iCR: Intent-Driven Adaptable Checkpoint/Restart

Olga Kogiou*, Hariharan Devarajan†, Chen Wang‡, Weikuan Yu*, Kathryn Mohror†
*Florida State University, USA
ok22b, wyu3@fsu.edu
†Lawrence Livermore National Laboratory, USA
hariharandev1, mohror1@llnl.gov
‡Nanyang Technological University, Singapore
chen.wang@ntu.edu.sg

***Abstract*—Modern High-Performance Computing workflows rely on Checkpoint/Restart (C/R) mechanisms for execution continuity, but their inherent task and data heterogeneity demands adaptability from the underlying C/R solution. In a case study with Montage, C/R with static configurations leads to an I/O overhead of 1.6× and a storage footprint of 176 GB, compared to C/R with an adapted configuration. To this end, we develop intent-driven C/R (iCR), a toolkit that leverages I/O intents as a semantic abstraction that bridges *high-level workflow behaviors* with *low-level C/R policies*. Through a trace-driven pipeline, iCR extracts I/O intents from workflows and maps them to representative C/R configurations using a scoring model. This way, iCR reduces the complexity of manual adaptation by 7× while systematizing fine-grained, per-task checkpointing policies that are seamlessly deployed atop popular C/R libraries. Through experiments, we show that seven reusable intents can be extracted from six representative workflows and enable workflow-agnostic adaptable checkpointing policies. Our evaluation demonstrates that, with I/O intents, iCR is able to reduce I/O overhead by up to 2.6× and improves storage efficiency by 1.8× compared to static C/R configurations for heterogeneous workflows such as Montage.**

***Keywords*-Checkpoint/Restart, workflows, intents, adaptable performance**

I. Introduction

High-Performance Computing (HPC) applications have shifted from monolithic scientific simulations [1] to heterogeneous workflows that integrate simulations with data analytics and AI-driven applications [2]. These heterogeneous workflows consist of diverse, long-running tasks, occupying thousands of CPU/GPU processes [3]. These processes commonly experience failures, with a projected Mean Time Between Failures (MTBF) as low as ≈ 18 minutes in exascale systems [4], [5]. To ensure continuity, Checkpoint/Restart (C/R) [6] remains the most essential technique to safeguard their progress.

On leadership HPC systems, popular C/R libraries such as the Scalable Checkpoint/Restart (SCR) library [7], the Very-Low Overhead Checkpoint (VeloC) system [8], and the Adaptable I/O System (ADIOS) [9] implement a multi-tiered storage model. In this model, checkpoint data are first written to local storage, replicated across neighboring nodes for redundancy, and eventually flushed to lower tiers in the storage hierarchy. Currently, these C/R libraries apply uniform policies across all checkpoints in a running job. However, recent studies demonstrate that C/R mechanisms must evolve to adapt to the specific computation and data needs of workloads, as static configurations can inadvertently degrade overall application performance by up to 2.5× [10].

We define a *task* as an individual application process, and a *workflow* as a pipelined group of such tasks. For C/R libraries to become adaptable and support emerging HPC applications such as workflows, they must address three key challenges. First, workflow tasks exhibit heterogeneous execution times, which dictate restart behavior upon failures. Long-running tasks require stricter checkpoint guarantees, as they may run for hours, whereas short tasks complete in minutes [11]. Second, intermediate data vary in importance for workflow continuation [12]. While less critical workflow data could be passed via in-memory task messages, this requires manual source code changes and can still trigger cascading rollbacks in dependent tasks when intermediate checkpoints are not persisted [13]. Third, mapping heterogeneous tasks and data to optimal storage requires navigating the large C/R configuration space. Even selecting a cache location is non-trivial due to the deep HPC storage hierarchy with tiers of varying performance and persistence [7]. This heterogeneity in tasks and data, along with the configurability of C/R libraries, necessitates a shift from static to adaptable, workflow-aware checkpoint solutions that can be applied in a systematic way.

In recent years, I/O intents have emerged as a technique to express workload behaviors from a user's perspective [14], [15]. I/O intents have been defined to capture *"why"*, *"what"*, and *"how"* users perform specific I/O operations and used to guide caching, compression and data placement decisions. In this work, we extend their definition to include *"when"* data are produced and consumed to explicitly express the amount of re-computation that would be required in case they were lost during an interrupt. In workflows, early produced data consumed by further downstream tasks must persist longer, by being placed at persistent storage tiers and having stronger redundancy. We leverage these extended user I/O intents as a semantic vehicle to provide adaptable C/R configurations.

However, enabling intent-awareness to guide adaptability in C/R systems remains challenging. First, collecting user I/O intents requires significant workflow knowledge and manual code annotations with appropriate APIs, making the optimization still manual and error-prone. Second, even when such information is available, there exists no systematic mapping technique that can translate these high-level intents into C/R

related optimizations. Third, these optimizations must integrate seamlessly with existing C/R libraries that workflows and HPC applications are already linked with.

To address these challenges while enabling adaptability in C/R systems, we develop the intent-driven Checkpoint/Restart (iCR) toolkit. iCR utilizes user I/O intents as a semantic abstraction to bridge workflow-level behaviors to low-level C/R configurations. To achieve this, iCR employs the ***Intent Collector***, which extracts I/O intents from collected workflow I/O traces. iCR's ***Intent Mapper***, then, maps these intents to the appropriate C/R configurations and storage tiers using a lightweight scoring model to avoid exact, expensive optimizations. The Mapper internally runs the scalable ***Conflict Resolver*** and ***Constraint Applicator*** to resolve inconsistencies among configurations and satisfy system constraints. This way, iCR improves checkpointing performance by 1.6–2.6×, and reduces storage footprint by 1.8× for heterogeneous workflows such as Montage. Overall, we make the following main contributions:

1) We propose a different approach in C/R, where user intents are used as a semantic abstraction layer to drive adaptability. By identifying a core set of seven user I/O intents (Table I), we demonstrate that they transcend six representative workflows.
2) We design iCR, a modular toolkit that operates atop well-adopted C/R frameworks (e.g., SCR, VeloC) to adapt them at a fine granularity (per workflow data or task).
3) We implement iCR as a trace-driven pipeline that extracts intents and maps them to representative C/R configurations to eliminate the need for user intervention.

II. Background and Motivation

A. Heterogeneity in workflow tasks and data

Modern workflows, such as Machine Learning-driven Multi-scale MD Workflows [16] (MuMMI), consist of heterogeneous tasks with execution times ranging from seconds to indefinite durations. When nodes fail, tasks must either resume from checkpoints or restart entirely. While short tasks can tolerate restarts, losing progress on long-running tasks can increase the total makespan by up to 40% [13], necessitating the use of checkpoints to persist intermediate data.

However, these checkpoint data exhibit highly diverse system-level *behaviors*. For instance, workflow tasks in 1000 Genome [17] exchange up to 21 M files [18]. These files vary drastically in their underlying **data types** (e.g., `.bai` and `.fai`), **data sizes**, **lifespans**, and the **number of processes** and **threads** accessing them. Furthermore, they are accessed using specific **access patterns** (e.g., random, sequential, or strided), **transfer sizes**, and **system-level APIs** (e.g., POSIX, MPI-IO). Finally, their accesses are performed with distinct **I/O operations** (e.g., `write`, `read`) and characterized by varying **I/O intensities** (steady vs. bursty [2]). Manually managing these diverse data behaviors for I/O optimizations is infeasible, highlighting the **need for a higher-level systematic abstraction** to bridge this gap.

B. Configurability of Checkpoint/Restart

Widely adopted C/R libraries such as SCR [7] and VeloC [8] offer roughly 30 tunable parameters. Primary parameters include, checkpoint interval, the cache location or base, and the multiple redundancy schemes. For example, SCR provides redundancy schemes `PARTNER`, where processes replicate checkpoints on partner nodes, `XOR`, where a single parity block is generated via XOR across processes, and `Reed-Solomon (R-S)`, where multiple parity blocks are used. Additional tuning knobs manage the cache size, different flushing APIS (e.g. POSIX threads, Cray DataWarp [19], IBM Burst Buffer (BB) API [20]) and failure set sizes.

Despite this high configurability, these settings remain **static**, in practice, and are applied uniformly across all files and tasks in the workflow. Even in flexible systems like ADIOS [9], optimizing these parameters **requires manual adaptation** per I/O phase. In large-scale workflows producing millions of files, such as 1000 Genome with 21 M files [18], this leads to an explosive **configuration space** that makes manual tuning infeasible.

C. User I/O Intent

Historically, I/O intent was handled by the POSIX interface (`posix_fadvise` API), which allowed a single process to inform the kernel of its I/O access and data requirements to manage the Operating System cache efficiently. Previous works, Mimir [14] and H5Intent [15], have extended this single-process API for HPC parallel POSIX and HDF5 workloads. These frameworks provide APIs for users to express *"why"* I/O is performed (e.g., a simulation performing checkpoints), *"what"* specific target or object is being operated on (e.g., an HDF5 file), and *"how"* the I/O method is executed (e.g., using a 4 KB transfer size). Because these intents capture high-level semantics, they can be used for optimizations beyond the kernel, such as intelligent data placement decisions.

D. Benefits of Adaptable C/R for Montage workflow

To motivate an adaptable C/R strategy, we consider the Montage astronomical workflow [21], where system-centric behaviors vary significantly across tasks and data. Tasks *mDiffExec* and *mFitExec* generate 27% of data as partially computed Flexible Image Transport System (FITS) images via 24 KB sequential transfers. *mImgtbl* and *mMakeHdr* produce 8% as Table (TBL) and Header (HDR) metadata. In contrast, other tasks produce 60% of data as short-lived *text* (TXT) logs in small irregular chunks (4–24 KB). This heterogeneity reflects distinct underlying user **intents** for I/O.

We evaluate four SCR configurations on a 64-node Montage run using a 100 GB 2MASS input dataset [22] on Corona cluster located in LLNL [23] (see Section V-A1 for hardware specification). Fig. 1 y1-axis shows I/O time (results in bars) and y2-axis storage footprint (results with line) with variance over five runs. The default SCR setup uses NVMe cache, `XOR` redundancy, and `PTHREAD` asynchronous flushes to Lustre Parallel File System (PFS). In this configuration, redundancy is applied uniformly increasing storage by 1.14× over Lustre.

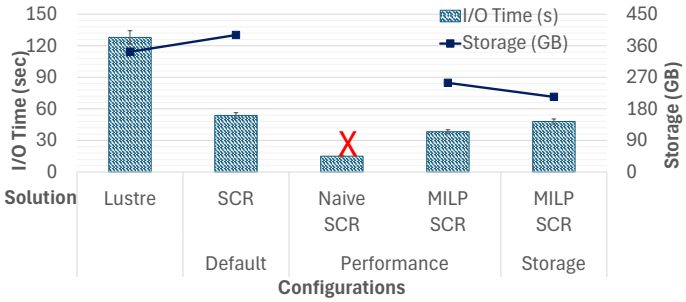


Fig. 1: Static SCR C/R policies applied uniformly to all I/O intents in Montage can degrade performance or even prevent workflow completion.

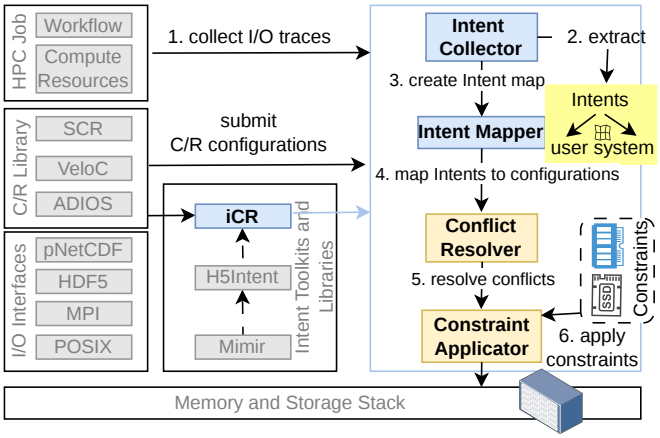


Fig. 2: iCR's position on the Software Stack and High-level Flowchart. iCR uses the intents identified in lower-level H5Intent and Mimir. iCR operates on top of C/R libraries.

To address this, one may choose a naive performance-oriented variant using DRAM caching, no redundancy, and PFS flushes reduces I/O time. As shown in Fig. 1, "Naive SCR" initially improves performance and storage but fails mid-execution due to DRAM exhaustion when TXT-producing tasks overlap.

To avoid uniform mis-configurations, users manually need to apply custom optimization, which is infeasible for all 37 K Montage files. To define an "ideal" baseline, we perform an I/O trace-based optimization using DFTracer [24] and DFAnalyzer [25]. Our analysis processes 348 MB of compressed I/O traces collected from Montage with 30+ Pandas-like queries on DFAnalyzer, a process that runs for **32 minutes** to collect the system-centric data behaviors identified in Section II-A. The result is a DataFrame of the collected data behaviors. We then use `PuLP` 3.3.0 Mixed-Integer Linear Programming (MILP) modeler [26] to map the collected data behaviors to SCR configurations, with a multi-objective function described in Section III-B1 (balancing performance, resilience and storage). The mapping process runs for **2.7 minutes** and **3.3 minutes** for performance and storage, respectively. The performance-optimized SCR ("MILP SCR Performance") improves I/O time by 1.62× but increases storage by 1.18× due to DRAM caching and `PARTNER` redundancy [27], which favors speed over storage efficiency for frequently accessed files. The storage-optimized variant ("MILP SCR Storage") reduces footprint by 1.83× by using `R-S` instead of `PARTNER` redundancy for files larger than 1 MB.

While effective, this approach is impractical, as it requires detailed workflow knowledge to implement manual queries for all 37 K files in Montage. Additionally, it is inefficient for future large workflows (e.g., 1000 Genomes with 21 M files) as even with all data behaviors collected, exact solvers may take hours to converge due to the combinatorial search over behaviors and possible configurations. This motivates the need for a principled semantic abstraction to automatically adapt C/R policies, to reduce both manual tuning and exact optimization complexity.

III. System Design

We propose a different approach for C/R in which workflow semantics determine granular C/R configurations with custom applicability to specific workflow tasks and data rather than uniformly. To this end, we develop *iCR**, a companion toolkit designed to bridge the gap between workflow-level semantics and C/R configurations.

iCR implements a semi-automated, end-to-end pipeline (Fig. 2). First, iCR allows the user to choose their preferred C/R library. iCR then employs the ***Intent Collector*** to extract user intents by analyzing workflow traces (Fig. 2, step 1), to alleviate the user from manual source code annotations that prior intent frameworks require [14], [15]. Additionally, iCR extends these works by introducing a decoupled taxonomy, separating intents into user-centric and system-centric components (Fig. 2, step 2). The user-centric component focuses on *"why"* the data are persisted, expressing the user's scientific purpose for checkpointing. In contrast, the system-centric component captures *"what"*, *"how"*, and *"when"* the data are accessed across processes and nodes, expressing the physical behavior observed by the system. Once the intents are identified (Fig. 2, step 3), iCR's ***Intent Mapper*** employs a scalable greedy mapping to associate the gathered intents with representative C/R configurations and their parameter values, avoiding computationally expensive optimizations (Fig. 2, step 4). Finally, the Mapper resolves configuration conflicts (using the ***Conflict Resolver***, Fig. 2, step 5) to address any parameter value inconsistencies, and applies physical constraints (using the ***Constraint Applicator***, Fig. 2, step 6) based on the system's storage and memory limits.

Design Philosophy iCR is built around three design principles: *(a) Flexibility*: iCR can operate atop any C/R library, adapting them to any workflow and system. *(b) Extensibility*: iCR extends generic user I/O intents (e.g., defined by Mimir [14] and H5Intent [15]) with new C/R-related semantics which can be easily expanded. *(c) Systematization*: iCR systematically extracts intents and maps them to C/R library configurations, eliminating per-file annotation and manual tuning. Functionally, iCR is designed as a companion toolkit that requires

*https://github.com/castl22/iCR

TABLE I: Seven workflow representative intents and their system-centric characteristics. Intents span across workflows.

User-centric Intent		**System-centric Intent**	
Why	*What*	*How*	*When*
Input dataset	YAML, JSON, FITS, NC, FSTQ, INP	Many procs/nodes, random access, `pread`, `read`, `fread`	job_start $<$ *access* $<$ first_comp
Partially Computed	FITS, VCF, PDB, FASTQ	Many procs/nodes, seq. access, `write` $\rightarrow$ `read`	in bursts
Metadata	HDR, TBL, VCF, BAM, CRAM	File-per-proc, `write`, `read`, `fwrite`, `fread`, `fputs`, `fgets`	freq. updated, long lifespan
Scratch / Logging	TXT, TMP	File-per-proc, seq. `write` / `read`	short lifespan
Anchor files	HTML, NC, VTK	File-per-proc, seq. `write`, seq. `read`	job_middle $<$ *access* $<$ job_end
Checkpoints	NC, PY, PT	File-per-proc, Write-only	In stable intervals, in bursts, short lifespan
Final result	PNG, VTU, VTR, CSV, TAR, GZ	Many procs/nodes, distributed, Write-only	job_end $<$ *access* $<$ job_end

the workflow to be previously coupled with a C/R library. Furthermore, iCR does not dictate *which* data must be persisted, as this selection reflects scientific requirements and remains the domain of the developer. Given that workflows have designated specific data for resilience, iCR's scope is adapting *how* these data are handled by the underlying C/R library.

A. Intent Collector

The Intent Collector translates raw workflow I/O traces into high-level logical intents. To achieve this, the Collector first abstracts file paths and numeric suffixes to group individual task files into *logical datasets*. It then performs Pandas-like queries [28] to capture system-centric data behaviors by utilizing its *parallel query engine*. This engine dynamically parses I/O traces in memory, partitioning them into threads to capture the ten behaviors described in Section II-A, which are then normalized and projected in a high-dimensional space. The Collector then employs *parallel K-means clustering* to group logical datasets with similar system-centric behaviors using *Euclidean distance* to express user-centric I/O intents.

The centroids of the clusters are initialized using domain knowledge from [2]. This work has guided us to define the seven representative C/R related user-centric intents presented in Table I using various scientific and AI workflows (e.g., Montage, MuMMI, 1000 Genome, ICECap [29], iPIC-3D [30], Megatron-DeepSpeed [31]). Partially computed data (Table I, row 2) are characterized by sequential write/read bursts across many tasks. Metadata (Table I, row 3) coordinate computed results and are small, long-lived and frequently updated. Finally, anchor files (Table I, row 5) are seen in data analytics workflows (e.g., MuMMI and iPIC-3D) to track the progress of the workflow tasks. The resulting clusters for these workflows have been checked using the related literature [2] as ground truth. Because our system-centric behaviors are highly discriminative, K-means algorithm exhibits rapid convergence, typically within seven iterations. To manage semantic overlap, the hyperparameter *sensitivity* $\in [0,1]$ serves as an inclusion radius. Our recommended range of $0.1-0.3$ yields high intent specificity, whereas higher values provide broader coverage by clustering datasets into overlapping intents.

Complexity The Controller has a time complexity of $O\left(\frac{|D|\cdot|UI|\cdot dim}{T}\right)$, and a space complexity of $O((|D|+|UI|)\times dim)$, where dim is the dimensionality of system-centric data behaviors (in our implementation $dim=10$, for the ten behaviors identified in Section II-A), and T is the number of Dask workers assuming ideal parallel execution among T.

B. Intent Mapper

Algorithm 1: Intent Mapper Algorithm

Input : $\mathcal{J}$, $\mathcal{S}$, D, $\mathcal{P}$, UI, $B(UI)$, $\mathcal{W}$
Output : $\hat{P}^*(D), \hat{V}^*(\hat{P}^*(D))$

foreach $ui_k \in UI$ **do**
  $P(ui_k) \leftarrow \emptyset$
  $\mathcal{M} : \{m(ui_k)_z : z \in [0,1] \mid B(ui_k)\}$
  foreach $p_i \in \mathcal{P}$ **do**
    if $B(ui_k)$ *impacts* p_i **then**
      foreach $v \in V(p_i)$ **do**
        compute $\text{score}_{p_i,v}(ui_k) \mid Eq.\ (3)$
      $v^*(p_i) \leftarrow \arg\max_{v \in V(p_i)} \text{score}_{p_i,v}(ui_k) \mid Eq.\ (4)$
      $P(ui_k) \leftarrow P(ui_k) \cup \{(p_i, v^*(p_i))\}$
$\mathcal{G} \leftarrow \text{GROUPBYSIGNATURE}(D, UI)$
foreach $Sig \in \mathcal{G}$ **do**
  compute $\text{priority}(Sig) \leftarrow \sum_{ui_k \in Sig} f(m(ui_k), \mathcal{W})$
$\mathcal{G}_{sorted} \leftarrow \text{SORTDESCENDING}(\mathcal{G}, \text{priority})$
foreach $(Sig, D_{Sig}) \in \mathcal{G}_{sorted}$ **do**
  if $Sig \in Cache$ **then**
    $P_{\text{prefs}}(Sig) \leftarrow Cache[Sig]$
  else
    $P(Sig) \leftarrow \emptyset$
    foreach $ui_n \in Sig$ **do**
      $P(Sig) \leftarrow P(Sig) \cup P(ui_n)$
    $P_{\text{prefs}}(Sig) \leftarrow \text{CONFLICT_RESOLVER}(P(Sig), \mathcal{W})$
    $Cache[Sig] \leftarrow P_{\text{prefs}}(Sig)$
  $(\hat{P}^*(D_{Sig}), \hat{V}^*(\hat{P}^*(D_{Sig}))) \leftarrow$
    $\text{CONSTRAINT_APPLICATOR}(D_{Sig}, P_{\text{prefs}}(Sig), \mathcal{S}, \mathcal{J})$
return $\hat{P}^*(D), \hat{V}^*(\hat{P}^*(D))$

The Intent Mapper translates high-level user intents into C/R configurations through a scalable, per-intent decision process. As workflows scale to thousands of nodes, their tasks and data behaviors can reach millions [2], [14]. Rather than selecting configurations at the intractable granularity of individual files,

TABLE II: Notations used in Design and Implementation.

Category	Notation	Definition	Description
Given	D	$\{d_i : i \in \mathbb{N}\}$	Set of all data in the workflow.
	$\mathcal{J}$	$\{j_i : i \in \mathbb{N}\}$	User job allocation (nodes, cores, wall time).
	$\mathcal{S}$	$\{s_i : i \in \mathbb{N}\}$	System specifications (free memory, storage capacities).
	$\mathcal{P}$	$\{p_i : i \in \mathbb{N}\}$	C/R parameters of underlying C/R library.
	$\mathcal{W}$	$\{w_{\text{perf}}, w_{\text{res}}, w_{\text{stor}}\}$	User-defined weights for performance, resilience, and storage trade-offs.
Inter-mediate results	$B(d_i)$	$\{\{b_j \in \prod_{k=1}^{10} F_k : j \in \mathbb{N}\}, d_i \in D\}$	System-centric behaviors per d_i, where F_k are domains of the core data characteristics.
	$P(d_i)$	$\{\{p_j : j \in \mathbb{N}\}, d_i \in D\}$	Candidate set of tunable parameters per d_i.
	$V(p_j)$	$\{\{v(p_j)_k : k \in \mathbb{N}\}, p_j \in P(d_i)\}$	Candidate parameter values per p_j.
	UI	$\{ui_k : k \in \mathbb{N}\}$	User-centric intents in the workflow.
Find	$\hat{P}^*(d_i)$	$\{\{\hat{p}_j^* : j \in \mathbb{N}\}, d_i \in D\}$	Adapted set of tunable parameters per d_i.
	$\hat{V}^*(\hat{p}_j^*)$	$\{\{\hat{v}^*(\hat{p}_j^*)_k : k \in \mathbb{N}\}, \hat{p}_j^* \in \hat{P}^*(d_i)\}$	Adapted parameter values per p_j^*.

the Mapper operates on *intents*. This reduces navigation from thousands of heterogeneous data behaviors to a small set of semantic categories, lowering mapping complexity.

For the remaining of our paper we define the notations in Table II. The Intent Mapper is detailed in Algorithm 1. The user must only provide the set of C/R library parameters ($\mathcal{P}$, i.e. in SCR, `SCR_CACHE_BASE`, `SCR_COPY_TYPE` etc) without specifying their values or combinations. The Intent Mapper first derives compact intent descriptors to guide mapping (Alg. 1, lines 1–3). Specifically, each intent is summarized using four normalized metrics. For a user intent ui_k:

- **Size:** Reflects *"what"* data size is associated with the intent to prune redundancy schemes exceeding tier limits.
- **Criticality:** Expresses *"how"* critical data are for restarts. According to literature [32], it is a continuous metric:

$$Crit(ui_k) = \frac{\text{\# I/O operations involving multiple processes}}{\text{Total \# I/O operations}} \in [0,1] \tag{1}$$

- **Access pattern:** Expresses *"how"* data interact with the system, where 0 represents fully sequential and 1 represents fully random access.
- **Ephemerality:** Represents *"when"* data are accessed or their lifespan relative to I/O bursts [33], [34]:

$$Eph(ui_k) = 1 - \frac{\text{Data lifespan}}{\text{Duration of access time intervals}} \in [0,1] \tag{2}$$

These descriptors provide a low-dimensional summary of intent behavior, sufficient to drive configuration decisions without fine-grained analysis of thousands of files. The Intent Mapper evaluates parameter–value pairs *for their appropriateness to the intent* (Alg. 1, lines 6–7) using a weighted scoring model that combines these descriptors. The weighted score for a parameter–value $p_i, v(p_i)_j$ for an intent ui_k is calculated as,

$$\text{score}_{p_i,v(p_i)_j}(ui_k) = \sum_{z=1}^{4} w_{p_i,v(p_i)_j,z} \cdot D_z(m(ui_k)), \tag{3}$$

where $\sum_{z=1}^{4} w_{p_i,v(p_i)_j,z} = 1$, and $D_z(\cdot)$ maps intent-derived metrics into a normalized utility space such that: $D_z(m(ui_k)) = m(ui_k)_z$ if larger values of a metric are preferred for parameter–value pair (e.g., ephemeral cache bases such as DRAM for ephemeral intents such as logging files), and $D_z(m(ui_k)) = 1 - m(ui_k)_z$ otherwise (e.g., shorter checkpoint intervals for more critical intents). Then the Intent Mapper selects the parameter value with the highest score (Alg. 1, lines 8–9),

$$v^*(p_i)_j = \arg \max_{v(p_i)_j \in V(p_i)} \text{score}_{p_i,v(p_i)_j}(ui_k) \tag{4}$$

TABLE III: Example weights for Eq. (3)

	Cache Base			Redundancy Schemes			
Metric	**DRAM**	**NVMe**	**Shared BB**	`SINGLE`	`PARTNER`	`XOR`	`R-S`
Size	0.12	0.30	0.40	0.50	0.30	0.40	0.20
Crit	0.20	0.20	0.25	0.10	0.50	0.30	0.60
Eph	0.38	0.10	0.10	0.40	0.20	0.30	0.20
Access	0.30	0.40	0.15	–	–	–	–

Weights are derived from a combination of empirical observations across representative workflows (Section III-A), documented characteristics of C/R mechanisms (e.g., latency of DRAM vs. NVMe, redundancy overheads) [27], [7], [8], and benchmark measurements (e.g., IOR [35], STREAM [36], uperf [37]). These weights are calibrated once using Montage workflow, and then held fixed across all workflows. This ensures that the scoring reflects both observed workload data and system behaviors. For instance, for the cache base, DRAM prioritizes *Ephemerality* (0.38) over *Criticality* (0.20), favoring short-lived data. Conversely, the `PARTNER` redundancy scheme emphasizes *Criticality* (0.40) and *Size* (0.30) over *Ephemerality* (0.20), suiting critical, small-to-moderate files (Table III).

Table IV illustrates representative mappings. Due to space constraints we only present the main configurations. For example, metadata (Table IV, row 3), which are characterized by high criticality and small size, are assigned fast storage with redundancy, whereas logging/scratch data, which are more ephemeral, are assigned `XOR` redundancy. Ties in parameter value selection are resolved by the Conflict Resolver.

1) Conflict Resolver: Due to the non-exclusive nature of intent classification, data may be associated with multiple user-centric intents when their behaviors span multiple semantic

TABLE IV: The Intent Mapper matches user I/O intents in workflows to representative C/R configurations

User I/O Intent	C/R Configurations	Parameter Values
Input dataset	Fetch to cache	1
	Cache base	NVMe
	Cache size	256
Partially Computed Data	Cache base	Shared mem.
	Cache size	48
	Redundancy scheme	R-S
Metadata	Cache base	DRAM
	Cache size	256
	Redundancy scheme	PARTNER\XOR\R-S
	Persistent storage	NVMe
Scratch / Logging Data	Checkpoint interval	3
	Cache base	DRAM
	Redundancy Scheme	XOR
Anchor files	Cache base	DRAM
	Cache size	48
	Redundancy scheme	SINGLE\PARTNER\XOR
	Persistent Storage	NVMe
Checkpoints	Checkpoint interval	1\2
	Cache base	NVMe
	Redundancy scheme	PARTNER\XOR
Final result	Cache base	NVMe
	Redundancy scheme	R-S

categories. For example, some FITS images in Montage can be classified as both logging and partially computed data. While this preserves workflow data behaviors, it can lead to conflicting configuration decisions. For example, overlapping intents may simultaneously suggest incompatible parameter values (e.g. cache base is both NVMe and DRAM).

To resolve such conflicts, iCR introduces the Conflict Resolver (Alg. 1, lines 10–22) that reconciles competing configuration candidates through a multi-objective decision process. The Resolver formulates conflict resolution as a *trade-off* across three goals: **performance**, **resilience**, and **storage**. These objectives are exposed to the user, who scores them based on their priorities ($\mathcal{W}$). We model performance using aggregate bandwidth, resilience using the total workflow makespan under a worst-case failure rate, and storage using the aggregate data footprint. For each parameter–value pair, the Resolver estimates its impact on these metrics using weighted scoring. Since these metrics are not directly comparable (e.g., bandwidth in MB/s vs. makespan in seconds), we normalize them to a common scale. To make a decision for the tied values, we do not use exact measurements. Instead, we rely on lightweight heuristics based on benchmark measurements (e.g., IOR [35]), and hierarchical models from literature on data caching and prefetching [12], [7], which have shown that exact measurements are not needed to achieve trade-offs. The Resolver evaluates candidate parameter–value configurations and selects the one with the highest weighted score. For makespan minimization, we borrow a literature defined failure rate [38], where failures follow a Weibull distribution. Let M denote the workflow makespan. The probability that at least one failure occurs during execution is given by:

$$P_{\text{fail}} = 1 - e^{-(M/\lambda)^k}, \tag{5}$$

where k is the shape parameter and λ is the scale parameter of the Weibull distribution (where $\lambda = MTBF$, when $k = 1$). To ensure scalability, the Conflict Resolver employs a **signature-based memoization** strategy. For each dataset, overlapping intents are encoded into a unique signature, and datasets are grouped by it, to form equivalence classes of datasets that share identical intents (e.g., metadata and anchor file) (Alg. 1, line 10). For each group, a priority score is computed as an aggregation of intent-derived metrics weighted by user preferences (Alg. 1, lines 11–12), and groups are processed in descending order of priority (Alg. 1, line 13). The dataset signature is used to cache previously computed decisions, avoiding redundant evaluations (Alg. 1, lines 14–16). In the case of a cache miss (Alg. 1, lines 17–20), the Resolver merges candidate parameters from all intents in the signature, evaluates them using the user-defined weights, and stores the result in the signature cache (Alg. 1, lines 21–22).

2) Constraint Applicator: The Constraint Applicator translates the selected configurations into physically feasible assignments under system resource constraints. Given that storage tiers have finite capacity, not all preferred configurations can be applied simultaneously. To maintain scalability, the Applicator employs a state-aware greedy assignment strategy. As mentioned earlier, datasets are previously processed in batches grouped by their intent signature (*Sig*), and sorted based on priority. For each batch (Alg. 1, line 14), the module computes the aggregate storage demand and compares it against the currently available capacity, derived from user-provided job limits ($\mathcal{J}$) and system specifications ($\mathcal{S}$). If sufficient capacity exists in the preferred storage tier (e.g., NVMe), the batch is assigned directly and the allocated space is deducted from the system state. Otherwise, the Applicator triggers a deterministic fallback by selecting the next-best configuration candidate (element of $P_{\text{prefs}}(Sig)$) provided by the Conflict Resolver.

Complexity The worst-case complexity of the Intent Mapper module yields an $O((|UI| \cdot |\mathcal{P}| \cdot |V(\mathcal{P})|) + (|D| \cdot |UI|) + (2^{|UI|} \cdot (|UI| + |\mathcal{P}| \cdot |V(\mathcal{P})|)) + |D|)$, where $O(|UI| \cdot |\mathcal{P}| \cdot |V(\mathcal{P})|)$ corresponds to scoring phase (Alg. 1, lines 1–9), $O(|D| \cdot |UI|)$ corresponds to dataset batching phase (Alg. 1, line 10), $O(2^{|UI|} \cdot |UI|)$ for computing signature priorities (Alg. 1, lines 11–12), $O(2^{|UI|} \log 2^{|UI|})$, which simplifies to $O(2^{|UI|} \cdot |UI|)$ is spent in sorting (given that in worst-case we have $|G| = 2^{|UI|}$, Alg. 1, line 13), $O(2^{|UI|})$ is spent looping through all signatures (and in worst-case we have $|G_{sorted}| = 2^{|UI|}$, Alg. 1, line 14), $O(|UI| + |\mathcal{P}| \cdot |V(\mathcal{P})|)$ is spent by the Conflict Resolver and $O(|D|)$ by the Constraint Applicator(given that in worst case we have $|D_{sig}| = |D|$).

C. Assumptions and Limitations

To reason about data dependencies in workflows, we add the functionality of node-to-node data transfers to the underlying C/R library for producer/consumer workflow data that have been cached locally (detailed in Section IV).

Therefore, we do not need to include data-dependencies as a constraint for data placement, because they do not need to be placed in shared storage for the workflow to continue execution. Second, we treat each group of tasks within the workflow as a separate entity of tightly coupled processes that produce concurrent checkpoints to reach a steady state. Consequently, iCR initiates a separate, independently configured instance of the C/R library for each task group. For example, this is possible in SCR by annotating the task's source code with `SCR_Config("SCR_CONF_FILE=%s", "/path/to/scr_config.conf");` [27].

One limitation of our work is that it relies on the collection of traces. These traces, however, are a one-time cost, and is outweighed by the long-term storage savings across repeated workflow executions from scientists (Section V). These traces remain relevant across different run configurations (e.g. number of nodes, processes), as they capture the invariant I/O intent logic of the workflow. In addition, our mapping algorithm is greedy as a deliberate design choice, sacrificing optimality in favor of computational efficiency. To bound the quality of our mapping we test iCR against classic optimization algorithms in Section V.

Another limitation is the need for storage and inter-connect bandwidth profiling for every new cluster. To do so, we run a suite of benchmark experiments (briefly mentioned in Section III-B) using the methodology in [39]. This includes 600 test combinations, using 4-128 nodes, four I/O natures (Write-Only, Read-Only etc), sequential and random access patterns, single process, intra and inter-node placement strategies, different transfer sizes and storage targets (i.e. NVMe). This is a costly process, but it is a one-time cost per HPC system which can be used in I/O optimizations beyond iCR.

IV. Implementation Details

We integrate iCR with DFTracer [24], which traces I/O events as compressed indexed JSON-line entities. The Intent Collector utilizes Python's Dask framework [40] for distributed random-access parsing, partitioning the trace into 1 MB byte windows assigned as independent tasks. Each partition is lazily loaded into a distributed Dask DataFrame across workers (threads) backed by PyArrow [41] columnar schemas. This parallel approach across workers ensures only a small portion of the trace is resident in memory, scaling to large traces while reducing the in-memory footprint compared to standard Pandas objects.

Intent extraction is implemented via the *parallel query engine* as a series of analytical Dask queries. These queries are lazily evaluated, transferring only the data required for computing specific data behaviors, (e.g., process counts and offsets) from disk to memory. System-centric behaviors are saved into thread-local dictionary, one per data item. We implement the final K-means in C++ with Python bindings. The Collector employs a gather-scatter operation to reduce the behavior dictionaries unordered map (converted from PyObject), where workers iteratively scatter datasets and synchronize to compute data-to-centroid distances. All workers update a shared in-memory map storing datasets and their associated user-centric intents.

The Intent Mapper, Conflict Resolver, and Constraint Applicator are Python modules using greedy algorithms. The Mapper encodes each intent as a 4-dimensional normalized vector (*Size, Crit, Access*, *Eph*) for constant-time access. C/R configurations are loaded from CSV files into nested dictionaries, and the Mapper computes a weighted inner product between intent and parameter weight vectors. For memoization, the Conflict Resolver utilizes a Python open-addressing hash table with SipHash-2-4, enabling average O(1) lookup for resolved signatures. The Constraint Applicator uses `psutil` and `os.cpu_count()` to dynamically cap configurations and prevent resource over-subscription. It employs a stateful first-fit strategy, tracking available capacity in a mutable dictionary.

To respect data dependencies, we integrate the C/R library with DYAD [42] for asynchronous node-to-node transfers. We extend the C/R library's restart API with a `CONSUMER=1` flag to trigger these transfers. Since DYAD is part of the Flux scheduler [43], it internally retains node location information for both producer and consumer tasks.

V. Performance Evaluation

A. Methodology

1) Hardware: All experiments were conducted on Corona cluster. The system comprises of 121 compute nodes, each equipped with an AMD Rome CPU featuring 48 cores and eight AMD MI50 GPUs. All nodes are interconnected via an InfiniBand high-speed fabric. Each node features 256GB of CPU memory with a 153GiB/s bandwidth, 32GB of memory per GPU, 1.5TB of local NVMe, and connects via an InfiniBand HDR network. Lustre is interconnected with EDR InfiniBand SAN using 100Gb OmniPath, it has 16 Metadata Servers and 36 Object Storage Servers and a capacity of 20TB. Each workflow runs on 64 nodes and 48 CPU cores or 8 GPUs per node to fully utilize the node.

2) Software: Our Python 3.9 environment includes `scr` 3.1.0, `pydftracer` 1.0.14, `torch` 2.2.1, `h5py` 3.10.0, `numpy` 1.26.0, `PuLP` 3.3.0 and `PyGAD` 3.6.0.

3) Baselines: As baselines, we use Lustre and SCR with default configurations. We choose SCR because its tuning knobs form a superset of VeloC's knobs for cache and persistent storage etc and ADIOS2's knobs for max buffer size etc, but iCR works similarly and is integrated with all three C/R libraries. Default SCR uses `XOR` redundancy (failure set size 8), checkpoint interval 1, and asynchronous PTHREAD flushes to Lustre. In this default configuration, we set the cache base to NVMe to avoid assumptions about dataset size. In the rest of our experiments, `R-S` is configured to tolerate two failures per set. To demonstrate and bound the quality of our mapping, we compare against MILP- and Genetic Algorithm (GA)-based SCR tunings as these types of optimizers have previously been used for workflow optimizations [12]. Our MILP global optimization implemented with `PuLP` solver, yields a complexity of $O(2^{|D|\cdot|B(D)|\cdot|P(D)|\cdot|V(P)|})$, formulated as a trade-off multi-objective function described in Section III-B1 using $\mathcal{J},\mathcal{S}$ as

constraints. That is because the algorithm has to traverse all possible data behaviors, parameters and their possible values and apply all constraints. The GA with `PyGA` has complexity $O(Gen \cdot Pop \cdot |D| \cdot |B(D)| \cdot |P(D)| \cdot |V(P)|)$, where $Gen = 50$ and $Pop = 20$ in our implementation. Prior works Mimir and H5Intent are not designed for C/R libraries but for POSIX and HDF5 APIs, and are therefore not used as baselines in our workflow experiments. We use Mimir as a baseline for testing iCR's scalability.

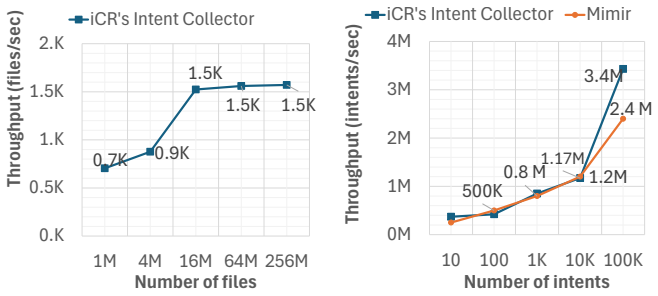
4) Metrics: For iCR scalability tests, we define four throughput types: (i) *files/s* to assess the Intent Collector's ability to classify data to our identified user intents (Table I), (ii) *intents/s* to evaluate the Intent Collector's intent storing time and (iii) *conflicts/s* to test the Conflict Resolver's performance under many conflicts.

5) Workflows: Modern workflows integrate scientific simulation tasks with AI-driven tasks. Therefore, to evaluate iCR we pick three different types of applications: Montage which is heterogeneous workflow with many intents, Megatron-DeepSpeed, to represent an AI tasks in workflows, and iPIC-3D for representing the simulation tasks. All runs are repeated five times as an aggregation across all tasks and the standard deviation is reported in the tables and figures.

B. Scalability

1) Intent Collector: To assess the Intent Collector's ability to classify datasets to known intents, we create a Wemul [44] synthetic workflow with 1 M–256 M files, where the user-centric intents are those in Table I. For these data, we generate random system-centric behaviors using a seed. We recommend using the Collector on at least one fully utilized compute node, to better utilize its *parallel query engine*. For this purpose, we run a Dask cluster of 96 workers (two nodes), and perform strong scaling (collected traces **5-50 GB** compressed). We do not perform weak-scaling tests, as this would require more processes and nodes to be used entirely for intent collection, which is not desirable. In Fig. 3(a), the y-axis shows the throughput in *files/s*, while the x-axis shows the number of files. We observe that the throughput increases until its saturation point at 1.5 K *files/sec*. Until 16 M files, the decompressed traces fit entirely in memory and are processed in parallel, achieving near-linear scaling. However, as the workload exceeds available memory, the collector reaches its computation capacity, where all workers are fully utilized and the system becomes bound by its complexity (Section III) and the overhead of managing a massive task graph in Dask.

To investigate the Intent Collector's ability to store new user I/O intents, we use a Wemul workflow of 16 M files. The collected compressed traces are **15 GB**. Our analysis runs on one fully utilized node. We run the Wemul workflow on 16 nodes and instrument it using Mimir's API to collect intents, as it is an online technique, and use it as our baseline. In Fig. 3(b), the y-axis shows throughput measured in *intents/s*, while the x-axis shows the number of intents. There is an almost linear increase in the throughput, with an average time of 0.0002 seconds per intent. That is because both tools use



(a) Classifying files to intents. (b) Storing new intents.

Fig. 3: iCR's Intent Collector can classify efficiently up to 16 M files and is comparable to Mimir.

TABLE V: Time (seconds) over number of intents.

Intents	10	100	1K
iCR Internal Components			
Intent Mapper	0.559 ± 0.12	6.344 ± 0.015	70.31 ± 0.08
Conflict Resolver	0.43 ± 1.4	4.985 ± 2.1	18.7 ± 3
Constraint Applicator	0.033 ± 0.004	0.132 ± 0.010	2.12 ± 0.12
External Baseline			
Mimir's Resolver	0.5 ± 1.2	5.01 ± 0.10	43.1 ± 0.006

unordered map to store and search intents efficiently, with an average case of $O(1)$ time complexity.

2) Intent Mapper: The Intent Mapper's scalability depends on the number of intents, the number of different candidate parameters and their values and the number of data files (Section III-B). As future workflows are projected to have 10-1 K intents [14], we run a Wemul synthetic workflow and scale the intents across 16 M files. Table V presents the results, with the Mapper including the entire time of scoring, mapping and running the Resolver and Applicator components. As we can observe, conflict resolution increases with the number of intents. This is expected, as the more user-centric intents there are, the more overlap there is among them, so data get classified to more than one intent. Therefore, the Conflict Resolver has an average constant rate of 78 K *conflicts/s* as each dataset is classified to 2.1 intents on average, while Mimir's is 60 K *conflicts/s*, which demonstrates the efficiency of signature memoization and caching of iCR's Resolver. The rest of the time is taken by dataset grouping into batches, signature calculation and priority sorting. For future workflows we expect a 1-100 seconds for the entire mapping pipeline.

Key Take-aways: iCR's Intent Collector saturates at 1.5 K *files/s* for dataset classification. iCR's Conflict Resolver outperforms similar solutions by 2.4×.

C. Montage Heterogeneous Workflow

The Two Micron All-Sky Survey (2MASS) is a major astronomical dataset that consists of FITS images [22] often processed with Montage workflow. Montage reprojects and rectifies these images to a uniform pixel grid that produces high-quality mosaics. We run the MPI version of Montage workflow, which consists of ten task groups, with a 100 GB input. Montage produces 37 K files, classified into six user intents. Loading the compressed traces in memory takes **5**

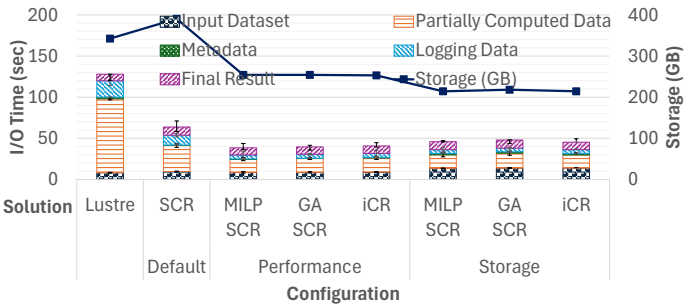


Fig. 4: Performance optimized iCR outperforms 1.6× and saves 1.82× space compared to default SCR for Montage.

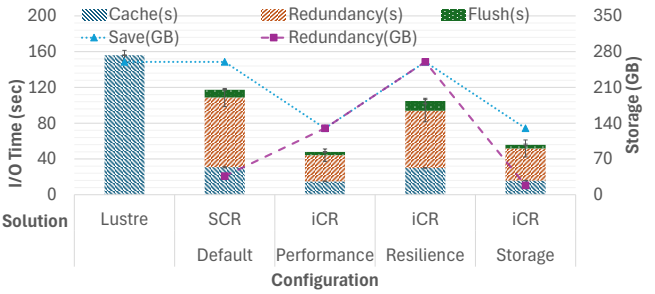


Fig. 5: iCR when optimized for performance reduces the checkpoint interval and chooses `PARTNER` redundancy while when optimized for storage it chooses `XOR` redundancy.

minutes. The entire pipeline of iCR runs in **6.2 minutes** with approximately 5.8 minutes spent on intent collection and the rest on mapping, adding only 0.08%, as the workflow makespan is 45 minutes, on 64 nodes. The overhead of collecting the DFTracer traces in the initial run is 10% and the storage footprint for storing the compressed traces is **348 MB**.

Fig. 4 y1-axis demonstrates the I/O time in seconds with bars and y2-axis storage in GBs with the line. We test two optimization modes for iCR: one for performance and one for storage. This tunes 16 SCR parameters to produce adapted configurations. We perform a workflow trace exploration using DFAnalyzer, to collect data behaviors, which runs for **32 minutes** and use MILP and GA tunings of SCR as baselines. MILP produces the right configurations in **2.7–3.3 minutes** and GA produces them in **10.6 minutes**. That is because the MILP and GA optimizations are global, while iCR uses greedy algorithms. We also compare against default SCR.

In Fig. 4, we observe that iCR when optimized for performance surpasses default SCR. That is because, iCR performance chooses the shared memory for the partially computed data, which is a dominant intent in this workflow, allowing for shorter I/O time compared to the NVMe used in default SCR. Logging/scratch data also significantly affect performance, where iCR performance reduces their I/O by 3× relative to default SCR by lowering the checkpoint interval to three, due to their high ephemerality (lifespan <15 minutes). Performance benefits also come from metadata, where iCR performance resolves the configuration tie by choosing `PARTNER` which is computed faster compared `XOR`, used by default SCR. Overall, iCR performance achieves a 1.6× I/O time reduction compared to default SCR, and a 3× compared to Lustre across all user intents.

When iCR is optimized for storage, it preserves storage by choosing a checkpoint interval of 3 for the logging/scratch data and preferring to classify FITS files in this intent instead of partially computed data, when conflicts arise. This way iCR storage saves 1.82× of storage across all intents compared to default SCR. However, slightly lower performance is encountered by iCR storage, as for the metadata files an `R-S` redundancy scheme is chosen, against `PARTNER` redundancy, to resolve the configuration tie. This creates a trade-off between performance and storage for the two optimization modes in iCR, and demonstrates the adaptability of iCR based on the user preferences. Specifically, iCR performance achieves 1.17× higher performance than iCR storage, while sacrificing 1.11× of storage.

Finally, both iCR modes remain within 5% of I/O time and storage footprint, compared to MILP and GA SCR tunings, demonstrating the accuracy of the dataset classification and mapping in iCR.

Key Take-away: iCR can reduce I/O time up to 1.6× and storage utilization by 1.82× compared to static configurations for Montage, while remaining within 5% of error compared to global optimizations which are 7× more expensive.

D. Megatron-DeepSpeed AI Framework

Megatron [31] is a large-scale transformer model which was integrated with DeepSpeed [45] to enable model-parallel, multi-node training of models like GPT and BERT. For our evaluation, we use a DLIO benchmark [46] version of Megatron-DeepSpeed 0.9.5 to emulate the I/O from training a 13-billion-parameter GPT model. The model is trained for one epoch of 1,000 steps with checkpointing every 200 steps. Checkpointing saves the optimizer state, layer parameters, and hyper-parameters per process, averaging 110 MB per transfer and ≈130 GB per checkpoint using mixed precision and Adam optimizer. Since 78% of the workload's I/O time is spent in checkpointing, we focus on this intent. iCR created the C/R configurations in **1.45 minutes**. The compressed trace size is **48 MB** and they take **2.3 minutes** to load in memory. DFTracer has a 5% overhead in the makespan of the workflow.

In this test, we analyze the source of performance and storage differences across the three different iCR modes when there are parameter value ties. Fig. 5 y1-axis presents the I/O time breakdown with bars and y2-axis presents the storage footprint with the lines. From the figure, we observe that in both default SCR and iCR resilience, redundancy computation dominates the I/O time, which reduces the effective flush time. This is expected, as the `XOR` redundancy scheme which is selected is computationally demanding and the main process has to block. iCR performance instead uses a checkpoint interval of 2 with the `PARTNER` scheme, reducing total I/O time by ≈2.6× due to lower checkpoint frequency and cheaper data redundancy. iCR storage follows a similar trend to iCR performance but demonstrates increased redundancy because it applies `XOR`.

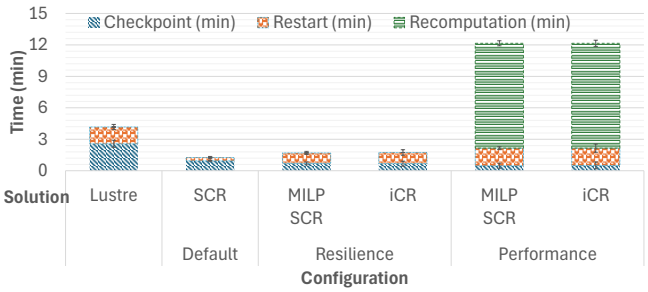


Fig. 6: iCR resilience does not lead to lost data under a worst-case failure rate for a particle-in-cell simulation.

The storage results are expected, as the main savings in storage footprint for iCR storage and iCR performance stem from the increased checkpoint interval. Across all 64 nodes with redundancy sets of eight processes, iCR requires 18.2 GB of cache space for redundancy data when optimized for storage, 260 GB when optimized for resilience, and 130 GB when optimized for performance. Based on this we observe a clear trade-off for the three modes of iCR.

Key Take-away: iCR can adapt to user's priorities in performance, resilience and storage to decrease I/O time by 2.6× and storage footprint by 2× for specific I/O intents.

E. Makespan of iPIC-3D Scientific Simulation

TABLE VI: iCR resilience maintains makespan.

Total Makespan (minutes)				
SCR Default	**iCR Resilience**	**MILP SCR Resilience**	**iCR Performance**	**MILP SCR Performance**
74.2 ± 0.4	73.4 ± 0.3	72.1 ± 0.2	83.3 ± 0.7	83.2 ± 0.6

iPIC-3D [30] is a three-dimensional, massively parallel particle-in-cell code designed for large-scale plasma physics simulations. We use the data-analytics version of iPIC-3D, which outputs visualization data while running the simulation, for every 3D subdomain. We use the Magnetic Reconnection data and run the solver for 800 cycles on 64 nodes. This results in a makespan of 70 minutes. We evaluate the workflow's makespan under the worst-case failure rate where a task fails every 18 minutes [4]. As described in section III-B1, according to Eq. (5) the probability that a single task fails system-wide, within time $M = 70$ minutes and $MTBF = 18$ minutes is 97.95%, when $k = 1$, which is standard when weighted score is reported [38]. Also, according to literature [7], nodes which have failed once, have the tendency to fail again in the near future. For this purpose, we fail one node every 200 cycles. We compare iCR's resilience and performance mode against MILP resilience and performance tunings of SCR, the default SCR and Lustre. iCR produces the configurations in **0.3 minutes** while the MILP solver in **2.6 minutes**. We omit GA results, as we have found that they underperform the MILP ones. The workflow checkpoints every 25 cycles. DFTracer traces occupy **13 MB** and its overhead in the makespan is 7% in the initial workflow run. The compressed traces take **0.9 minutes** to load in memory.

The workflows produces 2 K files. iCR collects two main user intents: the anchor files and the metadata. If data cached in DRAM is lost, the simulation must reconstruct it from neighboring nodes using redundancy data or fall back to an older checkpoint on the PFS, potentially increasing makespan due to re-computation when checkpoint intervals are sparse. This experiment evaluates how a worst-case failure scenario affects the data integrity and workflow makespan, when using different optimization modes of iCR.

Fig. 6 demonstrates the time in y-axis. Caching anchor files and metadata in DRAM and applying `R-S` redundancy to both intents results in better checkpointing performance for iCR resilience compared to default SCR. Due to the failures, restart time dominates iCR resilience. That is because, iCR chooses to cache these intents in the DRAM, which is volatile and apply `R-S` redundancy. Therefore, reconstructing the `R-S` dataset requires transferring and recomputing blocks across the set processes. In contrast, default SCR reads the dataset directly from the NVMe, which is not lost during failures. This reduces the restart time significantly, but some extra cost is paid during checkpointing. Finally, iCR performance applies no redundancy to the anchor data intent resulting in 2× lower checkpointing time. However, unflushed files are lost upon failure. Recovering requires reading the previous checkpoints from Lustre, adding ≈10 minutes of re-computation and resulting in the loss of roughly 100 computation cycles.

Table VI presents the impact of the worst-case failure rate in the makespan. iCR resilience is still able to provide data protection and maintain the same makespan with default SCR and performs similarly to MILP SCR for resilience.

Key Take-away: iCR, when optimized for resilience, does not lead to makespan increase, in a worst-case failure scenarios.

VI. Discussion

A. Integration with other C/R Libraries

While our evaluation focuses on SCR, iCR is designed to be backend-agnostic and we have integrated it with VeloC and ADIOS as well. VeloC exposes configurations such as cache base, flush buffer sizes and thread counts, while ADIOS exposes data aggregation, buffer allocation, and flushing intervals. The three backends exhibit different sensitivities to intents, but all benefit from iCR. For example, on a 64-node iPIC-3D run, iCR performance reduces per-checkpoint I/O time from 1.4 seconds to 0.9 seconds on VeloC and from 1.5 seconds to 0.8 seconds on ADIOS, comparable to the gains observed on SCR (Section V-E).

B. Cluster Assignment Robustness

A natural question is whether the extracted intents are stable across scale, or an artifact of a particular workflow. To examine this, we measure the Adjusted Rand Index (ARI) between intent assignments across scales, as it compares clusterings independently of labeling. For example, we find that on the Wemul workflow (1 M–256 M files, Section V-B), intent assignments remain highly consistent (average ARI 0.85–0.75).

C. Mapper Weight Sensitivity

To verify that our results do not depend on precise weight tuning, we perturb the Mapper weights using (i) a global Monte Carlo perturbation with multiplicative noise and re-normalization, and (ii) a local one-at-a-time perturbation of each weight component. For SCR, across 91 mapper decisions, the selected configurations exhibit a mean stability of 0.991 and a median of 1.0 (the fraction of decisions unchanged under perturbation), with multi-seed runs showing the same pattern. The mapping is therefore robust to moderate weight variation.

D. Intent Coverage Across Workflows

To demonstrate that the seven representative intents (Table I) generalize beyond the three evaluated workflows, we report the intents observed in the remaining three. ICECaps exhibits all seven intents (its JSON, NPY and YAML files are input dataset, NPZ, NPY, and HDF5 files are partially computed data, HDF5/DB are metadata, logs and JSON files are logging data, scalability study are anchor files, HYDRA files are checkpoints and NPY files are final result files); 1000-Genomes exhibits four (VCF and YAML files are input dataset, per-task TAR chromosome files are partially computed data, shifting annotations in `.bai` and `.fai` files are metadata, POP TAR are final results); and MuMMI exhibits two (its macro and micro scale are partially computed data and its ML selection files are anchor files).

VII. Related Work

C/R in scientific workflows has been explored in the literature as optimization problems over DAGs to minimize expected makespan or cost. Ding et. al [11] proposed a greedy heuristic with a cost model for elastic scheduling on cloud resources. Benoit et. al [47] and Han et. al [47] used dynamic programming to compute optimal checkpoint intervals and task replication against fail-stop errors. More general mechanisms [48], [49] use greedy or rule-based strategies, such as checkpointing at exit tasks, to reduce optimization costs. While fundamental, these works often rely on theoretical assumptions regarding failure rates and slow-down factors. In light of the benefits of approximate techniques, we have also implemented iCR to use a greedy mapper to reduce the complexity of exact optimizations. Because we are not able to compare with the aforementioned related works as their code is not open-sourced, we compare against other exact solvers (Section V) to bound the iCR's efficacy.

Recent efforts in intent-driven I/O optimization, Mimir [14] and H5Intent [15], define and use I/O intents to guide I/O optimizations in tuning POSIX and HDF5 APIs to cache, prefetch and compress data. Our work complements these systems by utilizing their intent definitions, but also extends them to capture *"when"*, as described in Section I. This need of extending I/O intent principles is critical for the C/R domain, as it can guide redundancy scheme, storage selection for caching and, flushing decisions, from a resilience perspective. Unlike the previous works on I/O intents, our work is purely offline. That is an intentional design choice, in order for iCR to avoid reliance on manual annotations of source codes for intent extraction, that the previous works were dependent on.

VIII. Conclusions

In this work, we have addressed the need for adaptable C/R strategy for the heterogeneity in workflows, data and the storage stack. We have presented iCR, an end-to-end toolkit for enabling intent-driven adaptable C/R configurations for workflows. Through our work, we have found that: (1) user intents can transcend across workflows (i.e. we have identified seven user intents across six representative workflows) (2) a trace-driven pipeline that extracts user I/O intents can systematize the process of granular fine-tuning of C/R libraries to eliminate user intervention and (3) greedily mapping user I/O intents can reduce the complexity of exploring the C/R configuration space by 7× while remaining within 5% error of exact optimizations. Our results demonstrate that iCR achieves up to 2.6× I/O time reduction, and 2× storage reduction compared to static C/R configurations. For future work, we are planning on extending iCR to work as an online framework by developing heuristics that allow C/R configurations to be adapted during run-time.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and Project No. 23-ERD-053. LLNL-CONF-2014335-DRAFT. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program. This work is also supported in part by the National Science Foundation awards 1763547 and 2403089. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] S. Kuo, M. Winslett, Y. Cho, J. Lee, and Y. Chen, "Efficient input and output for scientific simulations," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pp. 33–44, 1999.

[2] H. Devarajan and K. Mohror, "Extracting and characterizing i/o behavior of hpc workloads," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 243–255, IEEE, 2022.

[3] J.-C. Golaz, L. P. Van Roekel, X. Zheng, A. F. Roberts, J. D. Wolfe, W. Lin, A. M. Bradley, Q. Tang, M. E. Maltrud, R. M. Forsyth, *et al.*, "The doe e3sm model version 2: Overview of the physical model and initial model evaluation," *Journal of Advances in Modeling Earth Systems*, vol. 14, no. 12, p. e2022MS003156, 2022.

[4] A. Das, F. Mueller, and B. Rountree, "Systemic assessment of node failures in hpc production platforms," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 267–276, IEEE, 2021.

[5] S. Cui, A. Patke, H. Nguyen, A. Ranjan, Z. Chen, P. Cao, B. Bode, G. Bauer, C. Di Martino, S. Jha, *et al.*, "Characterizing gpu resilience and impact on ai/hpc systems," *arXiv preprint arXiv:2503.11901*, 2025.

[6] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, p. 494, IOP Publishing, 2006.

[7] 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*, pp. 1–11, IEEE, 2010.

[8] B. Nicolae, A. Moody, G. Kosinovsky, K. Mohror, and F. Cappello, "Veloc: Very low overhead checkpointing in the age of exascale," *arXiv preprint arXiv:2103.02131*, 2021.

[9] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[10] H. Devarajan, A. Moody, D. Dai, C. Stanavige, E. Gonsiorowski, M. McFadden, O. Faaland, G. Kosinovsky, and K. Mohror, "The impact of asynchronous i/o in checkpoint-restart workloads," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 397–405, IEEE, 2024.

[11] Y. Ding, G. Yao, and K. Hao, "Fault-tolerant elastic scheduling algorithm for workflow in cloud systems," *Information Sciences*, vol. 393, pp. 47–65, 2017.

[12] F. Chowdhury, F. Di Natale, A. Moody, K. Mohror, and W. Yu, "Dfman: A graph-based optimization of dataflow scheduling on high-performance computing systems," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 368–378, IEEE, 2022.

[13] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien, "Checkpointing workflows for fail-stop errors," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1105–1120, 2018.

[14] H. Devarajan and K. Mohror, "Mimir: Extending i/o interfaces to express user intent for complex workloads in hpc," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 178–188, IEEE, 2023.

[15] H. Devarajan, G. Heber, and K. Mohror, "H5intent: Autotuning hdf5 with user intent," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[16] H. I. Ingólfsson, H. Bhatia, F. Aydin, T. Oppelstrup, C. A. López, L. G. Stanton, T. S. Carpenter, S. Wong, F. Di Natale, X. Zhang, *et al.*, "Machine learning-driven multiscale modeling: bridging the scales with a next-generation simulation infrastructure," *Journal of Chemical Theory and Computation*, vol. 19, no. 9, pp. 2658–2675, 2023.

[17] X. Zheng-Bradley, I. Streeter, S. Fairley, D. Richardson, L. Clarke, P. Flicek, and . G. P. Consortium, "Alignment of 1000 genomes project reads to reference assembly grch38," *Gigascience*, vol. 6, no. 7, p. gix038, 2017.

[18] I. Yildirim, H. Devarajan, A. Kougkas, X.-H. Sun, and K. Mohror, "Wisio: Automated i/o bottleneck detection with multi-perspective views for hpc workflows," in *Proceedings of the 39th ACM International Conference on Supercomputing*, pp. 749–763, 2025.

[19] Cray Inc., "Cray datawarp applications i/o accelerator." https://www.cray.com, 2014. White paper / product documentation.

[20] IBM Corporation, "Ibm burst buffer api documentation." https://www.ibm.com, 2017. Documentation for BBAPI used on Summit/Sierra systems.

[21] 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, pp. 221–232, SPIE, 2004.

[22] G. B. Berriman, J. C. Good, E. Deelman, R. Tanaka, and K. Vahi, "Astronomical image processing at scale with pegasus and montage," *arXiv preprint arXiv:2111.11624*, 2021.

[23] Lawrence Livermore National Laboratory, "Corona — hpc @ llnl compute platform," 2025.

[24] H. Devarajan, L. Pottier, K. Velusamy, H. Zheng, I. Yildirim, O. Kogiou, W. Yu, A. Kougkas, X.-H. Sun, J. S. Yeom, *et al.*, "Dftracer: An analysis-friendly data flow tracer for ai-driven workflows," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–24, IEEE, 2024.

[25] L. L. N. Laboratory, "DFAnalyzer (software)." https://github.com/LLNL/dfanalyzer, 2025. Accessed: 2025-11-13.

[26] S. Mitchell, R. Lougee-Heimer, and K. Lodewijks, "PuLP: A Linear Programming Modeler in Python," 2011. Accessed: 2026-04-09.

[27] Lawrence Livermore National Laboratory SCR Team, *SCR 3.1.0 User Overview*. Lawrence Livermore National Laboratory, 2022. Available online: https://scr.readthedocs.io/en/latest/users/overview.html (accessed November 12, 2025).

[28] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, pp. 56–61, 2010.

[29] J. L. Peterson, T. Bender, R. Blake, N.-Y. Chiang, M. G. Fernández-Godino, B. Garcia, A. Gillette, B. Gunnarson, C. Hansen, J. Hill, *et al.*, "Toward digital design at the exascale: An overview of project icecap," *Physics of Plasmas*, vol. 31, no. 6, 2024.

[30] S. Markidis, G. Lapenta, *et al.*, "Multi-scale simulations of plasma with ipic3d," *Mathematics and Computers in Simulation*, vol. 80, no. 7, pp. 1509–1519, 2010.

[31] NVIDIA, "Megatron-LM," 2025. GitHub repository; accessed 2025-12-03.

[32] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, *et al.*, "Data criticality estimation in software applications," in *International test conference*, pp. 802–810, 2003.

[33] R. Li, Q. Ma, J. Gong, Z. Zhou, and X. Chen, "Age of processing: Age-driven status sampling and processing offloading for edge-computing-enabled real-time iot applications," *IEEE Internet of Things Journal*, vol. 8, no. 19, pp. 14471–14484, 2021.

[34] O. Kogiou, H. Devarajan, C. Wang, W. Yu, and K. Mohror, "Understanding highly configurable storage for diverse workloads," in *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, pp. 96–103, IEEE, 2024.

[35] H. I. Project, "IOR and mdtest parallel i/o benchmark." https://github.com/hpc/ior, 2025. Accessed: 2025-11-13.

[36] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers." https://www.cs.virginia.edu/stream/, 1995. Accessed: 2026.

[37] IBM Corporation, "uperf: A network performance measurement tool." https://github.com/uperf/uperf, 2010. Accessed: 2026.

[38] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2009.

[39] H. Devarajan and K. Mohror, "Tailorfs: An adaptive file system to support dynamic i/o requirements of hpc workloads," in *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 81–92, IEEE, 2024.

[40] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, vol. 130, pp. 136–141, 2015.

[41] Apache Software Foundation, "Apache arrow: A cross-language development platform for in-memory data." https://arrow.apache.org/, 2022. Accessed: 2026-04-21.

[42] H. Devarajan, I. Lumsden, C. Wang, K. Georgouli, T. Scogland, J.-S. Yeom, and M. Taufer, "Dyad: Locality-aware data management for accelerating deep learning training," in *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 13–24, IEEE, 2024.

[43] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, M. Springmeyer, and G. Voskuilen, "Flux: A next-generation resource management framework for large-scale systems," in *Proceedings of the 2014 International Workshop on Runtime and Operating Systems for Supercomputers*, pp. 9–16, ACM, 2014.

[44] F. Chowdhury, Y. Zhu, F. Di Natale, A. Moody, E. Gonsiorowski, K. Mohror, and W. Yu, "Emulating i/o behavior in scientific workflows on high performance computing systems," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, pp. 34–39, IEEE, 2020.

[45] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 3505–3506, 2020.

[46] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath, "Dlio: A data-centric benchmark for scientific deep learning applications," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 81–91, IEEE, 2021.

[47] 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.

[48] A. R. Setlur, S. J. Nirmala, H. S. Singh, and S. Khoriya, "An efficient fault tolerant workflow scheduling approach using replication heuristics

and checkpointing in the cloud," *Journal of Parallel and Distributed Computing*, vol. 136, pp. 14–28, 2020.

[49] L. Han, V. Le Fèvre, L.-C. Canon, Y. Robert, and F. Vivien, "A generic approach to scheduling and checkpointing workflows," in *Proceedings of the 47th International Conference on Parallel Processing*, pp. 1–10, 2018.