SciWIND: Effectively Exploiting Node-Local Storage for Data-Intensive High-Energy Physics Workflows

Jin Zhou, Colin Thomas, Barry Sly-Delgado, Connor Moore, Benjamin Tovar, Kevin Lannon, Douglas Thain
University of Notre Dame, South Bend, IN, USA
{jzhou24, cthoma26, bslydelg, cmoore24, btovar, klannon, dthain}@nd.edu

***Abstract*—Large-scale data analysis workflows can benefit substantially from leveraging node-local storage for intermediate data, but doing so reliably at scale is challenging. Unmanaged NLS usage can exhaust local disks and trigger worker crashes; even without disk exhaustion, random worker losses can permanently delete intermediate data, causing recovery cascades that inflate recomputation, prolong tail latency, and amplify storage pressure. We present SciWIND, a workflow-integrated framework built atop TaskVine that coordinates three mechanisms to address these coupled issues: *NLS Minimization* to bound per-node and global NLS usage, *Efficient Recovery* to maintain scheduler throughput under frequent data loss, and *Resilience Reinforcement* to add bounded redundancy that reduces recomputation without overwhelming storage or the parallel file system. We evaluate SciWIND on three scientifically meaningful high-energy physics workflows in an HTCondor environment with injected worker evictions. Across workloads, SciWIND reduces peak NLS usage by up to $99.0\,\%$, makespan by up to $70.1\,\%$, and recovery task volume by up to $99.8\,\%$. In an at-scale DV5 hero run processing $\sim 5\times$ more input data under the same failure model, SciWIND cuts recovery tasks by $91.95\,\%$, reduces makespan by $71.13\,\%$, and lowers total NLS consumption across workers by $94.75\,\%$, demonstrating that lightweight hybrid policies can simultaneously suppress storage growth and recomputation at larger scale.**

***Index Terms*—Scientific workflows, high-performance computing, workflow management systems, node-local storage, failure recovery, storage management mechanisms**

I. Introduction

A scientific workflow is a structured sequence of computational tasks, where each task is well-defined with specific input requirements and expected outputs. These tasks form a directed acyclic graph (DAG) with the task execution order determined by data dependencies between tasks. Workflows are widely used across scientific domains to break down complex computations into manageable tasks, executed by workflow management systems (WMS) in high-performance computing (HPC) systems, which provide massive parallel cores to accelerate the computation. Representative workflows include Montage for astronomical imaging [1], SciEvol for molecular evolution [2], RNA-seq pipelines [3], LIGO analyses [4], and RSTriPhoton [5] for particle collisions [5]. As the experimental data generation has been growing rapidly and scientific analysis becoming more complex [6]–[8], these workflows often struggle to scale up due to the data movement challenges [9]. One representative example is found in the high-energy physics (HEP) community, where the rate of data generation far exceeds available computing capacity [10].

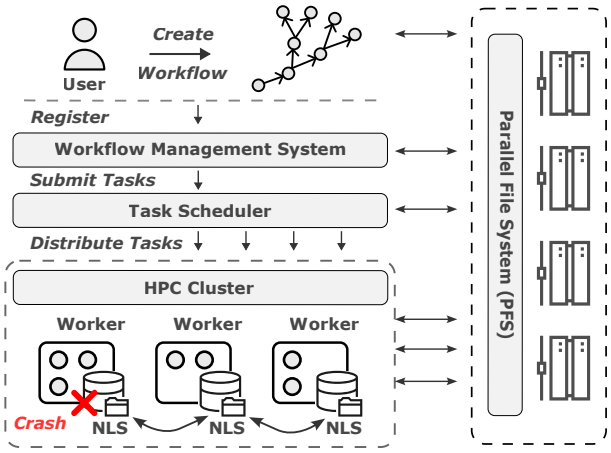

Fig. 1: Architecture of running workflows through a WMS in an HPC environment. PFS offers reliable storage accessible across all layers but has high I/O contention, while NLS provides isolated environments with larger aggregated bandwidth and higher I/O throughput but can crash and cause data loss.

HPC systems often combine a multi-layered storage architecture to reconcile performance, reliability, and resource utilization. Traditionally, data storage is provided via a parallel file system (PFS), which offers substantial capacity, reliable data persistence, and favorable performance for common workloads. With the growing computational demands of workflows, the high I/O contention and network file access of PFS can become a significant bottleneck. As an alternative, node-local storage (NLS) has been widely explored to shift the storage burden from the PFS to the compute nodes. Figure 1 shows the architecture of running workflows through a WMS in an HPC environment where PFS and NLS are integrated.

However, while promising, adding additional storage layers requires dedicated system management techniques, otherwise imprudent handling can lead to resource under-utilization or even workflow failures. We identify three key challenges: **First**, NLS is limited in capacity, careless usage can lead to storage exhaustion and node crashes; **Second**, even if NLS is not exhausted, runtime nodes can crash due to unpredictable factors

and result in permanent data loss, these lost data must be recovered reliably; **Third**, data redundancy helps to mitigate recovery cost, but excessive redundancy can saturate the storage space and slow down the system, a systematic approach is needed to balance resilience and performance.

These challenges stem from node failures, and addressing them requires a unified design for prevention, recovery, and tolerance. Prior studies have proposed promising optimizations for individual aspects [11]–[13], yet few have explored a systematic approach for managing NLS effectively, especially for large-scale workflows in failure-prone environments. To fill this gap, we present Scientific Workflows with Integrated NLS for Dependability (SciWIND), a comprehensive framework that systematically addresses these challenges and strikes a practical balance among resource utilization, system efficiency, and failure resilience. Guided by these challenges, SciWIND is structured around three key components:

- **NLS Minimization:** Reduce per-node and overall peak NLS consumption to prevent disk exhaustion and resulting worker crashes, through Disk Load Shifting, Aggressive Pruning, and Largest-Input-First Scheduling.
- **Efficient Recovery:** Manage recovery tasks efficiently when unpredictable node failures cause data loss, using Multi-Queue Dispatcher and Proactive and Prioritized Recovery.
- **Resilience Reinforcement:** Mitigate recomputation cost while sustaining performance, through Depth-Aware Pruning, Peer Replication, and PFS Checkpointing.

We implement SciWIND within the TaskVine [14], a WMS that has granular DAG-structured workflow management with strong support for NLS. Evaluations are conducted through three HEP workflows, TopEFT [15], RSTriPhoton [5], and DV5 [15], deployed in an HTCondor cluster [16], an opportunistic high-throughput computing environment widely used at research institutions. To simulate node failures, we force a fixed worker eviction rate so that workflow executions are comparable across experiments. Our evaluation on three representative HEP workflows shows that SciWIND significantly reduces storage consumption, makespan, and recovery overhead, confirming its effectiveness in large workflows running in failure-prone environments.

The contributions of this work are summarized as follows:

- We propose SciWIND, a comprehensive framework that systematically manages NLS for data-intensive scientific workflows.
- We show how coordinated strategies reduce NLS consumption while addressing reliability tradeoffs, and how hybrid solutions balance performance and overhead, offering practical guidance for system configuration.
- We implement SciWIND in TaskVine and evaluate it at scale on real-world scientific workflows, demonstrating significant improvements and providing valuable insights for workflow developers and system administrators.

II. Background

A. HEP Workflows

HEP experiments are among the world's largest producers of scientific data, primarily through facilities like the Large Hadron Collider (LHC) at CERN in Switzerland. The annual data output of the LHC experiments has grown from about 30 PB in 2013 to roughly 600 PB in 2025, with projections on the order of 100 EB by 2040 [17]–[19]. These massive datasets are distributed worldwide, enabling physicists across the globe to construct complex analysis workflows tailored to their research. A typical HEP workflow processes large volumes of raw collision event data by applying selection cuts, performing computations (e.g., physics calculations), and deriving higher-level results. Individual dataset sizes can range from tens of terabytes to multiple petabytes, placing extreme demands on both computing and storage infrastructures. To cope with this scale, HEP researchers rely on distributed high-performance computing resources and advanced WMS to orchestrate their computations. The WMS must evolve alongside storage technologies to handle ever-growing data volumes in the exascale era. Indeed, our work is motivated by experience scaling HEP workflows to such magnitudes. While we focus on HEP use cases, the challenges and solutions discussed here are broadly applicable to other data-intensive scientific domains.

B. Storage Substrates

In modern HPC clusters and supercomputers, tasks traditionally read and write data through a center-wide PFS such as Lustre [20], BeeGFS [21], CephFS [22], or the VAST Data system [23]. These PFS installations provide a shared, persistent storage layer accessible from all compute nodes, ensuring that once output data are written, they remain reliably stored until explicitly deleted by the user. While robust, a PFS can become a scalability bottleneck for two main reasons. First, a high volume of concurrent reads and writes from many tasks can saturate the PFS's aggregate bandwidth, leading to I/O contention and slowing down application performance [24]–[27]. Second, scientific workflows often generate enormous intermediate data products; if all such intermediate files are kept on the PFS, they risk quickly exhausting the finite storage capacity [28]. Conversely, attempting to delete intermediates on the fly is non-trivial, and real-time garbage collection would involve a flurry of serialized `unlink` operations on the PFS servers, incurring significant overhead and interfering with I/O throughput.

To alleviate these issues, modern HPC systems increasingly incorporate fast node-local tiers alongside the PFS. For example, many supercomputers provision each compute node with a local NVMe SSD or provide a rack-level "burst buffer" layer to absorb bursty I/O, while relying on the PFS for long-term persistence. Some systems further virtualize these node-local disks into a unified namespace via a distributed file system (DFS), whereas others expose them directly as per-node storage managed by the runtime. Oak Ridge's Summit system [29], for

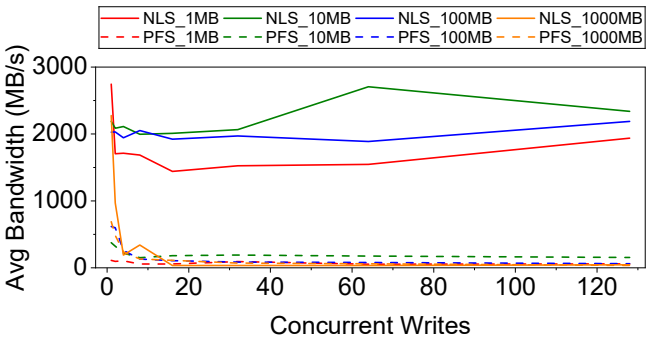

Fig. 2: Average write bandwidth to NLS and PFS (detailed in Section V) under varying levels of concurrency. Measured on 8 nodes with 16 cores each. NLS achieves consistently higher write bandwidth than PFS for mid-sized files, but not for very large files. (Similar results hold for reads.)

instance, employs an in-system layer of node-local SSDs that delivers substantially higher aggregate I/O rates than the center-wide file system; intermediate workflow data are written to this layer and only periodically flushed to the PFS based on user directives, which reduces network load and improves overall I/O efficiency. This kind of multi-tier storage architecture leverages the low latency and high throughput of local disks for short-lived data, reserving the PFS for final outputs and durability.

However, using heterogeneous storage hierarchies introduces new challenges in system management. A key issue is load imbalance and variability: nodes with faster processors or accelerators (e.g., GPUs) tend to complete more tasks and consequently accumulate more intermediate data on their local disks, potentially filling up their NLS while other nodes still have plenty of space. Such imbalance can lead to certain nodes running out of local storage (causing task failures) long before the job as a whole is finished. More broadly, ensuring that NLS is effectively utilized without sacrificing reliability has become an active research area in HPC and distributed systems. Our work contributes to this area by developing methods to make NLS usage in workflows more accessible, balanced, and fault-tolerant. In essence, providing the benefits of fast local storage to the workflow while transparently mitigating its pitfalls.

III. TaskVine Architecture

SciWIND is implemented within TaskVine [14], a workflow management system that aggressively exploits NLS. The general architecture of TaskVine is shown in Figure 1, consisting of a manager process and multiple worker processes that run within a cloud or HPC cluster. The end user writes a high-level application that chains together tasks and files (or functions and objects) into a large DAG: every task consumes one or more files, and produces one or more files. As their dependencies are satisfied, tasks are released to the task scheduler, which selects a worker for each task according to a scheduling policy, and transmits the task and input files to that worker, if they are not already there.

Each worker is responsible for managing the tasks and files assigned to it, both kept in an NLS cache area. The worker runs each task by creating a private sandbox, linking in the input files, executing the task, and moving the output files from the sandbox back to the NLS cache. All files sent by the manager or produced as task outputs accumulate in the NLS cache for use by future tasks. When necessary, the manager may direct a worker to replicate a file to another worker, or send it to the PFS for durable storage. If a worker node should crash or fail, the manager can recover by exploiting replicated files, or reconstruct missing files by re-executing tasks from the workflow DAG.

This overall architecture maximizes the use of NLS and minimizes the impact on PFS. Files that are produced by one task and then consumed by a later task can remain on a single worker, thus minimizing data movement. Widely shared data is effectively distributed through the cluster, fanning out in parallel through worker-to-worker transfers. Tasks are able to exploit the low-latency performance of NLS for fine-grained data access or metadata-intensive software access. This has substantial performance impact on data-analysis applications that make heavy use of shared read-only state and intermediate read-write state. [5], [30]

Figure 2 shows that NLS has the potential to achieve much higher write bandwidth than PFS for small and medium files, while PFS performs better for large files [31], [32], by exploiting local buffer/cache and avoiding remote synchronization. Since workflows often involve large numbers of small intermediate files, NLS has a potential performance advantage for such workloads.

In the best case, a small workflow may be able to fit entirely within the total NLS available across all worker nodes in the system. But a large workflow (as we show below) might easily fill up the limited NLS available on one (or all) workers, halting the workflow unless we take further steps. How should a workflow management system go about managing this limited storage space? SciWIND is our answer to this question.

IV. SciWIND Design and Implementation

SciWIND is a workflow data management layer built inside TaskVine that coordinates (1) how intermediate files are placed, replicated, and removed across NLS, (2) how the scheduler reacts efficiently when nodes fail and data become unavailable, and (3) how to add just-enough redundancy to reduce recomputation without overwhelming disks or the PFS.

Our design targets a three-way trade-off among *performance* (throughput and makespan), *storage* (peak NLS usage), and *resilience* (recovery task volume and tail latency). To navigate this triangle, SciWIND separates concerns into three coordinated components:

- **NLS Minimization** reduces per-node and global peaks to prevent disk exhaustion via Disk Load Shifting, Aggressive Pruning, and Largest-Input-First scheduling.
- **Efficient Recovery** keeps dispatch fast under failures by decoupling eligibility from readiness (Multi-Queue Dis-

patcher) and by launching the right work early (Proactive, Prioritized Recovery).
- **Resilience Reinforcement** bounds recomputation cost with Depth-Aware Pruning, selective Peer Replication, and topology-guided PFS Checkpointing.

In *Resilience Reinforcement*, we expose three adjustable knobs: pruning depth k (PD), replication count ω (RC), and checkpoint ratio α (CP), so users can tune resilience vs. performance/space to match observed failure tendencies and DAG structure. Other components use fixed policies with internal throttling/hysteresis and require no user tuning.

A. NLS Minimization

In this section, we describe how SciWIND prevents node failures from disk usage imbalance and overuse, where we aim to use as little NLS as possible while keeping throughput high.

*1) **Disk Load Shifting**:* To mitigate skew in worker cache utilization, SciWIND performs background disk load shifting: when the most loaded worker's cache occupancy substantially exceeds that of the least loaded one, and both peers are eligible for P2P transfers, it replicates temporary files from the former to the latter under per-end concurrency caps. The offload budget is set to roughly half the disparity, to ensure stable, monotone convergence without oscillation, with a hysteresis trigger so rebalancing only fires when a new high-water mark is crossed. File replicas are drawn from the source worker's cache, skipping those in active use and any already resident at the destination. Rebalancing only creates replicas, while deletion is deferred to the redundant-replica cleaner after the destination reports readiness, so availability is never reduced and dependent tasks are not forsaken. The procedure runs in the background every few seconds, not on the critical dispatch path. Each round is a quick linear scan and moves only a few files, so the cost is low and bandwidth use stays modest.

*2) **Aggressive Pruning**:* In stable deployments, such as reserved cloud nodes or dedicated batch partitions provided by Slurm or UGE, we use an aggressive policy: an intermediate is kept on NLS only while some downstream task that declares it as an input is pending or running; as soon as the last such consumer finishes, the intermediate becomes immediately eligible for eviction. In short, prune time $=$ time of last consumer completion. This yields the smallest steady-state disk footprint, lowers peak usage, and reduces the need for background offloading or peer replication (we typically keep at most one live replica). The trade-off is recovery cost: if the single remaining copy resides on a node that fails shortly after pruning, the system must recreate the intermediate by re-executing its producing subgraph, which can be expensive for deep pipelines. This mode assumes failures are rare and node tenure is long; recomputation is the accepted fallback when failures do occur.

*3) **Largest-Input-First Scheduling**:* With aggressive pruning, an intermediate is deleted once its last consumer completes. Thus, the consumer order directly controls how long large files remain on NLS. We model the retention time of a file F as $R(F) = t_{\mathrm{prune}}(F) - t_{\mathrm{create}}(F)$ and its time-weighted storage footprint as $\Phi(F) = S(F)\,R(F)$. Intuitively, consuming large intermediates earlier shortens their retention time and reduces peak disk pressure.

SciWIND prioritizes ready tasks by their cumulative input size:

$$P(T) = \sum_{F \in \mathrm{In}(T)} S(F),$$

so tasks that consume larger intermediates are scheduled earlier. To avoid starvation, we apply aging by increasing a task's priority with its waiting time:

$$P(T,t) = P(T) + \lambda\, W(T,t),$$

where $W(T,t)$ is the time that task T has waited in the runnable queue at time t, and λ controls how quickly waiting time is converted into priority. Runnable tasks are managed in a max-priority queue.

B. Efficient Recovery

In this section, we describe the inefficiencies that arise in task scheduling when node failures trigger recovery, and how SciWIND enables efficient data restoration under such failures. While recomputation is unavoidable, our goal is to minimize the additional overhead of managing recovery tasks.

*1) **Multi-Queue Dispatcher**:* Frequent failures expose a bottleneck in the single-queue design of WMS schedulers: many tasks in the ready queue are not runnable due to missing inputs, which leads to wasted scans and dispatch slowdown. To address this, we introduce a Multi-Queue Dispatcher. DAG-ready tasks first enter a pending queue, where lightweight checks determine their eligibility; only eligible tasks are promoted into a priority-based ready queue. The ready queue is always accessed from its top element; if the task is found ineligible, it is demoted back to the pending queue. Both queues are scanned in small prefixes per round, ensuring the scheduler spends bounded time on dispatch and can perform other control tasks.

*2) **Proactive Recovery**:* A node failure may delete intermediate data and render future tasks ineligible. In conventional recovery, this is remedied when the consumer is scheduled, at which point missing inputs must be regenerated by rerunning their producers. If those producers also lack inputs, recovery expands recursively and more tasks are submitted. This delay prolongs recovery and inflates tail latency, particularly when final merging tasks depend on all branches. SciWIND mitigates this cost with Proactive Recovery: upon a crash, it immediately identifies all lost files, scans undispatched tasks, and submits the necessary producers. Each file records its producer and DAG position, enabling recovery to start early, overlap with normal execution, and reduce tail latency.

*3) **Prioritized Recovery**:* Submitting recovery tasks early is necessary for performance but not sufficient: once in the scheduling queues, some recovery tasks are runnable while others are not, and their presence can also block existing ready tasks. Because overall progress depends on completing upstream work first, SciWIND raises the priority of recovery tasks above all regular tasks. Within the same subgraph, later-submitted tasks are treated as more upstream and assigned even

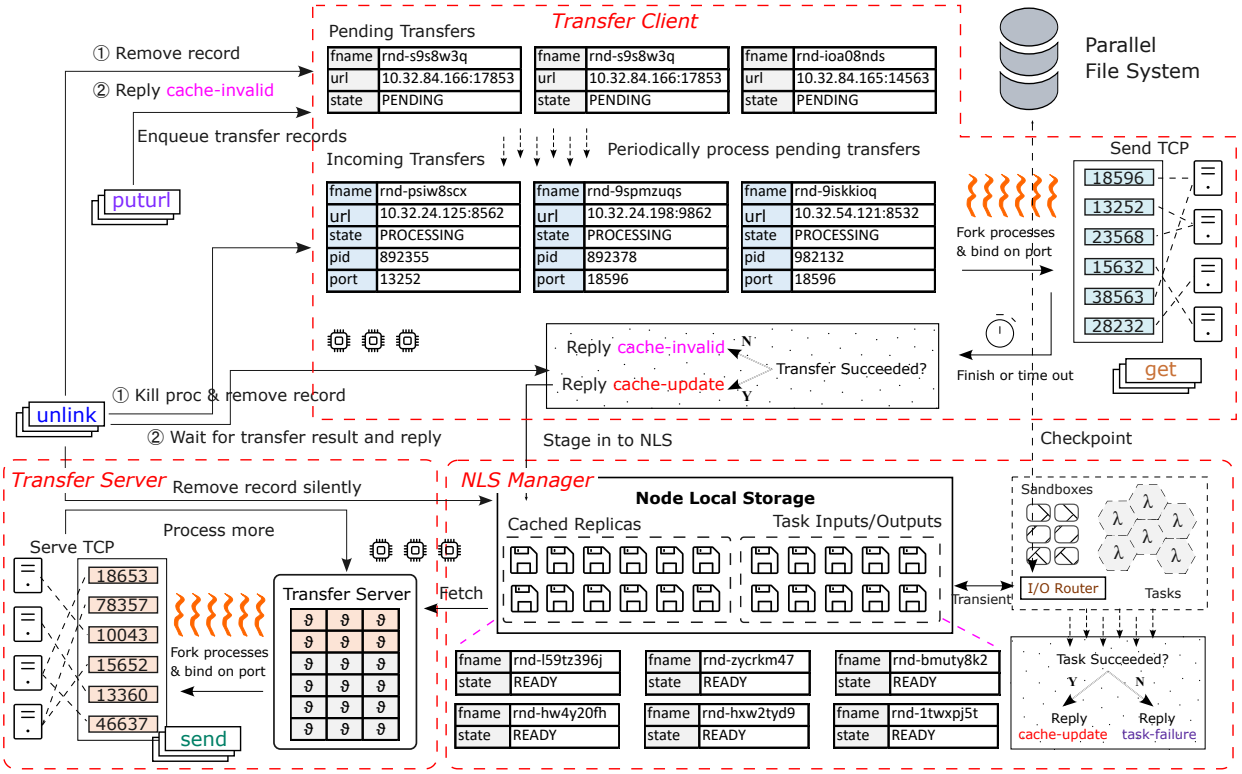

Fig. 3: Peer transfer architecture in SciWIND. The Transfer Client handles `puturl` *requests from the manager and periodically launches processes to fetch files from peer workers, while the Transfer Server accepts TCP connections from peers and forwards the requested files to them. NLS Manager is responsible for managing ready files appropriately on the worker.*

higher priority. This ensures that the most critical recovery work executes without delay, improving scheduling efficiency under failures.

C. Resilience Reinforcement

In this section, we describe how SciWIND leverages data redundancy to reduce the need for recomputation when inevitable node failures occur.

1) ***Depth-Aware Pruning****:* Although higher NLS usage risks node crashes, retaining some ancestors provides redundancy, shrinks recovery sub-DAGs, and lowers recomputation cost. Aggressive Pruning minimizes NLS usage but also amplifies recovery cost. To strike a balance between the two, SciWIND applies Depth-Aware Pruning, where a file is retained for k generations of its consumers and pruned only after those generations complete. In the implementation, pruning is triggered at the moment a task finishes, when its outputs are checked for eligibility to be discarded. Each file carries a pruning depth parameter that specifies how many generations of consumer tasks must finish before the file can be pruned. Depth-1 pruning is equivalent to aggressive pruning, while depth-k pruning ($k \geq 2$) retains the file until every output of its consumer tasks has itself satisfied the depth-$(k-1)$ criterion. Formally, the pruning time is defined as

$$t^{(k)}_{\text{prune}}(F) = \begin{cases} t_{\text{consumed}}(F), & k = 1, \\ \max\limits_{T \in \text{Cons}(F)} \; \max\limits_{F' \in \text{Out}(T)} t^{(k-1)}_{\text{prune}}(F'), & k \geq 2. \end{cases}$$

where $\text{Cons}(F)$ denotes the consumer tasks of file F and $\text{Out}(T)$ the outputs of task T. This recursive evaluation avoids expensive global scans by localizing pruning decisions to task completion events, with overhead proportional only to the chosen depth. As k increases, more ancestor files are preserved, providing redundancy and shrinking recovery sub-DAGs, whereas smaller k values reduce storage usage at the expense of higher recovery cost.

2) ***Peer Replication****:* To tolerate failures without excessive recomputation, SciWIND maintains multiple replicas of each intermediate. A target replication factor ω is specified, and whenever a file is created, it is inserted into a replication queue until the desired redundancy is reached. Transfers are scheduled between workers that already hold the file and those that do not, ensuring continued availability if a node fails. Figure 3 shows the peer transfer architecture in SciWIND.

To keep replication lightweight, two strategies are applied. First, a *multi-level throttle* limits the number of concurrent transfers: the manager issues at most ζ_{M} per round, while each worker caps concurrent transfers at $\zeta_{\text{W}_\text{r}}$, with a stricter bound $\zeta_{\text{W}_\text{t}}$ for task inputs to avoid overload. Second, a *balanced*

redundancy policy prioritizes files with fewer replicas. The replication queue is implemented as a binary heap keyed by current replica count, so that under-replicated files are always handled first. This ensures a fair distribution of redundancy across files, reducing the risk of losing vulnerable intermediates while bounding the cost of replication.

*3) **PFS Checkpointing**:* SciWIND provides another way to tolerate failures through checkpointing selected intermediates to the PFS. This strategy combines the persistence of PFS with the scalability of NLS: PFS guarantees that checkpointed files are never lost, while NLS provides high-throughput execution for large workflows. To mitigate I/O contention, only a subset of intermediates are marked as checkpointable before graph execution. When a task completes on a worker, its checkpointable outputs are persisted to PFS. The manager decides which intermediates to checkpoint, while the actual write is performed by workers. This works under the assumption that both can access the PFS.

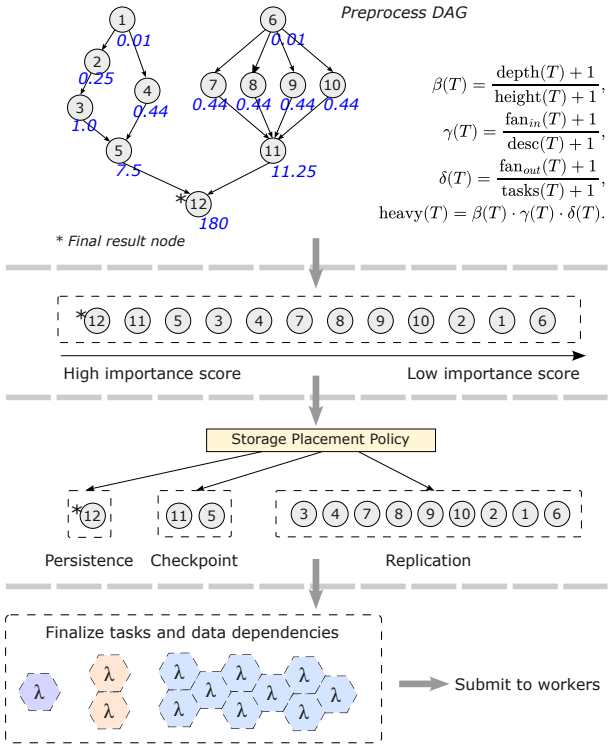

Fig. 4: DAG pre-processing in checkpointing. Nodes are sorted by their computed heavy scores and the top α fraction are marked for checkpointing and written to the PFS.

Figure 4 shows SciWIND's static DAG preprocessing, which determines which intermediates should be checkpointed. Before execution, SciWIND performs a topological pass and computes per-node structural metrics, including depth, height, upstream/downstream sizes, and fan-in/out. Each node T receives a heavy score:

$$\begin{cases} \beta(T) = \dfrac{\text{depth}(T)+1}{\text{height}(T)+1}, \\ \gamma(T) = \dfrac{|\text{anc}(T)|+1}{|\text{desc}(T)|+1}, \\ \delta(T) = \dfrac{\text{fan\_in}(T)+1}{\text{fan\_out}(T)+1}, \\ \text{heavy}(T) = \beta(T)\cdot\gamma(T)\cdot\delta(T). \end{cases}$$

The heavy score is intended to approximate the expected recomputation impact of losing a node after failures. The recovery cost of a task in a DAG depends on three structural aspects: (i) how much upstream work must be redone, (ii) how much downstream execution is blocked, and (iii) whether the node aggregates multiple inputs.

The terms $\beta(T)$, $\gamma(T)$, and $\delta(T)$ approximate these effects using only structural properties of the DAG. $\beta(T)$ captures position in the workflow by favoring nodes with large depth and small remaining height, $\gamma(T)$ compares accumulated upstream work to remaining downstream work, and $\delta(T)$ reflects aggregation by comparing fan-in to fan-out. Their product therefore estimates how disruptive the loss of T would be to overall progress.

Here, $\text{depth}(T)$ denotes the longest path from any source to T, $\text{height}(T)$ the longest path from T to any sink, $\text{anc}(T)$ and $\text{desc}(T)$ the ancestor and descendant sets, and $\text{fan\_in}(T)$ and $\text{fan\_out}(T)$ the local connectivity.

Rather than solving a global optimization problem for checkpoint placement, SciWIND ranks nodes by this estimated recovery impact and selects the top-α fraction for checkpointing. This favors nodes whose loss would trigger large recomputation and delay completion, reducing recovery from recompute-dominated to I/O-dominated cost ($\text{recompute}(F) \gg \text{io}(F)$).

V. Evaluation

A. Environment Setup

All mechanisms described in this paper are implemented in the public TaskVine codebase. The version used in this study corresponds to commit `227b25fc7`. The experimental environment is a campus-held HTCondor cluster, which provides access to over 300 machines and 20,000 cores. We run TaskVine manager on the front node, then use another factory process to submit workers and manage runtime connections. We use a campus-scale VAST [23] PFS with NVMe fabric and QLC flash; usable capacity is $\sim 10^2$–10^3 TB; typical read/write throughput is on the order of 1–10 GB s^{-1}; read and write IOPS are on the order of 10^5 and 10^3, respectively.

Three HEP workflows, TopEFT, RSTriPhoton, and DV5, are used for evaluation. All runs use 640 total CPU cores (Workers $\times$ Cores/Worker). Per-worker memory and disk are fixed at 40 GB and 500 GB. We vary the number of workers and the cores per worker to control memory per core according to each workflow's memory intensity: TopEFT and RSTriPhoton are highly memory-intensive, while DV5 is moderate. Table I summarizes the DAG statistics and the resources used

TABLE I: Workflow configurations for experiments.

	Metrics	TopEFT	RSTriPhoton	DV5
DAG Info	Nodes	19,220	105,193	246,428
	Edges	19,190	141,895	403,200
	Depth	7	60	4
	Width	17,300	5,648	112,000
	Max Fan-In	10	2	800
	Max Fan-Out	1	4	5
	Sources	17,300	1,412	22,400
	Sinks	30	10	28
	Components	30	10	28
Resources	Workers	80	80	40
	Cores/Worker	8	8	16
	Memory/Worker [GB]	40	40	40
	Disk/Worker [GB]	500	500	500

Depth = longest path length; Width = maximum parallel width.
Max Fan-In/Out = largest in-/out-degree.
Resources denote requested capacities per worker node.

to run each workflow. We evaluate SciWIND along the three dimensions described in Section IV.

B. NLS Minimization Results

This section evaluates how excessive disk consumption causes node failures and how SciWIND reduces per-node disk usage to ensure successful completion. In default TaskVine settings, tasks are scheduled in a First-In-First-Out (FIFO) manner. We then incrementally enable SciWIND's three *NLS Minimization* strategies (Disk Load Shifting, Aggressive Pruning, and LIF Scheduling) to examine their individual and combined effects in preventing failures from NLS exhaustion, collectively referred to as *SciWIND-Min*.

TABLE II: Storage Consumption after applying SciWIND-Min.

	TopEFT	RSTriPhoton	DV5
TaskVine [GB]	14.22	502.65	500.19
SciWIND-Min [GB]	5.11	29.62	4.85
Reduction [%]	64.06	94.11	99.03

Percentages denote relative reduction compared to TaskVine.

Figure 5 shows the per-node storage consumption observed during the execution of the three workflows. As shown in Figure 5(a), storage usage becomes imbalanced as some workers accumulate excessive intermediate data, causing node failures in RSTriPhoton and DV5, while TopEFT remains unaffected due to low disk demand. Worker failure wipes all intermediate data on that node, thereby forcing upstream recomputation and extending the makespan. Figures 5(b)–(d) show the effect of incrementally enabling SciWIND *NLS Minimization* strategies. With Disk Load Shifting, data are periodically moved from overloaded to idle workers, lowering peak storage. Adding Aggressive Pruning cleans stale data and promptly frees up space. With LIF scheduling, data pruning is accelerated, and thus the peak storage is further reduced. Each strategy mitigates peak usage, and together they save substantial NLS storage space. Quantitatively, peak storage drops from $14.22\,\mathrm{GB}$ to $5.11\,\mathrm{GB}$ for TopEFT ($64.06\,\%$), from $502.65\,\mathrm{GB}$ to $29.62\,\mathrm{GB}$ for RSTriPhoton ($94.11\,\%$), and from $500.19\,\mathrm{GB}$ to $4.85\,\mathrm{GB}$ for DV5 ($99.03\,\%$), as can be seen in Table II. The reductions for RSTriPhoton and DV5 are conservative, since TaskVine runs hit the $500\,\mathrm{GB}$ node disk capacity, meaning the true percentages are even higher.

C. Efficient Recovery Results

This section measures how *Efficient Recovery* improves the scheduling efficiency when node failures are frequent and the scheduling queue is mixed with both eligible and ineligible tasks. We simulate frequent node failures by manually kicking off workers during execution. At every $2\,\%$ of task completion, one random worker is removed, totaling 50 evictions. We compare *SciWIND-Min*, which enables only *NLS Minimization*, against *SciWIND-Core*, which additionally integrates the *Efficient Recovery* strategies.

Figure 6 shows waiting and executing tasks at runtime for the three workflows. For TopEFT and RSTriPhoton, *Efficient Recovery* brings limited benefit. In TopEFT, the task count is small, waiting declines steadily, and executing stays near core saturation. In RSTriPhoton, waiting accumulates early as new tasks are submitted but soon clears, while executing remains saturated, indicating timely discovery of runnable tasks.

By contrast, in DV5, this scheduling overhead becomes evident. Under *SciWIND-Min*, the first half follows trends similar to (b), but waiting surges midway as early data losses later become inputs for bottom-level tasks. The mix of eligible and ineligible tasks overwhelms the scheduler, reducing concurrency and lengthening the makespan. With *Efficient Recovery*, ineligible tasks are isolated and recovery tasks promptly prioritized, so the ready queue remains rich in runnable tasks, concurrency tracks available cores, waiting decreases steadily, and the makespan shortens substantially. Despite a comparable number of recovery tasks (120,040 under SciWIND-Min versus 155,083 under *SciWIND-Core*), the makespan drops drastically from 7,017 to 2,098s ($70.1\,\%$), which indicates that performance gains come from scheduling efficiency rather than recovery volume. As a complementary observation, Figure 7 shows that in DV5, recovery tasks under SciWIND-Min accumulate early, surge midway, and then decline, revealing slowdown from recovery overhead, whereas *SciWIND-Core* swiftly schedules recovery and keeps the queue compact.

D. Resilience Reinforcement Results

This section evaluates how *Resilience Reinforcement* improves system resilience by reducing the need for recomputation. We keep the same worker eviction policy as in Section V-C, where one random worker is removed at every $2\,\%$ of task completion. We use *SciWIND-Core* as the reference configuration. On top of this, we separately enable the SciWIND *Resilience Reinforcement* strategies and a series of parameter sweeps to examine their individual effects.

We do not tune parameters to individual workflows; instead, we select representative values based on coarse DAG properties

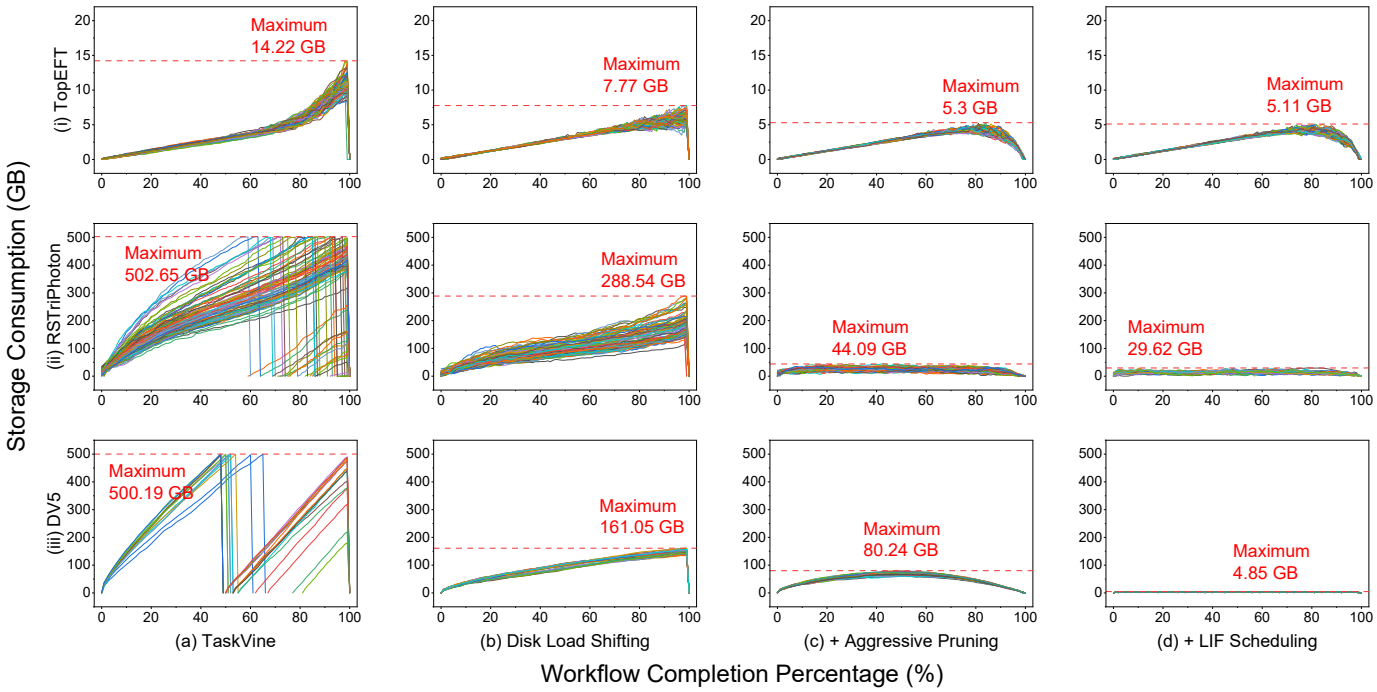

Fig. 5: Storage consumption per worker across workflows. Each line shows single worker's NLS usage over workflow progress (x-axis: progress percentage, y-axis: NLS consumption in GB*). Peak node usage drops with more aggressive strategies.*

and reuse them across all workloads. Performance is sensitive mainly to extreme settings, while a broad middle range yields similar behavior, indicating the parameters act as robustness knobs rather than fine-grained optimizers.

We further combine some representative parameter choices to show how integrating the three strategies yields stronger resilience than any single one. We also report results from *TaskVine* and *SciWIND-Min*, while *SciWIND-Core* serves as the comparison point for this evaluation.

Table III reports end-to-end metrics for the three workflows, in which each value is the average of 3 repetitions. Among the four metrics, *Recovery Task Count* and *Makespan* capture the benefits of each strategy, as fewer recovery tasks and shorter makespan indicate higher effectiveness, while *Storage Peak* and *Pruning Overhead* expose the trade-offs: stronger *Resilience Reinforcement* requires redundancy that increases storage consumption, and more replicas together with frequent checkpointing raise garbage collection overhead. We report the results below.

1) ***Pruning Depth****:* Compared to *SciWIND-Core* (depth$=1$), deeper pruning barely moves the needle for shallow DAGs such as TopEFT and DV5 but delivers substantial gains for deep ones such as RSTriPhoton. TopEFT (depth $=7$) reduces RTC from $5{,}417$ to $4{,}716$ at PD4 ($-12.94\,\%$) and DV5 (depth $=4$) from $29{,}921$ to $24{,}617$ at PD3 $-17.7\,\%$); both improvements are modest and non-monotonic. RSTriPhoton (depth $=60$) achieves the largest reduction, collapsing RTC from $20{,}103$ to $5{,}342$ at PD6 ($-73.4\,\%$), as retaining deeper ancestors protects large descendant sets.

For makespan, RSTriPhoton is also the clear winner: it decreases monotonically and reaches the global minimum of $1{,}998\,$s at PD6 ($-29.9\,\%$) across all strategies. DV5 improves to $3{,}037\,$s at PD5 ($-15.7\,\%$) and TopEFT to $3{,}771\,$s at PD3 ($-3\,\%$), but both remain well above RSTriPhoton and offer limited benefit relative to the best strategy.

The trade-off mainly lies in NLS consumption, which rises steadily with depth. It grows from 5 to 7 GB in TopEFT, 35 to 112 GB in RSTriPhoton, and 5 to 144 GB in DV5. Pruning overhead, by contrast, remains within seconds and is negligible.

2) ***Replication Count****:* Compared to the *SciWIND-Core* (RC$=1$), increasing replication count sharply reduces RTC across all workflows. TopEFT almost eliminates recoveries, dropping from $5{,}417$ to just 11 tasks at RC6 $-99.8\,\%$. DV5 also benefits substantially, falling from $29{,}921$ to $3{,}774$ at RC6 ($-87.4\,\%$). RSTriPhoton reaches its minimum at RC4, decreasing from $20{,}103$ to $1{,}659$ ($-91.7\,\%$), but then rises again at higher replication levels. Notably, these reductions far exceed those achieved with pruning depth, and even RC2 already provides substantial benefit, while further replication yields only marginal gains.

For makespan, TopEFT improves steadily to $2{,}875\,$s at RC6 ($-26.1\,\%$), and DV5 improves to $2{,}739\,$s at RC5 ($-23.9\,\%$). In contrast, RSTriPhoton shows only a transient improvement: its makespan drops from $2{,}848\,$s to $2{,}411\,$s at RC2, but then increases monotonically to $3{,}375\,$s at RC6.

The trade-offs are threefold. First, higher replication introduces communication and metadata management overhead, which can extend makespan, as seen in RSTriPhoton. Second, NLS faces heavier pressure as more replicas are retained, with RSTriPhoton increasing most significantly from 35 to 143 GB.

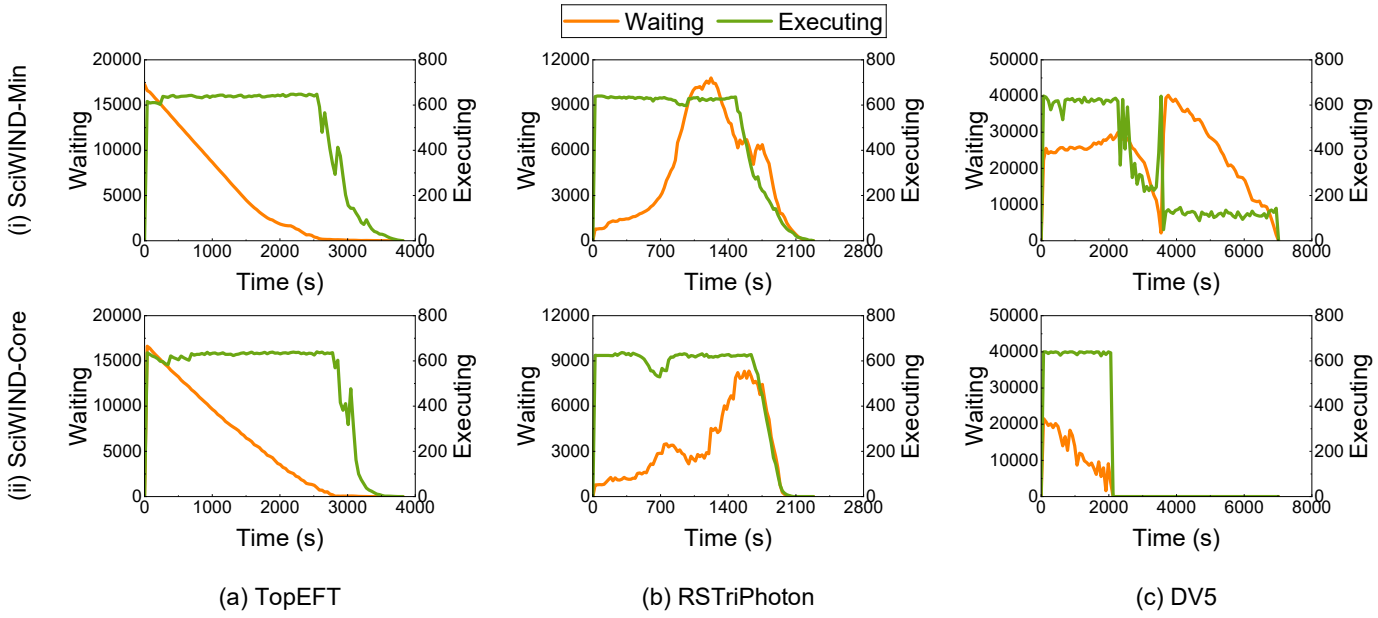

Fig. 6: Waiting and executing tasks during runtime under SciWIND-Min *(i) and* SciWIND-Core *(ii) across workflows.* SciWIND-Core *has little effect on TopEFT and RSTriPhoton, but significantly improves task concurrency and makespan in DV5.*

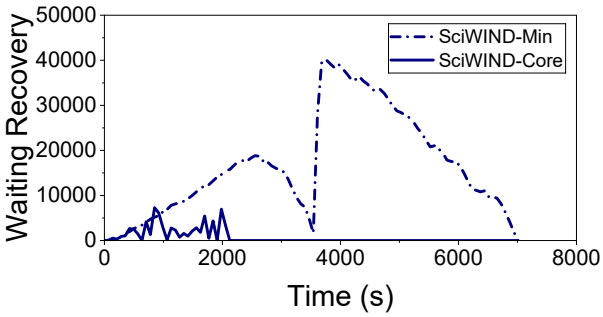

Fig. 7: Recovery tasks waiting during runtime in DV5. Efficient Recovery *eliminates the mid-execution surge of recovery tasks and thereby improves scheduling efficiency.*

Third, pruning overhead also rises slightly with RC but remains within seconds and is negligible relative to execution times.

3) ***Checkpoint Percentage****:* Compared to the *SciWIND-Core* (CP=0 %), increasing checkpoint frequency sharply reduces RTC across all workflows and reaches zero at 100 %. Early gains are already large: TopEFT drops from 5,417 to 4,305 at 10 % (−20.5 %), RSTriPhoton from 20,103 to 11,617 (−42.2 %), and DV5 from 29,921 to 2,621 (−91.2 %).

For makespan, effects are workload-dependent: TopEFT improves steadily to 2,884 s at 100 % (−25.8 %); DV5 reaches its minimum at 50 % (2,609 s, −27.6 %) but degrades at high ratios (100 %: 3,752 s, 4.2 %). RSTriPhoton improves only at 10 % (2,467 s, −13.4 %) and then increases monotonically to 4,991 s at 100 % (75.2 %).

The trade-offs are dominated by pruning overhead: it rises steeply with higher checkpoint ratios and becomes prohibitive at the high end. The dominant cost is deletion: the manager unlinks checkpointed files serially against the PFS metadata service, so latency accumulates. A second cost appears under heavy concurrency: all intermediate data must be read from the PFS; large numbers of concurrent reads and writes create I/O contention and can increase makespan, though this effect is not pronounced in Table III. Storage peak remains low because the burden is shifted to the PFS.

4) ***Hybrid Strategies****:* To highlight the difference between individual strategies and their combinations, we select three representative hybrid settings (Hybrid-1/2/3). They are not necessarily optimal but serve to illustrate typical gains and trade-offs. Table III shows that even the most lightweight combination (Hybrid-1: PD2+RC2+CP10%) already achieves dramatic benefits. For DV5, RTC plummets from 29,921 to just 66 (−99.8 %), while makespan shortens from 3,601 s to 2,590 s (−28.1 %), all with low storage peak and pruning overhead. By comparison, more aggressive combinations (Hybrid-2 and Hybrid-3) deliver only marginal additional reduction in RTC, yet steadily inflate storage peak and pruning overhead. Similar but less pronounced trends hold for TopEFT and RSTriPhoton. This indicates that simple lightweight hybrids can provide excellent tolerance at low cost, and thus Hybrid-1 represents the default configuration of SciWIND.

E. At-scale Results

We further validate SciWIND with an at-scale "hero run" on DV5 under frequent failures, using it as a representative workload. Compared to the scaled-down setting, this hero run processes roughly $\sim 5\times$ more input data while preserving the same workflow structure. To model failure-prone execution, we adopt the same eviction policy as earlier sections: at every 2 % of completion progress, we randomly evict one worker,

TABLE III: End-to-end workflow metrics by applying SciWIND Resilience Reinforcement *strategies.*

		Recovery Task Count			Makespan [s]			Storage Peak [GB]			Pruning Overhead [s]		
		TopEFT	RSTriPhoton	DV5	TopEFT	RSTriPhoton	DV5	TopEFT	RSTriPhoton	DV5	TopEFT	RSTriPhoton	DV5
TaskVine		4,357	2,928	184,765	3,861	2,356	9,965	15	503	500	0	0	0
SciWIND-Min		5,947	20,158	33,308	3,882	2,279	6,987	5	53	17	1	4	10
SciWIND-Core		5,417	20,103	29,921	3,888	2,848	3,601	5	35	5	1	4	9
PD	2	5,096	18,241	30,144	4,115	2,200	3,412	6	78	68	1	5	8
	3	4,902	9,845	24,617	3,771	2,122	3,192	7	85	101	1	5	4
	4	4,716	7,440	25,041	3,887	2,041	3,215	7	100	144	1	5	10
	5	4,796	6,435	25,139	3,787	2,036	3,037	7	112	148	0	5	2
	6	4,863	5,342	25,651	3,803	1,998	3,044	7	112	143	0	6	2
RC	2	1,143	3,893	16,909	3,270	2,411	2,837	8	55	6	1	6	13
	3	371	2,281	13,870	3,070	2,559	2,952	11	73	7	1	8	16
	4	103	1,659	7,924	2,937	2,840	2,799	14	92	8	1	11	18
	5	31	2,108	4,225	2,908	3,191	2,739	17	112	10	2	10	21
	6	11	2,607	3,774	2,875	3,375	2,869	21	143	12	2	11	24
CP	10 %	4,305	11,617	2,621	3,822	2,467	2,976	4	51	6	8	374	536
	20 %	3,937	11,264	2,944	3,737	2,788	3,158	4	49	6	16	706	672
	30 %	3,339	10,896	2,275	3,684	2,825	2,838	4	46	5	22	926	465
	40 %	3,002	10,751	2,151	3,692	2,828	2,830	3	47	5	34	961	652
	50 %	2,734	4,807	2,005	3,634	2,872	2,609	3	63	5	41	1,111	679
	60 %	2,007	1,811	1,784	3,538	2,892	2,621	3	77	6	46	1,436	873
	70 %	1,563	1,334	1,170	3,543	3,054	2,766	2	72	6	60	1,570	1,024
	80 %	1,080	731	446	3,420	3,250	3,509	1	72	6	71	1,689	1,700
	90 %	511	372	29	3,297	3,783	2,963	1	64	4	87	2,157	1,378
	100 %	0	0	0	2,884	4,991	3,752	0	0	0	92	3,117	1,948
Hybrid-1 (PD2+RC2+CP10 %)		1,090	1,402	66	3,173	2,507	2,590	7	72	10	8	367	169
Hybrid-2 (PD3+RC3+CP20 %)		310	1,787	43	3,132	3,374	2,453	10	140	40	24	534	662
Hybrid-3 (PD4+RC4+CP30 %)		120	1,199	45	2,937	4,108	2,734	11	282	99	35	743	398

PD: Prune Depth; RC: Replication Count; CP: Checkpoint Percentage.
SciWIND-Core serves as the reference configuration, with PD=1, RC=1, and CP=0 %.
All runs are with human-injected failures.

which triggers failure recovery and may substantially increase the number of recovery tasks and the overall makespan.

We compare the baseline *TaskVine* against *SciWIND-Hybrid-1* (PD2+RC2+CP10 %). Figure 8 reports two metrics over time: (i) *total storage consumption across all workers*, and (ii) *completed tasks over runtime*, which includes both regular tasks and recovery tasks. Under this failure condition, DV5 contains 1,196,555 regular tasks. With SciWIND, the run executes 197,241 recovery tasks in total. By contrast, the baseline executes 3,647,891 total tasks, including 2,451,336 recovery tasks, indicating severe recomputation. Overall, SciWIND reduces recovery tasks by 91.95 %.

These reductions translate into substantial end-to-end improvements. The makespan drops from 37,921 s to 10,946 s (71.13 %). Meanwhile, total NLS consumption across all workers decreases from 8,149 GB to 428 GB (94.75 %), confirming that SciWIND keeps the distributed NLS footprint bounded even under repeated worker losses. Taken together, the hero run demonstrates that lightweight hybrid policies can simultaneously suppress storage growth, curb recomputation, and shorten the tail at a substantially larger scale.

VI. Related Work

SciWIND is a workflow-integrated framework for managing node-local caches in large-scale scientific workflows. Prior works offer strong building blocks, but they typically treat storage and recovery as separate concerns or pushes policy outside the workflow engine. SciWIND's core contribution is the end-to-end integration: one WMS-level control loop that coordinates placement, pruning, scheduling, and recovery around NLS locality and failure behavior. We therefore present related work mechanism-by-mechanism, clarifying which strategies SciWIND adopts, extends, or introduces, and why.

Disk Load Shifting. Traditional distributed storage like Ceph [22] uses the CRUSH [33] algorithm to statically place data; upon device failures it only remaps the lost device's contents . Similarly, BeeOND [34] (BeeGFS [21] on Demand) creates an ephemeral PFS on job nodes but does not adapt during execution. Data Jockey [35] automates bulk data placement across tiers based on workflow hints, but it plans moves in advance rather than reacting to runtime cache imbalance. In contrast, SciWIND's Disk Load Shifting

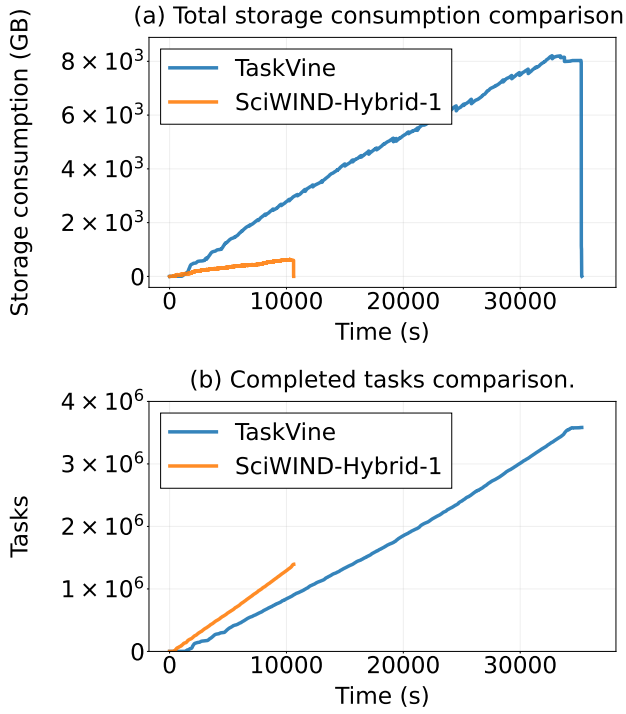

Fig. 8: DV5 hero run at scale under frequent worker evictions, comparing baseline TaskVine *and* SciWIND-Hybrid-1. *(a) Total storage consumption across workers: SciWIND keeps the total NLS consumption bounded while the baseline accumulates intermediate data over time. (b) Completed tasks over runtime, including both regular and recovery tasks: the baseline performs substantial recomputation after failures, whereas SciWIND suppresses recovery cascades and finishes earlier.*

continuously migrates intermediate files from overloaded to idle workers during execution, evening out hot spots on-the-fly. This workflow-engine-level data migration (deciding what to move and where, at runtime) has no direct analog in prior systems, making it a novel mechanism.

Depth-Aware Pruning. Workflow engines often delete intermediates once they are no longer needed. For example, Pegasus [36] and Snakemake [37] garbage-collect outputs as soon as downstream tasks complete, and Nectar [38] (for iterative analytics) similarly recycles data. SciWIND adopts this aggressive cleanup by evicting a file right after its last consumer finishes, thus bounding the NLS footprint. The novelty lies in tuning this behavior for failure-prone, skewed caches. SciWIND extends pruning with Depth-Aware Pruning: by default it deletes intermediates immediately (depth=1), but it can retain data for up to k downstream generations. With larger k, more ancestors are kept (reducing recomputation at the cost of storage); with $k = 1$, only last-generation data is kept (minimizing disk use). This user-adjustable retention horizon is new in workflow engines, giving a smooth trade-off between storage use and recompute overhead.

Largest-Input-First Scheduling. Classic schedulers emphasize data locality or computational fairness. Hadoop's [39] Delay Scheduling [40] and Quincy [41] prioritize locality, while general task scheduling might use Shortest/Longest-Job-First for CPU efficiency. SciWIND takes a different angle: it gives higher priority to ready tasks with larger total cached input size. Intuitively, this runs big-input tasks sooner so they release their large data quickly, shrinking peak cache usage. To our knowledge, no previous scheduler optimizes explicitly for minimizing NLS footprint. Even the recent WOW [42] scheduler, which speculatively replicates data for locality, aims to reduce network I/O, not to reduce storage peaks. SciWIND's Largest-Input-First policy is therefore a novel objective targeting disk-usage reduction.

Multi-Queue Dispatcher. High-throughput schedulers like Sparroww [43] and Falkon [44] decouple dispatch from global state by sampling or multi-queue designs. SciWIND extends this idea to handle dependency blocking after failures. When a node crashes, many tasks may appear ready but their inputs are missing. Our Multi-Queue Dispatcher keeps two queues: a pending queue for tasks waiting on inputs, and a ready queue for runnable tasks. The dispatcher then ignores blocked tasks, focusing only on truly eligible ones. This queue separation under failures is a novel twist: it lets the scheduler progress without scanning tasks that cannot run, improving dispatch scalability in failure scenarios.

Proactive Recovery. Lineage-based recovery is well-known from MapReduce [45] and Spark [46]. Spark's RDDs and Ray [47] recover lost data lazily by recomputing partitions when accessed. SciWIND extends this by being proactive: upon detecting NLS data loss, it immediately submits recompute tasks for all missing files (rather than waiting for their use) and effectively reduces the tail latency under failure. This overlaps recovery with normal execution, akin to asynchronous database rollback.

Prioritized Recovery. Prior recovery work largely focuses on *how* to recompute lost data via lineage (e.g., Spark [46], Ray [47]) or *what* to protect via checkpointing/replication analyses (e.g., Aupy [48], Benoit et al. [49]). These studies do not explicitly address a queueing pathology under aggressive recovery: recursive upstream submissions can create a long recovery backlog where early-submitted recovery tasks are often blocked, while later-submitted ones are more likely runnable. SciWIND targets this specific issue by prioritizing recovery tasks and reversing priority *within* a recovery cascade (LIFO by submission order), preventing starvation and restoring forward progress when recovery pressure is high.

Peer Replication. Replicating data to peer nodes is a standard resilience technique. HPC checkpoint libraries such as SCR [50] and FTI [51] use partner-style replication, and data-parallel systems often duplicate key blocks for availability. SciWIND applies replication to workflow intermediates on NLS, keeping up to ω replicas across workers. Our extension is a simple but effective control policy: replication is throttled to bound

bandwidth, and files with fewer existing replicas are prioritized for copying to quickly raise the minimum redundancy level. The underlying idea is established, but the replica-count-aware prioritization is specific to SciWIND's NLS setting.

PFS Checkpointing. Flushing state from fast local storage to a shared PFS is a classic durability layer. Multi-level checkpointing systems such as SCR [50], FTI [51], and VeloC [52] selectively and asynchronously persist critical data beyond ephemeral tiers. SciWIND extends this pattern in a workflow context by using a per-task heavy score to decide what to persist: tasks with higher impact on downstream recomputation are preferentially checkpointed to the PFS when they complete. This keeps the mechanism lightweight while making PFS writes goal-directed, rather than periodic or uniform, under the workflow engine's control.

VII. Discussion and Conclusion

Node-local storage offers substantial performance benefits for data-intensive workflows but also poses critical risks: unmanaged usage can exhaust disk capacity, random failures can crash nodes, and these crashes often lead to costly recovery. Our evaluation highlights how SciWIND addresses these challenges through coordinated strategies: *NLS Minimization* avoids NLS exhaustion and worker crashes, *Efficient Recovery* shortens response times by carefully scheduling recovery tasks, and *Resilience Reinforcement* mitigates recomputation costs through replication, pruning, and checkpointing. Hybrid strategies demonstrate that lightweight combinations capture most of the benefits at substantially lower cost than aggressive single-strategy deployments.

One important consideration is that SciWIND is deeply integrated into TaskVine's execution model, which makes direct adoption by other workflow systems nontrivial; however, the design workflow and decision process are transferable, and individual systems can selectively incorporate similar mechanisms. Future work will study how workflow structural properties interact with parameters such as pruning depth, replication degree, and checkpoint ratio, and develop adaptive mechanisms that automatically tune these parameters based on DAG characteristics and failure patterns. We also plan to evaluate SciWIND across broader workflow domains and gradually abstract it into a reusable framework that other WMSs can adopt more directly. Users can reproduce the results using the public TaskVine codebase [53], following the TaskVine manual [54], and applying the configuration parameters described in this paper. Overall, by enabling workflows to exploit NLS effectively while maintaining resilience, SciWIND provides a practical foundation for robust large-scale workflows and informs the design of future exascale systems.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Award Nos. OAC-1931348 and OAC-2411436. This work was also supported by a CMS Fellowship and by NSF Cooperative Agreement PHY-2121686.

References

[1] G. B. Berriman, E. Deelman, J. C. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. C. Laity, T. A. Prince, G. Singh, and M.-H. Su, "Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand," in *Optimizing scientific return for astronomy through information technologies*, vol. 5493. SPIE, 2004, pp. 221–232.

[2] K. A. Oca
textasciitilde na, D. de Oliveira, F. Horta, J. Dias, E. Ogasawara, and M. Mattoso, "Exploring molecular evolution reconstruction using a parallel cloud based scientific workflow," in *Advances in Bioinformatics and Computational Biology: 7th Brazilian Symposium on Bioinformatics, BSB 2012, Campo Grande, Brazil, August 15-17, 2012. Proceedings 7*. Springer, 2012, pp. 179–191.

[3] M. I. Love, S. Anders, V. Kim, and W. Huber, "Rna-seq workflow: gene-level exploratory analysis and differential expression," *F1000Research*, vol. 4, p. 1070, 2015.

[4] A. Abramovici, W. E. Althouse, R. W. Drever, Y. Gürsel, S. Kawamura, F. J. Raab, D. Shoemaker, L. Sievers, R. E. Spero, K. S. Thorne *et al.*, "Ligo: The laser interferometer gravitational-wave observatory," *science*, vol. 256, no. 5055, pp. 325–333, 1992.

[5] B. Sly-Delgado, B. Tovar, J. Zhou, and D. Thain, "Reshaping high energy physics applications for near-interactive execution using taskvine," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 1–13.

[6] P. W. Hatfield, J. A. Gaffney, G. J. Anderson, S. Ali, L. Antonelli, S. Başeğmez du Pree, J. Citrin, M. Fajardo, P. Knapp, B. Kettle *et al.*, "The data-driven future of high-energy-density physics," *Nature*, vol. 593, no. 7859, pp. 351–361, 2021.

[7] C. S. Liew, M. P. Atkinson, M. Galea, T. F. Ang, P. Martin, and J. I. V. Hemert, "Scientific workflows: moving across paradigms," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–39, 2016.

[8] Z. Wang, Z. Sun, H. Yin, X. Liu, J. Wang, H. Zhao, C. H. Pang, T. Wu, S. Li, Z. Yin *et al.*, "Data-driven materials innovation and applications," *Advanced Materials*, vol. 34, no. 36, p. 2104113, 2022.

[9] S. Usman, R. Mehmood, and I. Katib, "Big data and hpc convergence: The cutting edge and outlook," in *International Conference on Smart Cities, Infrastructure, Technologies and Applications*. Springer, 2017, pp. 11–26.

[10] A. Bashyal, P. Van Gemmeren, S. Sehrish, K. Knoepfel, S. Byna, and Q. Kang, "Data storage for hep experiments in the era of high-performance computing," *arXiv preprint arXiv:2203.07885*, 2022.

[11] O. Tatebe, S. Moriwake, and Y. Oyama, "Gfarm/bb—gfarm file system for node-local burst buffer," *Journal of Computer Science and Technology*, vol. 35, pp. 61–71, 2020.

[12] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. Foster, "Design and analysis of data management in scalable parallel scripting," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[13] S. Perarnau, J. A. Zounmevo, M. Dreher, B. C. Van Essen, R. Gioiosa, K. Iskra, M. B. Gokhale, K. Yoshii, and P. Beckman, "Argo nodeos: Toward unified resource management for exascale," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 153–162.

[14] 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.

[15] A. Basnet, K. Bloom, F. Canelli, S. Sanchez Cruz, J. E. Palencia Cortezon, J. R. González Fernández, A. Trapote Fernandez, R. Goldouzian, B. Alvarez Gonzalez, M. Hildreth *et al.*, "Topeft/topcoffea: Topcoffea 0.1," *Zenodo*, 2021.

[16] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005, doi: 10.1002/cpe.v17:2/4.

[17] C. collaboration *et al.*, "Development of the cms detector for the cern lhc run 3," *Journal of Instrumentation*, vol. 19, no. 5, p. P05064, 2024.

[18] H. S. F. hsf-editorial-secretariat@ googlegroups. com, J. Albrecht, A. A. Alves, G. Amadio, G. Andronico, N. Anh-Ky, L. Aphecetche, J. Apostolakis, M. Asai, L. Atzori *et al.*, "A roadmap for hep software and computing r

&d for the 2020s," *Computing and software for big science*, vol. 3, pp. 1–49, 2019.

[19] J. Shiers, F. O. Berghaus, G. Cancio Melia, J. Blomer, S. Dallmeier-Tiessen, G. Ganis, and T. Simko, "Cern services for long term data preservation," CERN, Tech. Rep., 2016.

[20] P. Braam, "The lustre storage architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[21] J. Heichler, "An introduction to beegfs," *Introductionto BeeGFS by ThinkParQ. pdf*, 2014.

[22] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, 2006, pp. 307–320.

[23] VAST Data, "The vast datastore," https://vastdata.com/whitepaper/#ThePromiseofAI-EnabledDiscovery, [Online; accessed 2025-10-02].

[24] A. Al-Shafei, H. Zareipour, and Y. Cao, "High-performance and parallel computing techniques review: Applications, challenges and potentials to support net-zero transition of future grids," *Energies*, vol. 15, no. 22, p. 8668, 2022.

[25] M. Arifuzzaman and E. Arslan, "Online optimization of file transfers in high-speed networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis, MO, USA.: ACM, 2021, pp. 1–13.

[26] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touri
textasciitilde no, and R. Doallo, "Performance analysis of hpc applications in the cloud," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 218–229, 2013.

[27] T. Herault, Y. Robert, A. Bouteiller, A. Arnold, K. B. Ferreira, G. George, and J. Dongarra, "Checkpointing strategies for shared high-performance computing platforms," *International Journal of Networking and Computing*, vol. 9, no. 1, pp. 28–52, 2019.

[28] G. Singh, K. Vahi, A. Ramakrishnan, G. Mehta, E. Deelman, H. Zhao, R. Sakellariou, K. Blackburn, D. Brown, S. Fairhurst *et al.*, "Optimizing workflow data footprint," *Scientific Programming*, vol. 15, no. 4, pp. 249–268, 2007.

[29] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley *et al.*, "End-to-end i/o portfolio for the summit supercomputing ecosystem," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[30] T. S. Phung, C. Thomas, L. Ward, K. Chard, and D. Thain, "Accelerating Function-Centric Applications by Discovering, Distributing, and Retaining Reusable Context in Workflow Systems," in *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2024.

[31] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay, "Accelerating large-scale data exploration through data diffusion," in *Proceedings of the 2008 international workshop on Data-aware distributed computing*, 2008, pp. 9–18.

[32] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–11.

[33] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, pp. 122–es.

[34] ThinkParQ GmbH, "Beeond: Beegfs on demand," https://doc.beegfs.io/latest/advanced_topics/beeond.html, [Online; accessed 2025-10-02].

[35] W. Shin, C. D. Brumgard, B. Xie, S. S. Vazhkudai, D. Ghoshal, S. Oral, and L. Ramakrishnan, "Data jockey: Automatic data management for hpc multi-tiered storage systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 511–522.

[36] 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.

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

[38] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: automatic management of data and computation in datacenters," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[39] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[40] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 265–278.

[41] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 261–276.

[42] F. Lehmann, J. Bader, F. Tschirpke, N. De Mecquenem, A. Lößer, S. Becker, K. E. Lewińska, L. Thamsen, and U. Leser, "Wow: Workflow-aware data movement and task scheduling for dynamic scientific workflows," in *2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2025, pp. 525–538.

[43] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 69–84.

[44] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a fast and light-weight task execution framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–12.

[45] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[46] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing," in *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, 2012, pp. 15–28.

[47] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 561–577.

[48] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Impact of fault prediction on checkpointing strategies," *arXiv preprint arXiv:1207.6936*, 2012.

[49] A. Benoit, A. Cavelan, F. M. Ciorba, V. Le Fèvre, and Y. Robert, "Combining checkpointing and replication for reliable execution of linear workflows with fail-stop and silent errors," *International Journal of Networking and Computing*, vol. 9, no. 1, pp. 2–27, 2019.

[50] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[51] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–32.

[52] M. Jami, "Optimizing checkpoint/restart and input/output for large scale applications," Ph.D. dissertation, Mathematisch-Naturwissenschaftliche Fakultät, 2024.

[53] Cooperative Computing Lab, "CCTools TaskVine Source Code," 2026, accessed: 2026-02-16. [Online]. Available: https://github.com/cooperative-computing-lab/cctools/tree/master/taskvine

[54] ——, *TaskVine User Manual*, 2026, accessed: 2026-02-16. [Online]. Available: https://cctools.readthedocs.io/en/latest/taskvine/

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

The paper evaluates SciWIND on TopEFT, RSTriPhoton, and DV5; the public artifact package is DV5-focused and reproduces the same mechanisms and analysis workflow.

A. Paper's Main Contributions

- C_1 We design *SciWIND*, an integrated framework in TaskVine for robust node-local storage (NLS) management in data-intensive scientific workflows.
- C_2 We propose and implement three coordinated components: *NLS Minimization*, *Efficient Recovery*, and *Resilience Reinforcement* (with knobs PD, RC, CP).
- C_3 We experimentally validate SciWIND under injected failures and show improvements in storage peak, recovery behavior, and end-to-end performance.

B. Computational Artifacts

- A_1 **Project entry artifact (orchestration and step-by-step guidance).** DOI: https://doi.org/10.5281/zenodo.18707521
- A_2 **TaskVine report tool artifact (log parsing).** DOI: https://doi.org/10.5281/zenodo.18706027
- A_3 **SciWIND/CCTools implementation artifact.** DOI: https://doi.org/10.5281/zenodo.18706040
- A_4 **DV5 input dataset artifact.** DOI: https://doi.org/10.5281/zenodo.18665133

Art.	Contrib.	Paper elems.
A_1	C_2, C_3	Full DV5 reproduction entry pipeline and regenerated appendix outputs
A_2	C_3	Parsed CSVs that drive Fig. 5/6/7 style trends and Table III style metrics
A_3	C_1, C_2	Runtime mechanisms in Sec. IV (*SciWIND* design/implementation)
A_4	C_3	Public DV5 input used by reruns and trend validation

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

Artifact A_1 is the *project entry* for the full reproduction workflow. It provides run/analysis scripts and step-by-step guidance in `README.md`, and connects artifacts A_2–A_4.

Expected Results

Expected outcomes:

- `data/plots/nls_minimization.png`
- `data/plots/efficient_recovery.png`
- `data/plots/resilience_reinforcement.csv`

Expected Reproduction Time (in Minutes)

Artifact Setup: 30–90 min.
Artifact Execution: tens of hours to several days (depends on cluster size and repeat count).
Artifact Analysis: 5 to 20 hours (depends on log volume and repeat count).

Artifact Setup (incl. Inputs)

Hardware: Linux/HTCondor-style environment with manager + workers. The experiments must run on a PFS/shared filesystem visible from all participating nodes, so every node can access the same working directory and log/output paths.

Software: Project entry DOI: https://doi.org/10.5281/zenodo.18707521.
Dependencies: A_2 (https://doi.org/10.5281/zenodo.18706027), A_3 (https://doi.org/10.5281/zenodo.18706040), and A_4 (https://doi.org/10.5281/zenodo.18665133).
`README.md` provides step-by-step setup, execution, and analysis guidance.

Datasets / Inputs: DV5 input from A_4 and logs generated during execution.

Installation and Deployment: Main commands: `bash create_conda_env.sh`, `bash download_input_dataset.sh`, `python ecf_calculator.py --preprocess -M dv5-manager --libcores 16`, and `python ecf_calculator.py --all --checkpoint-to data/dv5.pkl`.
Then run `part1_nls_minimization.sh`, `part2_efficient_recovery.sh`, `part3_resilience_reinforcement.sh`, parse logs, and run plotting scripts.

Artifact Execution

$$T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$$

- T_0 Setup env/runtime (`create_conda_env.sh`) and data (`download_input_dataset.sh`).
- T_1 Generate DV5 graph checkpoint (`data/dv5.pkl`).
- T_2 Execute three experiment parts with repeats.
- T_3 Parse logs with `vine_parse -R --logs-dir data/logs`.
- T_4 Produce appendix outputs via three `plot_*.py` scripts.

Artifact Analysis (incl. Outputs)

Regenerated DV5 appendix outputs and the parsed CSV directories under `data/logs/.../csv-files/`.

B. Computational Artifact A_2

Relation To Contributions

Artifact A_2 (`taskvine-report-tool`) converts raw runtime logs into normalized CSV metrics for downstream analysis, supporting C_3.

Expected Results

Expected results are two-stage: (i) setup success marker: `vine_parse --version` prints version info; (ii) after parsing logs, each run template contains `csv-files/` with metrics such as `task_execution_details.csv`, `task_concurrency.csv`, `time_domain.csv`, and `worker_storage_consumption.csv`.

Expected Reproduction Time (in Minutes)

Artifact Setup: 5–10 min.
Artifact Execution: hours for large log trees (depends on number/size of runs).
Artifact Analysis: 5–20 min for downstream scripts.

Artifact Setup (incl. Inputs)

Hardware: Linux node that can access experiment log directories.

Software: `taskvine-report-tool` DOI: https://doi.org/10.5281/zenodo.18706027.

Datasets / Inputs: Input: raw TaskVine logs produced by experiment runs under `data/logs/...`.

Installation and Deployment: Install tool per artifact instructions, validate installation with: `vine_parse --version`. Then parse logs using: `vine_parse -R --logs-dir data/logs`.

Artifact Execution

Metric extraction dependency:

$$T_{\text{raw-logs}} \rightarrow T_{\text{parse-with-vine\_parse}} \rightarrow T_{\text{csv-files}}$$

The resulting CSVs are consumed by A_1 plotting/analysis scripts.

Artifact Analysis (incl. Outputs)

`csv-files/*.csv` per run template directory.

C. Computational Artifact A_3

Relation To Contributions

Artifact A_3 provides the SciWIND-enabled CCTools runtime implementation, supporting C_1 and C_2.

Expected Results

Expected outcome: a buildable CCTools runtime exposing SciWIND controls used by execution scripts.

Expected Reproduction Time (in Minutes)

Artifact Setup: 30–60 min.
Artifact Execution: N/A (runtime substrate).
Artifact Analysis: N/A.

Artifact Setup (incl. Inputs)

Hardware: Linux build machine.

Software: SciWIND/CCTools DOI: https://doi.org/10.5281/zenodo.18706040.

Artifact commit used by project entry scripts: `227b25fc7`.

Datasets / Inputs: No dataset input required.

Installation and Deployment: In project entry workflow, build/install is automated by: `bash create_conda_env.sh`, which checks out `227b25fc7`, builds, and installs CCTools.

Artifact Execution

Runtime dependency role:

$$T_{\text{A3-build}} \rightarrow T_{\text{A1-execution}}$$

where A_3 supplies the runtime substrate.

Artifact Analysis (incl. Outputs)

Compiled runtime binaries (e.g., `vine_worker`) available to the project pipeline.

D. Computational Artifact A_4

Relation To Contributions

Artifact A_4 provides the public DV5 input dataset required for reproduction runs, supporting C_3.

Expected Results

Expected outcome: DV5 input archive is downloaded/extracted and staged into the expected directory layout consumed by preprocessing scripts.

Expected Reproduction Time (in Minutes)

Artifact Setup: network-dependent (typically 10–60 min download).
Artifact Execution: N/A.
Artifact Analysis: N/A.

Artifact Setup (incl. Inputs)

Hardware: Disk space for archive and extracted dataset.

Software: DV5 dataset DOI: https://doi.org/10.5281/zenodo.18665133.

Downloader/extractor tools (`curl`, `tar` with zstd support).

Datasets / Inputs: `hgg.tar.zst` (∼22.1 GB).

Installation and Deployment: Within project entry workflow, dataset retrieval/staging is automated by: `bash download_input_dataset.sh`.

Artifact Execution

Data preparation dependency:

$$T_{\text{download}} \rightarrow T_{\text{extract}} \rightarrow T_{\text{replicate-input-copies}}$$

which feeds preprocessing and execution tasks in A_1.

Artifact Analysis (incl. Outputs)

Prepared dataset tree under `data/dv5-input-samples/`.

Artifact Evaluation (AE)

For full evaluation reproducibility, follow Artifact A1 (`README.md`) in the AD section and run the documented pipeline in order (environment setup, data setup, experiment execution, log parsing, and output generation). The expected results are the regenerated figures/tables and parsed CSV files listed in A1.