the notebook and used by the repeat kernel on the target (reproduced from [1]).

applications [14, 15], or single processes [32], and the minimal record needed for faithful replay has been characterized for causally consistent shared memory [21]. *Audit and repeat* targets reproducibility: it logs the files and libraries a program uses and redirects its accesses to the recorded copies during repeat, using a mechanism called application virtualization (AV), as in CDE [16], PTU [35], ReproZip [8], and Sciunit [44]. Building on Sciunit, FLINC [1] extends AV to interactive notebooks, while CHEX [30] audits per-cell lineage to replay multiple notebook versions efficiently by reusing ordered checkpoints. Other notebook tools flag stale state [29], checkpoint and restore only the variables a cell changed [27], or migrate live kernels [28], but they work on a single machine and produce no portable package.

3 Extending Record and Replay to Interactive and Distributed Environments (RQ1)

We model a notebook workflow as a sequence of cell executions $S = \langle c_1, \dots, c_n \rangle$ in an order chosen by the user. After c_i , its state is

$$\sigma_i = \kappa_i \cup \bigcup_w \omega_{w,i}, \quad (1)$$

the union of the kernel state κ_i and the states $\omega_{w,i}$ of the workers w . Audit, therefore, must produce a package that transcends the boundary of a single process and covers both κ_i and every $\omega_{w,i}$.

A notebook has no natural termination point, and its execution is split between a client, which holds the notebook file, and a server-side kernel, which performs all computation and is where hidden state accumulates. FLINC [1] therefore audits the kernel rather than the client, since the client already exists as a shareable notebook file. Its *audit* kernel wraps the user’s kernel, traces it, copies every accessed file into a notebook container, and flushes provenance when the server shuts the kernel down. On the target cluster, the *repeat* kernel reconnects the shared notebook to that container and redirects file accesses to it, so the notebook runs with the dependencies it was audited with rather than those installed on the target (Figure 2).

Since the notebook file travels outside the container, a container can silently be paired with an edited version of the notebook. FLINC binds the two through per-cell lineage, which also exposes hidden state. For cell c_i with code hash h_i and ordered system events E_i , including the content hashes of the external data they access, it records

$$g_i = (g_{i-1}, E_i, h_i), \quad g_0 = \emptyset, \quad (2)$$

and defines two program states ps_i and ps_j of notebook versions N_1 and N_2 to be equal if and only if $h_i = h_j$ and $g_i = g_j$. When a re-audited notebook’s lineage diverges from the stored one, FLINC

discards the stale association rather than repeating against a mismatched container; in Figure 1, the deleted cell that set the threshold to 0.3 appears in the audited lineage but not in the shared file, so FLINC can detect the mismatch instead of silently replaying with 0.5. On nine scientific notebooks, auditing the kernel costs only about 16% more than auditing the equivalent batch script, and repeat adds only 3.5% over plain execution.

FLINC captures the kernel, but in a distributed workflow most of the state lives on remote workers; in Figure 1, the results computed with 0.3 exist only there. A faithful record of each task is its system-call lineage $\lambda_k = (\lambda_{k-1}; h_k; \delta_k)$ with $\lambda_0 = \emptyset$, where h_k hashes the code running at step k and δ_k is the k -th system event with the content hash of any file it touches [5]. Tracing every task would capture it, but at the cost of auditing shared libraries once per task, and with records keyed by scheduler identifiers and worker paths that cannot be matched in another run. We therefore capture the distributed half where the system already has control. NBReplay [4] records at the kernel and the manager: it checkpoints the kernel state after each cell and logs every task the manager dispatches together with its outputs (Section 5). Fidelis [5] records on the workers: an in-process library logs the files each task accesses into a per-worker manifest, from which portable containers are built (Section 6). Both identify a task by a stable fingerprint. Each payload $\rho(t)$ is normalized into $\hat{\rho}(t)$ by replacing non-deterministic keys, file handles, and paths with canonical tokens, and references to upstream tasks with their fingerprints, giving

$$\phi(t) = \text{SHA-256}(\hat{\rho}(t)), \quad (3)$$

which two runs assign identically if and only if the task performs the same computation over the same upstream chain.

4 Packaging Workflows for Portability (RQ2)

A notebook file captures code but not its software, data, and resources, which is why the rerun in Figure 1 fails on workers that lack the notebook’s libraries. Floability [20] packages a workflow as a *backpack* $B = \langle W, S_m, S_w, D, R \rangle$ comprising the notebook W without site-specific settings; separate software specifications for the manager, S_m , and the workers, S_w , since workers usually need a small subset of the manager’s packages; a data specification D with each dataset’s source, location, and checksum; and a resource specification R of worker count, cores, memory, and disk.

Backpack generation can be automated using floability audit, which derives S_m , S_w , and D from a local run of the notebook with a single worker. The run is traced with trace and produces separate logs L_m and L_w for the manager and worker processes. For each role $x \in \{m, w\}$, every accessed path in L_x under a package directory is mapped to its package and version, and only packages that `conda list` or `pip list` reports for the active environment are kept, forming S_x . Data dependencies come from wrapping Python’s open function: files opened outside system and package directories form D , and each is attributed to the manager or a worker by matching against L_m and L_w . To run a backpack, Floability verifies each dataset against its checksum and launches the manager in the notebook kernel with a factory that scales workers through the site’s batch system, so the user never writes a job script or container recipe. On three applications, auditing costs 16–49% more

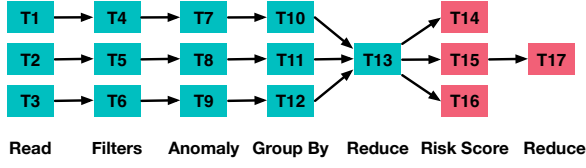


Figure 3: Partial re-execution of a task graph in NBReplay. After the user function is edited, only the red tasks rerun and the green tasks return cached results (reproduced from [4]).

than plain notebook execution, and the resulting backpacks ran unchanged on four HTCondor and Slurm clusters.

5 Efficient Re-execution during Iterative Development (RQ3)

After a small edit, a user expects only the affected computation to re-run. Because engines such as Dask and TaskVine are stateless across invocations, however, a one-line edit to featurize in Figure 1 re-runs every tile, and in Figure 3 an edit that affects only the final map and reduce tasks re-executes the whole graph. NBReplay [4] enables partial re-execution of the workflow through incremental kernel checkpoints and a stateful manager. For the kernel, persisting the whole namespace after every cell is costly, so NBReplay parses each cell’s abstract syntax tree to obtain the variables $A(c_k)$ it reads or writes. Because static slicing misses aliasing, it keeps a reverse index $\mathcal{R}$ from memory addresses to variable names and also checkpoints every variable that shares an object with one in $A(c_k)$. Each entry stores the serialized object when possible, the code that produced it, and its dependencies, and identical entries across cells are stored once. Since a checkpoint is taken after every executed cell, it can preserve the kernel state that actually produced each result, including values set by cells that were later deleted.

For distributed work, a ReplayManager wraps the TaskVine manager and interposes on task submission and completion. For a Python task it computes a two-stage fingerprint: a core hash over the canonicalized function source and arguments, with framework-generated identifiers stripped as in Equation 3, combined with a hash $H_{in}(t)$ of the sorted content hashes of its declared input files, so that $f(t) = H(H_{core}(t) || H_{in}(t))$. If $f(t)$ is in the transaction log, the cached outputs are returned without dispatch; otherwise the task runs and its outputs are logged. This assumes deterministic tasks with immutable, declared inputs. Because checkpoints and the task cache travel with the notebook, a collaborator on another cluster gets the same savings. On five notebooks, auditing adds 18% overhead on average. Repeating an unchanged notebook is 18.4× faster than rerunning it, and repeating a modified notebook is 2.08× faster, with cache hit rates of 50–92.3%.

6 Eliminating Redundancy in Distributed Auditing (RQ4)

Many tasks in a scientific workflow load the same libraries, and the strawman of Section 3 traces and copies them once per task (Figure 4, left), so the cost of capture grows with the size of the workflow rather than with the variety of its tasks. Fidelis [5] instead audits each group of similar tasks once. At submission, Fidelis scans a task’s bytecode for imported modules to obtain its module

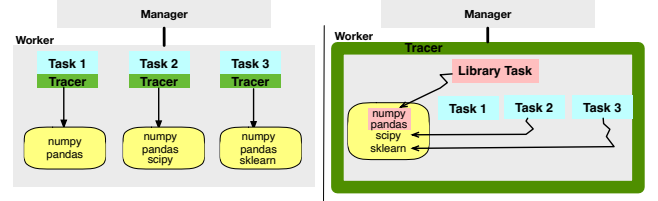


Figure 4: Per-task auditing (left) traces every task and copies shared libraries into each container. Fidelis (right) audits a shared library task once and records only task-specific accesses (reproduced from [5]).

signature $\sigma(t)$. Each worker keeps a pool C of library tasks, long-running processes L_σ that have pre-loaded the modules in σ . A task is served by the smallest library task that covers it, $\sigma_L(t) = \min\{\sigma \in C : \sigma(t) \subseteq \sigma\}$, or by a new one with $\sigma_L(t) = \sigma(t)$ if none does, and the tasks served by L_σ form the audit equivalence class $E_\sigma = \{t : \sigma_L(t) = \sigma\}$. Each task runs as a forked child of L_σ , inheriting the loaded libraries, while copy-on-write memory isolates tasks.

To avoid `ptrace`’s privilege-mode transitions on every system call, a library loaded with `LD_PRELOAD` shadows the `glibc` file, metadata, library-loading, and directory functions inside each process, so capture stays in user mode. It appends entries (τ, p, L_σ) of file type, path, and serving library task to a per-worker manifest $\mathcal{M}_w$, once per class; libraries a task loads beyond its signature are still recorded, which preserves completeness. Containers are materialized from $\mathcal{M}_w$ after execution or during idle periods, and each replayed task runs in the filtered view $\mathcal{M}_w[E_\sigma]$ of entries attributed to L_σ , located through an audit map $\mathcal{A}[\phi(t)]$ that records the task’s class, worker, and manifest. If interposition sees every access, tasks interact only through declared files, and those files do not change before being copied, a replayed t' with $\phi(t') = \phi(t)$ reproduces $\lambda_n(t)$ up to process identifiers and interleavings. Because the containers carry no worker-specific paths, they are designed to replay on workers that share no filesystem with the original run.

On seven workers, Fidelis audits four homogeneous workflows in 90–97% less time than per-task `ptrace` audit, and 84–93% less than even unaudited execution, since context reuse also removes start-up costs.

7 Conclusion

We extend record and replay from single batch processes to workflows that are interactive and may span a distributed cluster. FLINC captures a notebook kernel into a container and detects when the notebook no longer matches the execution it records. Floability audits a local run into a backpack of software, data, and resource needs that runs unchanged across clusters. NBReplay restores kernel state from incremental checkpoints and re-executes only the tasks an edit invalidates, and Fidelis audits groups of similar tasks once through execution context reuse, system-call interposition, and deferred container materialization. Together, these systems expose and preserve hidden state, keep the cost of capture and re-execution low, and make workflows portable without burdening their users.

References

- [1] Raza Ahmad, Naga Nithin Manne, and Tanu Malik. 2025. Improving reproducibility of interactive notebooks using application virtualization. *Future Generation Computer Systems* (2025), 108043.
- [2] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTC: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*.
- [3] Apache Foundation. 2026. Apache Zeppelin. <https://zeppelin.apache.org/>. [Online; accessed 2-September-2026].
- [4] Talha Azaz, Raza Ahmad, Md Saiful Islam, Douglas Thain, and Tanu Malik. 2026. Efficiently Reproducing Distributed Workflows in Notebook-based Systems. In *2026 IEEE 26th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 74–83.
- [5] Talha Azaz, Raza Ahmad, Douglas Thain, and Tanu Malik. 2026. Fidelis: Exploiting Task Similarity for Low-overhead Distributed Workflow Audit and Replay. In *27th ACM International Middleware Conference, 2026*. IEEE.
- [6] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM.
- [7] K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems* 3, 1 (1985), 63–75.
- [8] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational reproducibility with ease. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*.
- [9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of machine learning research* 24, 240 (2023), 1–113.
- [10] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*.
- [11] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *Comput. Surveys* 34, 3 (2002). doi:10.1145/568522.568525
- [12] Brian D Etz, David M Rogers, Michael J Brim, Ketan Maheshwari, Kellen Leland, Tyler J Skluzacek, Jack Lange, Daniel Pelfrey, Jordan Webb, Patrick Widener, et al. 2025. Enabling Seamless Transitions from Experimental to Production HPC for Interactive Workflows. In *International Conference on High Performance Computing*. Springer, 363–375.
- [13] Rafael Ferreira da Silva, Rosa M. Badia, Deborah Bard, Ian T. Foster, Shantenu Jha, and Frédéric Suter. 2024. Frontiers in Scientific Workflows: Pervasive Integration With High-Performance Computing. *Computer* 57, 8 (2024), 36–44.
- [14] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2007. Friday: Global comprehension for distributed replay. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*.
- [15] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay debugging for distributed applications. In *USENIX Annual Technical Conference*.
- [16] Philip J. Guo and Dawson Engler. 2011. CDE: Using system call interception to automatically create portable software packages. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*.
- [17] Paul H. Hargrove and Jason C. Duell. 2006. Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series* 46 (2006).
- [18] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* (2022).
- [19] Dave Hudak, Doug Johnson, Alan Chalker, Jeremy Nicklas, Eric Franz, Trey Dockendorf, and Brian L. McMichael. 2018. Open OnDemand: A web-based client portal for HPC centers. *Journal of Open Source Software* 3, 25 (2018), 622. doi:10.21105/joss.00622
- [20] Md Saiful Islam, Talha Azaz, Raza Ahmad, A S M Shahadat Hossain, Furqan Baig, Shaowen Wang, Kevin Lannon, Tanu Malik, and Douglas Thain. 2025. Backpacks for Notebooks: Enabling Containerized Notebook Workflows in Distributed Environments. In *Proceedings of the 21st IEEE International Conference on e-Science (eScience '25)*. 169–177.
- [21] Russell L. Jones, Muhammad S. Khan, and Nitin H. Vaidya. 2018. Brief Announcement: Optimal Record and Replay under Causal Consistency. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*.
- [22] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks—a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*. IOS press, 87–90.
- [23] Donald Ervin Knuth. 1984. Literate programming. *The computer journal* 27, 2 (1984), 97–111.
- [24] Richard Koo and Sam Toueg. 1987. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering* SE-13, 1 (1987), 23–31.
- [25] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. 2023. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access* 11 (2023). Preprint: arXiv:2205.02302.
- [26] Thomas J. LeBlanc and John M. Mellor-Crummey. 1987. Debugging parallel programs with Instant Replay. *IEEE Trans. Comput.* C-36, 4 (1987), 471–482.
- [27] Zhaoheng Li, Supawit Chockchowwat, Ribhav Sahu, Areet Sheth, and Yongjoo Park. 2024. Kishu: Time-Traveling for Computational Notebooks. *Proceedings of the VLDB Endowment* 18, 4 (2024), 970–985. doi:10.14778/3717755.3717759
- [28] Zhaoheng Li, Pranav Gor, Rahul Prabhu, Hui Yu, Yuzhou Mao, and Yongjoo Park. 2023. ElasticNotebook: Enabling Live Migration for Computational Notebooks. *Proceedings of the VLDB Endowment* 17, 2 (2023), 119–133.
- [29] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-Grained Lineage for Safer Notebook Interactions. *Proceedings of the VLDB Endowment* 14 (2021).
- [30] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. 2022. CHEX: Multiversion Replay with Ordered Checkpoints. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1297–1310.
- [31] Jay McGlothlin and Christopher Carothers. 2025. Implementing support for Interactive and AI workloads in a traditional HPC environment. In *Proceedings of the SC'25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2146–2150.
- [32] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 377–389.
- [33] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.
- [34] Fernando Pérez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007), 21–29.
- [35] Quan Pham, Tanu Malik, and Ian Foster. 2013. Using provenance for repeatability. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*.
- [36] Min Ragan-Kelley, Matt Henderson, Fernando Pérez, Rollin Thomas, Shreyas Cholia, and Lavanya Ramakrishnan. 2024. Enabling Scientific Collaboration with JupyterHub. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1993–2002.
- [37] Ioan Raicu, Ian T. Foster, and Yong Zhao. 2008. Many-task computing for grids and supercomputers. In *2008 Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*. IEEE, 1–11. doi:10.1109/MTAGS.2008.4777912
- [38] Albert Reuther, Nick Brown, William Arndt, Johannes Blaschke, Christian Boehme, Antony Chazapis, Bjoern Enders, Robert Henschel, Julian Kunkel, and Maxime Martinasso. 2024. Interactive and urgent HPC: challenges and opportunities. *arXiv preprint arXiv:2401.14550* (2024).
- [39] Matthew Rocklin et al. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *SciPy*. 126–132.
- [40] Tyler J Skluzacek, Paul Bryant, Arthur J Ruckman, Daniel Rosendo, Suzanne Prentice, Michael J Brim, Ryan Adamson, Sarp Oral, Mallikarjun Shankar, and Rafael Ferreira da Silva. 2025. Secure api-driven research automation to accelerate scientific discovery. In *Practice and Experience in Advanced Research Computing 2025: The Power of Collaboration*. 1–5.
- [41] Barry Sly-Delgado, Thanh Son Phung, Colin Thomas, David Simonetti, Andrew Hennessee, Ben Tovar, and Douglas Thain. 2023. Taskvine: Managing in-cluster storage for high-throughput data intensive workflows. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 1978–1988.
- [42] Rob Strom and Shaula Yemini. 1985. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems* 3, 3 (1985), 204–226.
- [43] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience* 17, 2–4 (2005), 323–356.
- [44] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. 2017. Sciunits: Reusable research objects. In *2017 IEEE 13th International Conference on e-Science (e-Science)*. IEEE, 374–383.
- [45] Nitin H. Vaidya. 1997. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.* 46, 8 (1997), 942–947. doi:10.1109/12.609281
- [46] Nitin H. Vaidya. 1999. Staggered consistent checkpointing. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (1999).
- [47] Elias Werner, Anton Rygin, Andreas Gocht-Zech, Sebastian Döbel, and Matthias Lieber. 2024. JUMPER: Performance Data Monitoring, Instrumentation and Visualization for Jupyter Notebooks. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2003–2011.

- [48] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing (JSSPP 2003) (Lecture Notes in Computer Science, Vol. 2862)*. Springer, 44–60.

doi:10.1007/10968987_3