# Kelvin: Approaching Absolute Zero-Copy Single-Machine Data Pipelines

Yifan Dai, Jacopo Tagliabue[†], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau,
and Tyler R. Caraza-Harter
*University of Wisconsin–Madison, [†]Bauplan Labs*

***Abstract.*** We present Kelvin, a DAG execution engine for single-machine data pipelines that approaches absolute zero-copy by enabling containerized stages to directly share physical memory through the in-memory file system. At the core of Kelvin is DeAnon, a lightweight Linux kernel module that converts anonymous memory pages into file-backed pages in place, allowing data produced by unmodified applications to become referable across processes without copying. Building on this capability, Kelvin extends Arrow-based communication to preserve sharing across pipeline transformations and incorporates runtime support to reuse deserialized inputs and manage shared memory across containers. Experiments show that Kelvin eliminates most intermediate copying, enabling pipeline stages to directly share physical memory; as a result, Kelvin improves pipeline throughput by up to 28× while reducing memory consumption and swapping.

## 1 Introduction

Data pipelines are a popular paradigm for data analysis and machine-learning workloads. These pipelines are frequently implemented as directed acyclic graphs (DAGs), where each node of a DAG describes a transformation to perform on the data [6, 30, 34, 41] and edges represent data passed between nodes. For data pipelines composed of many fine-grained nodes, efficient communication between nodes is critical for good performance [15, 28, 40]. Data passing may occur via network, disk, or memory [11, 18, 19, 24, 41].

Recent workload and hardware trends [20, 33] demonstrate the feasibility of deploying most data pipelines on a single machine, with memory as the most efficient medium for data passing. Cloud virtual machines with 32 TB of RAM are now available [27]; in contrast, the largest (p99.9) datasets for OLAP workloads occupy a mere 0.25 TB [26].

In single-machine deployments, intermediate data is passed through *in-memory file systems* such as Linux `tmpfs` [16]. Files in these systems are backed directly by physical memory and can be mapped into the address spaces of multiple processes, allowing data to be shared without copying. Ideally, pipeline stages could communicate by simply sharing references to the same underlying memory pages.

However, this ideal is difficult to achieve in practice due to a mismatch between how applications allocate memory and how operating systems enable sharing. User code and libraries typically allocate data in *anonymous memory* (e.g., via `malloc`), which is visible only within the allocating process. In contrast, sharing data across processes requires the data to reside in *file-backed memory*, such as pages belonging to an in-memory file system. As a result, data produced by one pipeline stage must typically be copied from anonymous memory into the in-memory file system before it can be accessed by downstream stages. Similar forms of duplication arise in other scenarios as well, such as when pipeline transformations reuse portions of their inputs or when multiple pipelines independently load the same input data. These copies increase both execution time and memory consumption, and complicate resource management when data is shared across containers.

Although zero-copy communication has long been a goal of systems research [13–15, 25, 31], existing techniques do not fully address the challenges of data sharing in DAG pipelines. In particular, while formats such as Apache Arrow [7, 38] enable readers to map shared data without copying, they do not eliminate the copies required to expose newly generated data to other processes. Similarly, existing kernel and user-space mechanisms for zero-copy communication do not address the broader problem of sharing intermediate data across containerized pipeline stages.

To approach absolute zero-copy single-machine data pipelines, we present *Kelvin*, a system that enables pipeline stages to share physical memory directly through the in-memory file system while remaining compatible with existing libraries and unmodified user code. At the core of Kelvin is *DeAnon*, a Linux kernel module that converts anonymous memory pages into file-backed pages. By transforming user-allocated memory (e.g., via `malloc`) into pages backed by an in-memory file system, DeAnon allows user-produced data to become referable by other processes without copying. Building on this capability, Kelvin extends Arrow-based communication to preserve sharing across pipeline transformations, detecting when outputs reuse existing buffers and avoiding redundant copies (DeDuce). Kelvin also incorporates runtime mechanisms to reuse previously deserialized inputs across pipelines (DeCache) and to track shared memory objects across containers, enabling accurate memory accounting as well as admission control and eviction when memory pressure arises (DeLimit).

By enabling pipeline stages to share memory directly rather than copying intermediate data, Kelvin reduces both the time overhead of data movement and the memory overhead of duplicated data. These reductions allow more pipeline stages to run concurrently and significantly improve overall throughput. We evaluate Kelvin using a combination of microbenchmarks, pipeline transformations, and multi-DAG workloads running on a single machine. Our experiments show that de-anonymization eliminates write-side copies when exposing intermediate results, while shared communication and caching substantially reduce duplication across transformations and pipelines. Overall, Kelvin

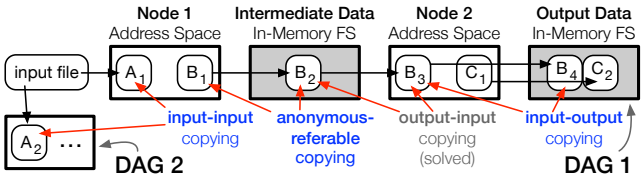



Figure 1: **Types of Copying.** Copies of the same in-memory data are labeled with the same letter with different subscripts.

eliminates most intermediate copying, improving pipeline throughput by 1.2–28× across a variety of workloads while reducing memory consumption and swapping.

This paper is organized as follows. We provide further motivation (§2) and background on the relevant components (§3), introduce Kelvin's design and implementation on Linux (§4), evaluate Kelvin's performance (§5), describe related work (§6), and conclude (§7).

## 2 Motivation: Copying in Data Pipelines

DAG-based data pipelines can now be deployed on a single machine with ample memory, creating opportunities for data sharing. Pipeline stages should be able to communicate by sharing references to the same physical memory rather than copying data. However, nodes in modern DAGs typically execute in separate containers [10, 22] to provide each node with its own address space and environment. This isolation means that nodes must communicate through OS mechanisms rather than direct memory access. Unfortunately, most popular data processing libraries are not designed to leverage these mechanisms; as a result, DAG nodes using these libraries incur superfluous memory copies despite co-location on the same machine.

Our goal is to build a pipeline execution platform that avoids copying memory whenever possible. Two realistic constraints shape this goal. First, to minimize developer effort, the system must support arbitrary user code and existing libraries in the Arrow ecosystem (§3.2) without modification. Second, each pipeline node must run in its own container to ensure isolation and accurate resource accounting.

To illustrate how these constraints typically lead to superfluous copies, Figure 1 shows a minimal example of two DAGs executing on a single machine. In DAG 1, node 1 (i.e., function 1) loads data from storage and constructs an in-memory representation of a table ($A_1$). The function computes over input $A_1$ to produce output $B_1$. The node then writes this data to the in-memory file system ($B_2$) so it can be accessed by downstream nodes. Node 2 loads the data into its address space ($B_3$), extends the input to produce output (e.g., original data with an added column, $C_1$), and writes it as $B_4 + C_2$ back to the in-memory file system. DAG 2 performs separate computation starting from the same input file.

Although all address spaces and the in-memory file system reside in the same physical memory, each unique piece of data ($A$, $B$, and $C$) has multiple physical copies. We categorize four forms of copying and duplication in the example. The first type, *output-input copying*, occurs when downstream nodes copy intermediate results into their address spaces. This copy can be avoided using in-memory file systems together with pointer-less formats such as Apache Arrow: downstream nodes can simply map upstream output files ($B_2$) into their address spaces ($B_3$). However, three additional sources of copying remain unresolved challenges in current systems.

**Challenge 1: Anonymous-referable copying.** User code and libraries typically allocate Arrow data in *anonymous memory* (e.g., via `malloc`). Anonymous pages are private to a process and cannot be directly referenced by other processes. In contrast, data shared across containers must reside in *file-backed memory*, such as pages belonging to an in-memory file system. Consequently, data generated in anonymous memory must first be copied into a referable location before downstream nodes can access it. In Figure 1, this copy occurs when $B_1$ is copied to $B_2$.

**Challenge 2: Input-output copying.** Many data transformations partially reuse input data in their outputs (e.g., filtering, sorting, projection, and appending columns). Arrow IPC, the standard protocol for sharing Arrow data between processes, serializes the entire output dataset inline when producing messages. As a result, even when large portions of the output are identical to the input, the protocol copies the data rather than reusing existing buffers. This duplication appears as the copy from $B_3$ to $B_4$ in Figure 1.

**Challenge 3: Input-input copying.** Multiple pipelines often share common input sources [26]. Each pipeline typically reads the same source data and independently converts it into its in-memory representation (e.g., Arrow). This process creates duplicated deserialized data in memory. In Figure 1, these duplicates correspond to $A_1$ and $A_2$.

These challenges reveal a fundamental difficulty in achieving zero-copy: data produced by unmodified applications resides in anonymous memory, while sharing across containers requires file-backed memory. Eliminating copying requires mechanisms to bridge this gap while preserving compatibility with existing libraries and execution models.

## 3 Background

We now examine existing kernel (§3.1) and user-space (§3.2) mechanisms for zero-copy communication and explain why they are insufficient for eliminating copying.

### 3.1 In-Memory File System Limitations

Operating systems provide mechanisms for zero-copy communication through shared memory and in-memory file systems. Linux, like most modern kernels, gives each process its own virtual address space implemented through page tables that map virtual pages (typically 4 KB) to physical memory.

Virtual pages are either *anonymous* or *file-backed*. Anonymous pages are private to a process and cannot be referenced by unrelated processes. File-backed pages, in contrast, correspond to offsets within a file and can be mapped by multi-

ple processes simultaneously. In-memory file systems such as `tmpfs` provide file-backed pages that reside entirely in physical memory, enabling processes to share data through standard file-mapping operations.

However, these mechanisms impose an important constraint: pages must be allocated as file-backed memory to be shared across processes without copying. If data is first created in anonymous memory—as is typical for user programs—then sharing the data through an in-memory file system requires copying it into a file-backed page (Challenge 1).

Sharing data across containers also complicates memory accounting. When multiple processes map the same in-memory file, the operating system must determine which container is responsible for the underlying physical memory. Linux addresses this problem through control groups (cgroups), which track memory usage and enforce limits for containers and micro-VMs. Memory allocated for in-memory file systems is typically charged to the cgroup that created the file, and memory pressure within a cgroup determines which pages are swapped or reclaimed. As a result, sharing memory across containers interacts closely with the kernel's accounting and eviction mechanisms.

## 3.2 Arrow Zero-Copy Limitations

Apache Arrow is a widely used in-memory format designed for analytics workloads. Arrow represents tabular data using pointerless columnar structures that can be mapped directly into process address spaces. This design allows readers to operate on shared data without copying it.

**Tabular Format:** Arrow represents tabular data as a collection of column buffers stored in contiguous memory. Fixed-length values (e.g., `float64`) are stored directly in arrays, while variable-length values (e.g., strings or lists) use separate buffers of values and offsets. This pointerless design allows Arrow data to be mapped into address spaces and accessed without copying. Arrow also supports optional *dictionary encoding*, where repeated values are stored once in a dictionary and referenced by integer codes in the column. Arrow buffers are immutable once written, enabling multiple processes to safely access the same data and allowing computations to operate directly on shared memory.

**Compatibility Requirements:** The Arrow ecosystem spans multiple languages and implementations, including PyArrow (C++), Polars (Rust), and DuckDB (C++). Applications frequently combine these libraries within the same pipeline. As a result, pipeline systems cannot assume control over memory allocation or require modifications to Arrow libraries or user code. Any solution for zero-copy communication must operate transparently on memory produced by existing libraries and runtimes.

**Communication:** Arrow also defines a communication protocol, Arrow IPC, for transferring data between processes. IPC messages include a schema describing the table layout, optional dictionary messages that encode repeated values, and record batches containing column data stored in the Arrow format. When Arrow IPC messages are stored in an in-memory file system, downstream processes can map them and construct tabular views directly. This approach eliminates output-input copying on the reader side.

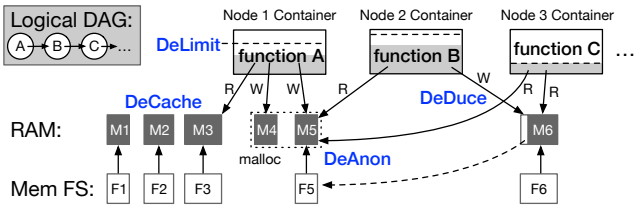



Figure 2: **Kelvin Architecture.** Both in-memory files and address spaces refer to unique physical memory. Address space references are read-only (R) or also writeable (W).

However, Arrow does not eliminate copying. When data originates in anonymous memory, it must be copied to an in-memory file before it can be shared across processes (Challenge 1). Furthermore, Arrow IPC serializes output data inline, preventing reuse of buffers across transformations or pipelines. Consequently, input-output and input-input duplication (Challenges 2 and 3) remain unresolved.

These limitations highlight the gap between existing zero-copy mechanisms and modern in-memory pipelines: while formats such as Arrow allow processes to read shared data without copying, systems still lack mechanisms to expose application-generated memory as shareable objects and to preserve sharing across pipeline transformations.

# 4 Kelvin: Design and Implementation

Kelvin is a data pipeline platform designed to execute DAG workloads on a single machine while minimizing data copying between containerized nodes. The key idea in Kelvin is to treat intermediate pipeline data as shared memory objects backed by an in-memory file system. Rather than copying data between processes, pipeline stages exchange references to the same underlying physical pages.

Kelvin's reference passing is transparent to users, who simply submit jobs describing DAGs to run. Each node specifies a set of library dependencies and a user function that accepts/returns Arrow data. Each node also specifies its parents, which are either other nodes or input files. A function may be executed after all its parent nodes have completed. Kelvin currently supports Python functions and packages, but could easily be extended to other languages, given Arrow's many implementations.

Figure 2 provides a broad overview of how Kelvin runs such a DAG. Each node executes in its own container. When a node begins, its input data is mapped directly from the in-memory file system into the container's address space using the Arrow format (e.g., $M3$ to Node 1). The node then executes arbitrary user code, which typically allocates Arrow data in anonymous memory using standard language runtimes, libraries, and allocators (e.g., $M4 + M5$).

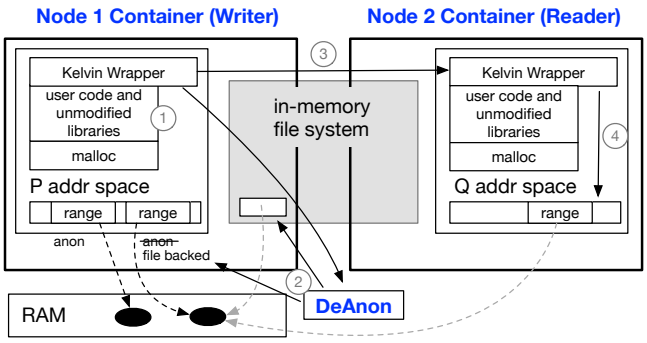



Figure 3: **DeAnon.** The Kelvin Wrapper runs inside a container and uses DeAnon to transparently pass intermediate data without a copy.

After the user code finishes, Kelvin exposes the output data (*M*5) to downstream nodes without copying. A kernel mechanism (DeAnon) converts the anonymous memory pages containing the output into file-backed pages in the in-memory file system (*F*5). As a result, the output buffers become referable by other processes. Kelvin then publishes references to these buffers so that downstream nodes (such as Node 2) can map the same physical memory directly into their address spaces.

Kelvin further preserves sharing across common pipeline transformations. When outputs reuse buffers from their inputs (e.g., Node 2's output is an extension of the input, *M*5), the communication layer (DeDuce) identifies these relationships and records references instead of copying data. In addition, Kelvin reuses previously deserialized input data across different pipelines when they originate from the same input sources (DeCache). Finally, because multiple containers may share the same physical data, Kelvin tracks shared memory objects and coordinates admission control and eviction decisions across containers (DeLimit).

These mechanisms allow Kelvin to approach zero-copy execution for in-memory data pipelines while remaining compatible with existing Arrow libraries and unmodified user code. The remainder of this section describes the mechanisms that enable this design: de-anonymizing application memory (§4.1), preserving sharing during communication (§4.2), reusing deserialized inputs across pipelines (§4.3), and managing shared memory across containers (§4.4).

## 4.1 DeAnon: De-Anonymization

The first challenge addressed by Kelvin is exposing data produced in anonymous memory so that it can be shared across containers without copying. DeAnon is a Linux kernel module that converts anonymous pages into file-backed pages. Instead of copying data into a tmpfs file, DeAnon reassigns the existing physical pages to that file, allowing downstream processes to access the same memory directly.

### 4.1.1 Design

DeAnon converts a range of anonymous pages into file-backed pages by modifying kernel metadata rather than copying data. Given a memory range, DeAnon allocates a new file and updates the process's virtual-memory metadata so that the specified pages are associated with the new file. The pages themselves remain in place; both the process address space and the in-memory file now reference the same physical memory. DeAnon returns the file identifier and offsets corresponding to the de-anonymized range so that other processes can map the data.

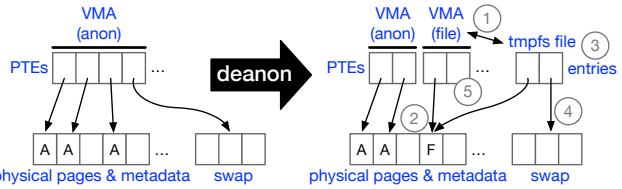



Figure 4: **DeAnon Kernel Module.** DeAnon's changes to virtual memory areas (VMAs), page-table entries (PTEs), and other metadata is shown.

Figure 3 illustrates how Kelvin uses DeAnon to pass data between containerized processes without copying. Each DAG node runs inside its own container, and all containers mount the same tmpfs instance. Execution begins in the Kelvin wrapper, which invokes the user-defined function for the node. (1) The function produces Arrow data allocated in anonymous memory using standard language runtimes and libraries. (2) After the function returns, the wrapper invokes DeAnon to expose the relevant buffers through the in-memory file system and (3) includes this information in the node's output. (4) A downstream process maps the corresponding in-memory files into its address space to access the same physical memory without copying.

### 4.1.2 Interfaces and Implementation

We implement DeAnon as a Linux kernel module consisting of roughly 1000 lines of C code. DeAnon exposes two kernel interfaces. The first, `new_file()`, allocates a new file in the in-memory file system that will receive de-anonymized memory. The second, `deanon(file_id, addr, len)`, converts the anonymous pages in the address range `[addr, addr+len)` into file-backed pages associated with the specified file by modifying kernel metadata.

Figure 4 illustrates how de-anonymization works. DeAnon first (1) changes the anonymous virtual memory area (VMA) of the target range to file-backed, splitting it if necessary, and then (2) walks the page table to link pages in the target range with the tmpfs file and (3) adds the pages to the file's page list.

When the working set exceeds available memory, Linux may swap pages to disk. In this case, page-table entries contain swap references rather than physical pages. When DeAnon encounters such entries, a naive implementation would first swap the pages back into memory before performing de-anonymization. Instead, Kelvin implements a *direct swap* optimization. As shown in Figure 4, (4) the swap entry is inserted directly into the tmpfs file metadata and (5)

the page-table entry is updated accordingly. This approach avoids unnecessary swap-in operations.

For our container implementation, we use SOCK [22] with minor modifications; each container mounts the same tmpfs instance (typically `/dev/shm`) so that de-anonymized files are visible across containers. We disable container reuse to simplify memory accounting.

#### 4.1.3 Correctness and Safety

Kelvin exposes memory across containers while preserving the semantics expected by existing applications and libraries. In particular, Kelvin must ensure that shared pages are not modified after publication to be safely shared to multiple downstream containers without synchronization. To achieve this, DeAnon operates only after user code has returned to the Kelvin wrapper, when the Arrow objects to be shared have been fully constructed. At this point, the buffers are no longer modified by application code, avoiding races with language runtimes. To prevent premature deallocation, the wrapper retains references to Arrow objects until container exits. Once exposed by DeAnon, shared Arrow buffers are treated as immutable and can only be mapped as read-only.

Kelvin exposes outputs and tracks their dependencies only after the producing node completes. If a container crashes before completion, Kelvin discards the untracked in-memory files that DeAnon may have generated and reruns the node.

In rare cases where a page cannot be safely shared, Kelvin falls back to copying. The main fallback case is an anonymous page with multiple mappings, typically due to forked shared-mapped pages or forked copy-on-write pages that are unmodified after the fork. Data-processing libraries we evaluate do not have this behavior: they produce outputs on private pages and do not fork after producing outputs.

DeAnon operates on page-granular mappings even though its interface accepts byte ranges. If the target range begins or ends within a page, the partial pages are copied into newly allocated pages in the tmpfs file. With 4 KB pages, this copies at most 8 KB per buffer. For production-scale Arrow tables, buffers are typically much larger (e.g., a 100K-row INT64 column occupies roughly 800 KB), thus the boundary copying accounts for less than 1% of the data. For tiny buffers, Kelvin heuristically disables DeAnon and copies them directly to avoid the overhead of de-anonymization.

### 4.2 DeDuce: Zero-Copy Communication

While DeAnon eliminates copying when data is first shared between processes, many pipeline transformations introduce copying when producing outputs. In common analytics operations such as filtering, projection, sorting, or appending columns, large portions of the output data are identical to buffers present in the input. However, existing communication protocols such as Arrow IPC serialize outputs independently, copying all data even when buffers could be reused.

To eliminate this redundancy, Kelvin introduces *DeDuce*, a communication mechanism that preserves buffer sharing

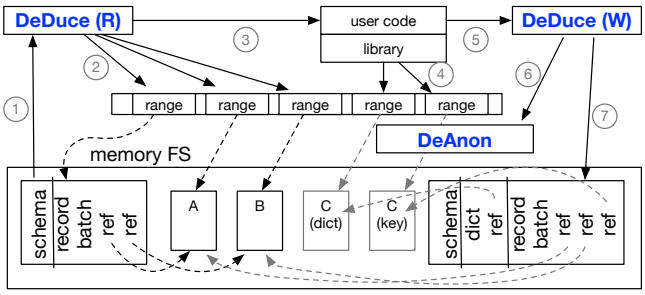



Figure 5: **Zero-Copy Inter-Process Communication.** (1) File generated by DeDuce is passed to the DeDuce reader (2) Reader maps the referenced files into its address space and returns a tabular view over the mapped data. (3) Mapped data is passed to user code, which may perform arbitrary actions and (4) generate new data. (5) Kelvin wrapper receives the Arrow output data. (6) If DeDuce identifies that some buffers refer to anonymous memory, DeAnon de-anonymizes those pages into in-memory files. (7) Finally, DeDuce writes references to the input files and new files in the output.

across pipeline transformations. DeDuce observes that Arrow data consists of immutable buffers and that many transformations reuse existing buffers while constructing new metadata. By detecting when output buffers correspond to existing input buffers, DeDuce emits references to previously shared memory instead of copying the data. Combined with DeAnon, this allows pipeline stages to share both newly produced data and reused input buffers through the in-memory file system with minimal copying.

#### 4.2.1 Design

DeDuce eliminates input–output copying by detecting when output Arrow buffers reuse memory from input buffers and emitting references instead of copying the data. Figure 5 illustrates the read (R) and write (W) behavior of DeDuce during a transformation that produces new outputs from mapped inputs. Both inputs and outputs reside in the in-memory file system. DeDuce is invoked by the Kelvin wrapper (shown in Figure 3 but omitted here for clarity).

**IPC Inspection:** Because Kelvin executes arbitrary user code, it cannot rely on explicit data lineage to determine how outputs relate to inputs. Instead, DeDuce performs *IPC inspection*, comparing the memory addresses of output buffers with the address ranges of previously mapped input buffers. If an output buffer refers to memory in an input file, DeDuce writes a reference rather than copying the data.

Figure 5 illustrates this process. The input table contains two columns (A and B). After the user function executes, the output table contains columns A, B, and a newly created column C. During IPC inspection, DeDuce detects that the output buffers for A and B reference the same memory as the inputs and therefore emits references to the existing files. If an output buffer instead refers to anonymous memory (e.g., for C), DeAnon is invoked to convert those pages into file-backed memory before they are referenced in the output.

**Dictionary Sharing:** Some transformations (e.g., ap-

pending columns or slicing rows) reuse entire input buffers. Others (e.g., filtering or sorting rows) reuse data at a finer granularity. In these cases, DeDuce often must copy data, because the output buffers differ from the inputs. However, with dictionary encoding, DeDuce can still preserve sharing. Dictionary buffers containing unique values are reused directly, while only the column of dictionary lookup codes is copied. As a result, dictionary encoding enables sharing even for transformations that reorder or filter rows.

#### 4.2.2 Implementation

DeDuce represents Arrow data in the in-memory file system using a set of files that reference shared memory buffers. Instead of serializing buffers inline as in standard Arrow IPC, DeDuce stores buffers in tmpfs files and writes references to those buffers in a small Arrow message file.

**Per-Column Files:** DeDuce organizes shared buffers using a small set of tmpfs files. Each column in a table is assigned a file, and an additional file is used for any dictionaries. During output generation, DeDuce de-anonymizes the memory backing each column buffer using DeAnon, transferring ownership of the pages to the corresponding file. Using separate files per column allows unused columns to be garbage collected. Finer granularity is possible (e.g., per-batch files), but Kelvin avoids creating many small files.

**Reference-Based IPC:** Instead of copying buffers into the Arrow IPC stream, DeDuce writes a tuple containing the file identifier, offset, and length of each buffer. These tuples reference the corresponding data in tmpfs files. On the read side, DeDuce reconstructs the Arrow table by mapping the referenced files using `mmap`. The Kelvin wrapper records the returned address ranges so that later writes can detect buffer reuse during IPC inspection.

DeDuce extends the Arrow IPC implementation (v12.0.1) to support reference-based messages. Small schema messages are copied, while large buffers such as record batches and dictionaries are referenced through tmpfs files.

**Container Integration:** Kelvin modifies SOCK to provide Arrow-oriented entry points rather than HTTP-based invocation. The Kelvin wrapper runs inside containers, receiving inputs and sending outputs over a UNIX domain socket, allowing references to tmpfs-backed Arrow buffers to be passed between the engine and containers. All interactions with DeDuce occur within the wrapper; user code simply receives and returns Arrow objects.

### 4.3 DeCache: Deserialization Cache

Duplication also occurs when multiple jobs operate on the same source data (Challenge 3). Independent DAGs often begin by deserializing identical inputs (e.g., Parquet files in persistent storage) into Arrow format. Kelvin avoids this duplication by restructuring such pipelines so they begin from a shared loader node. The loader caches the deserialized data and serves future DAGs with the same inputs. This service forms the deserialization cache, or DeCache.

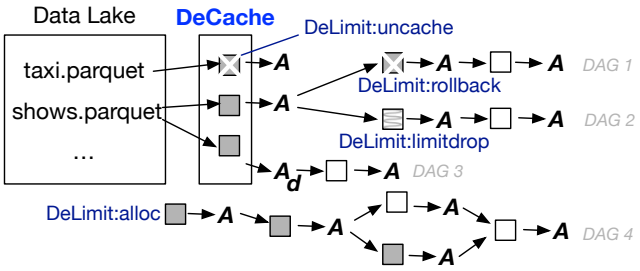



Figure 6: **Resource Management.** Finished nodes are gray.

#### 4.3.1 Design

Figure 6 illustrates DeCache. DeCache consists of a set of loader containers that deserialize Parquet data from storage. These loaders use DeDuce to efficiently de-anonymize the resulting Arrow data, just as regular DAG nodes do. However, unlike regular nodes, loader containers persist after DAG completion so their data can be reused by later DAGs. Different DAGs may deserialize the same source data with different options (e.g., with or without dictionary encoding for specific columns). In these cases, separate loader containers produce each representation. Each DeCache entry is keyed by its input file name and deserialization options.

#### 4.3.2 Implementation

We implement DeCache in Go, building upon DeAnon and DeDuce. When a DAG specifies loading a Parquet file with given options, Kelvin checks the cache for the same input and options, and maps that data upon a hit. Upon a miss, Kelvin launches a loader container whose "user code" calls PyArrow's `parquet.read_table` to deserialize the file into Arrow format. Although Parquet loading could be implemented to write Arrow data directly into the in-memory file system, we instead use the unmodified `parquet.read_table` implementation, which produces Arrow data in anonymous memory. Kelvin then applies DeAnon (§4.1) to efficiently expose the resulting buffers in our modified Arrow format.

### 4.4 DeLimit: Resource Management

Source data, intermediate data, and running containers all consume memory. Kelvin must track and limit overall usage to maximize parallelism while avoiding excessive swapping and process kills. Memory consumption is particularly difficult to track when many containers map the same physical data via DeDuce (§4.2) and DeCache (§4.3). DeLimit is the resource manager in Kelvin that tracks resource usage and makes admission and eviction decisions accordingly.

#### 4.4.1 Design

DeLimit performs three functions: tracking dependencies on de-anonymized data, admitting ready DAG nodes, and evicting in-memory files when necessary. DeLimit also includes an adaptive eviction policy.

**Garbage Collection:** DeLimit associates Arrow outputs with the in-memory files that store their buffers; this relation-

ship may be many-to-many due to DeDuce and DeCache. Using this information, DeLimit maintains reference counts for in-memory files and deletes them when no dependent nodes remain. These counts also guide eviction decisions because freeing a shared file may require removing multiple dependent outputs. DeLimit obtains this information directly from DeDuce. DeCache files are treated differently: they are not immediately deleted at zero references because future DAGs may reuse them.

**Admission Control:** DeLimit admits a node only when available memory (not reserved by running nodes or consumed by in-memory files) satisfies its memory requirement. When multiple nodes are waiting, DeLimit prioritizes nodes closer to completing a DAG (depth-first order). Completing DAGs earlier allows intermediate data to be deleted sooner, freeing memory for other jobs.

**Eviction of In-Memory Files:** Admission control may lead to deadlock if intermediate data exhausts memory and all DAGs are waiting for additional memory to proceed. DeLimit therefore evicts in-memory files to free up enough memory for the next node. As illustrated in Figure 6, DeLimit performs three types of eviction: *uncache*, *rollback*, and *limitdrop*. First, DeLimit uncaches DeCache entries with no active references by deleting their files. Second, it selects outputs from low-priority nodes (earliest depth) and either rolls them back or limit-drops them. Rollback deletes a node's container and outputs so the node can be re-executed later. Limit dropping, adapted from Senpai [39], swaps the node's in-memory files to disk; this is preferable to default swapping, which may evict data needed by scheduled nodes.

**Adaptive Eviction:** The choice between rollback and limit dropping depends on the relative cost of recomputation versus swapping. DeLimit therefore uses *adaptive eviction*, selecting rollback or limit dropping based on the ratio between a node's execution latency and the size of its output.

#### 4.4.2 Implementation

DeLimit is implemented in Go and interacts with other subsystems to collect dependency information and perform memory-management actions. SOCK containers use cgroups to enforce memory limits. Tmpfs files produced by de-anonymization (§4.1) are charged to the cgroup that created them, regardless of which containers later map the file; this is illustrated in Figure 3 with the file placed within Node 1's boundary.

To maintain control over these files after a node finishes, DeLimit allows the process in the container to exit but retains the container and its cgroup until the DAG completes. For limit dropping, DeLimit lowers the container's memory limit by reconfiguring the cgroup so that its in-memory files are swapped out. Before running downstream nodes, DeLimit raises the limit again so the data can be swapped back in if needed. In our implementation, the threshold for choosing between rollback and limit dropping is tuned offline based on the throughput of the swap device.

| | |
|---|---|
| CPU | Two Intel Xeon Silver 4314 16-core CPUs |
| RAM | 256GB ECC DDR4-3200 Memory |
| DISK | Two 960GB Samsung PCIe4 x4 NVMe SSD |
| OS | Ubuntu 22.04, kernel version 5.15 |

Table 1: **Hardware for Evaluation**. Note that the actual memory limit of each experiment is enforced by cgroup, not the RAM size. Input parquet files reside on one disk and the other is used as the swap device. SMT is disabled on CPUs.

### 4.5 Generalizability and Deployment

Kelvin's zero-copy path targets single-machine DAGs where large intermediate objects are immutable and exposed through mmap-compatible, memory-backed storage. Therefore, DeAnon requires immutable, pointerless data structures. Arrow and tmpfs are thus natural fits, but systems based on POSIX shared memory, memory-mapped files, or file-backed local storage could also be applicable.

The ideas behind other Kelvin components are more generalizable. Internal data duplication detection (DeDuce), input reuse across DAGs (DeCache), many-to-many dependency tracking, and DAG structure-based admission/eviction decisions (DeLimit) could be applied to other DAG runtimes, even when the underlying mechanism for sharing differs.

The main deployment challenges for Kelvin are disabling container reuse and using a kernel module. Kelvin disables container reuse to simplify accounting; it imposes only a small cost to our targeted workloads, as OpenLambda-like systems can start containers in roughly 100 ms for our data pipelines running for tens of seconds [35]. Container reuse breaks the one-to-one relationship between a container and a node output, adding complexity to dependency tracking and limit dropping. Dependency tracking could be extended with additional bookkeeping, but limit dropping would require more sophisticated container management, such as rebinding reused containers to a new cgroup for each execution.

We value kernel module's transparency to existing applications and libraries. An alternative would be to modify runtimes or libraries to allocate directly into shared memory. However, many libraries have their own Arrow implementations [21] and memory allocators, making ecosystem-wide changes much more invasive than a single kernel extension.

## 5 Evaluation

We evaluate Kelvin to understand how eliminating redundant data copying and duplication affects the performance, memory usage, and scalability of DAG-based data pipelines. Our evaluation proceeds in three stages. First, we study the four main components of Kelvin (DeAnon, DeDuce, DeCache, and DeLimit) in isolation to understand their impact (§5.1). Second, we evaluate Kelvin on a broader set of pipelines built using common data ecosystem libraries (§5.2). Finally, we run complex real-world DAGs derived from the DABstep benchmark to evaluate Kelvin end-to-end (§5.3).

Our experiments run on the hardware shown in Table 1. Unless otherwise stated, we use a scaled-down system mem-

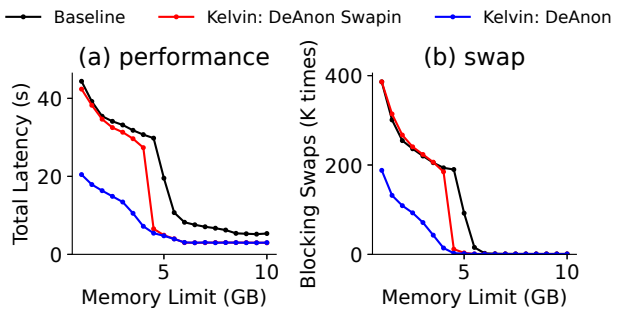



Figure 7: **Anonymous-Referable Copy Avoidance.** In (b), the y-axis is the number of foreground swap-in events.

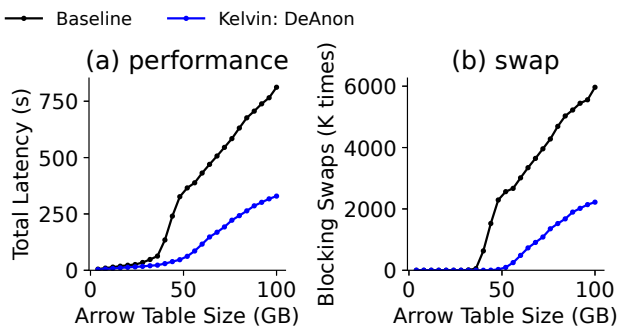



Figure 8: **DeAnon Scalability.** In (b), the y-axis is the number of foreground swap-in events.

ory limit of 50 GB alongside proportionally scaled-down datasets to keep experiment turnaround time reasonable. The 50 GB limit does not imply that Kelvin targets machines with little memory. Instead, this setup evaluates the same core regimes as larger deployments in a controlled manner: cases where intermediate data fits comfortably in memory, where redundant copies create memory pressure, and where eviction or recomputation is required. We then use the scale-up experiment in Section 5.1.1 to show Kelvin's scalability.

For all experiments, input Parquet files are loaded from local storage to avoid the variability of cloud storage (e.g., AWS S3). Our baseline is Kelvin with all new optimizations disabled: it uses Arrow IPC with read-side memory mapping for data passing, but otherwise copies all intermediate data. The baseline resource manager performs only admission control, data passing, and reference counting, without DeCache or advanced eviction.

## 5.1 Individual Kelvin Components

Our experiments on the four separate components of Kelvin answer the following questions. How much can **DeAnon** reduce latency and memory consumption by avoiding the copy between anonymous and referable memory? Which workload characteristics influence **DeDuce**? How much can **DeCache** reduce memory consumption and increase parallelism by avoiding repeated data loads? Which eviction strategies in **DeLimit** perform best under memory pressure?

### 5.1.1 DeAnon: Anon-Referable Deduplication

We first evaluate the benefits of avoiding the copy between anonymous and referable memory (Challenge 1). When a parent node produces anonymous Arrow output that child nodes may consume, Kelvin uses DeAnon to convert the output to referable memory without copying.

Figure 7 quantifies the benefits of DeAnon for a simple single-function DAG. The function uses PyArrow to deserialize a 0.5 GB Parquet file (compressed) into about 4.0 GB of Arrow data (uncompressed). The resulting Arrow output is then made referable for downstream nodes. Additional memory is required during processing, leading to a peak memory footprint of about 5.8 GB.

Figure 7a shows the latency of executing the DAG with varying cgroup memory limits for the baseline system, Kelvin with DeAnon enabled, and a variant of DeAnon without the swap-aware optimization. Overall, DeAnon is 1.8× faster than the baseline for high memory limits (i.e., 10 GB) and 2.2× faster for low limits (i.e., 1 GB).

For high memory limits, DeAnon is faster because it avoids the additional copy performed by the baseline system. For low memory limits, DeAnon performs less swapping, as shown in Figure 7b. The baseline temporarily duplicates the Arrow data (anonymous and referable copies), increasing memory pressure and triggering more swap activity. In contrast, DeAnon maintains only a single copy. When swapping is unavoidable, DeAnon can directly copy swap entries from the page table to the tmpfs file, avoiding additional swap-in operations. Disabling this optimization ("DeAnon Swapin") causes Kelvin to behave similarly to the baseline under low memory limits.

We further study the scalability of DeAnon using the same single-function DAG with Parquet inputs producing Arrow tables up to 100 GB. The memory limit is fixed at 50 GB. Figure 8 shows the latency comparison. DeAnon scales linearly when the working set fits in memory (Arrow tables $<$ 50 GB). Because it avoids duplicating data, DeAnon can also process larger tables without swapping: the baseline begins swapping at about 36 GB tables, whereas DeAnon remains in memory longer due to its smaller footprint. When the working set exceeds available memory, DeAnon's performance degrades more gracefully than the baseline. Overall, DeAnon provides roughly 2× speedup across all tested sizes.

### 5.1.2 DeDuce: Input-Output Deduplication

We evaluate the benefits of input–output deduplication (Challenge 2). DeDuce records the address ranges of buffers read from upstream nodes and reuses those references in outputs, avoiding additional copying or calls to DeAnon.

We quantify these benefits using simple two-node DAGs: the first node loads data and the second performs a transformation whose output overlaps with its input. To isolate DeDuce's impact from DeAnon, we measure only the latency of the second node; the baseline's copy of the input from

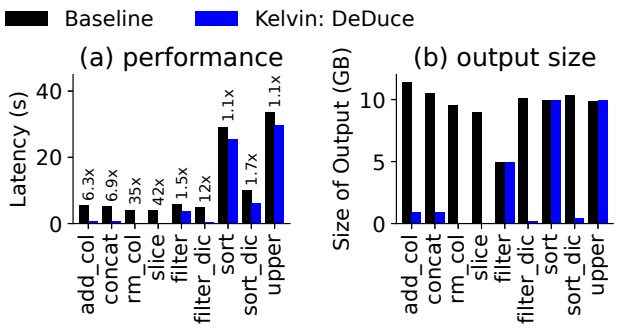



Figure 9: **Input-Output Deduplication: Various Transformations.** Numbers above bars in (a) are performance gains of Kelvin over Baseline.

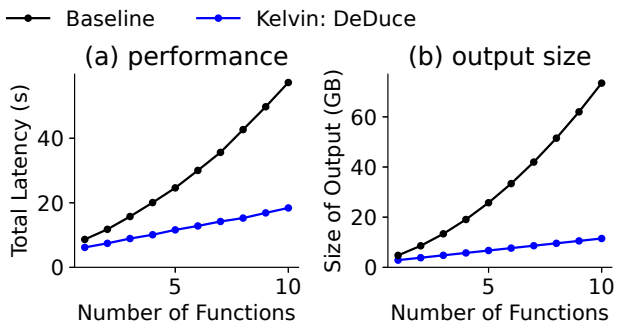
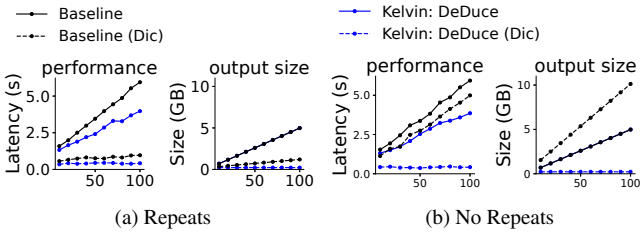



Figure 10: **Col_Add DAGs of Different Lengths.** The x-axis is the number of column-adding functions in a DAG.




(a) Repeats (b) No Repeats

Figure 11: **Deduplication of Dictionaries for Filter Operations.** The x-axis is the string size in bytes. Each unique string appears 10 times in (a) and once in (b). Strings are dictionary encoded for dashed lines.

anonymous to referable memory is not counted. We evaluate nine transformations: `add_col`, `concat`, `rm_col`, `slice`, `filter`, `filter_dic`, `sort`, `sort_dic`, and `upper`. The first four operate on tables with ten 1 GB integer columns; the remaining five operate on tables with ten 1 GB string columns (100-byte strings with no repetition). For all, data fits in memory and swapping is not triggered.

Figure 9 shows that DeDuce significantly reduces both latency and output size for most transformations. These workloads fall into three categories.

**Simple Additive and Subtractive Transformations:** For additive operations (e.g., `add_col` or `concat`), DeDuce incurs time and space costs only for the new data while reusing all existing input buffers. For subtractive operations (e.g., `rm_col` or `slice`), DeDuce completes almost immediately and produces minimal new data. These transformations exhibit coarse-grained overlap where large ranges of input buffers are reused directly.

Figure 10 shows the impact of this reuse in deeper pipelines. Starting with a 2 GB table (two columns), each node adds a new 1 GB column derived from existing columns. With DeDuce, both latency and cumulative output size grow linearly because each column is written to tmpfs only once. In contrast, the baseline rewrites all intermediate data at every stage, causing intermediate data size and runtime to grow quadratically. Thus, DeDuce's benefit increases with pipeline depth.

**Transformations using Dictionaries:** Filtering and sorting involve fine-grained overlap: many rows appear in both input and output, but their order changes, preventing large contiguous reuse. Consequently, DeDuce provides limited benefit for `filter` and `sort`. However, when dictionary encoding is used (`filter_dic`, `sort_dic`), DeDuce shares the large dictionary buffers while copying only the lookup codes, significantly reducing latency and output size.

Figure 11 examines this interaction for filter operations as a function of string size and repetition. The dataset contains 10M rows with one integer column and ten string columns. In Figure 11a, each string value appears ten times per column. Dictionary encoding benefits both the baseline and DeDuce by storing each string once. As string size increases, both systems improve due to reduced duplication.

In Figure 11b, each string appears only once. With dictionary encoding, the baseline's output grows slightly due to the additional lookup table. In contrast, DeDuce shares the dictionary buffers directly, producing negligible intermediate output and completing quickly. Thus, Kelvin introduces a new motivation for dictionary encoding: enabling fine-grained sharing even when values are not repeated.

**Transformations with No Deduplication:** `Upper` converts strings to uppercase, producing new buffers. Arrow's UTF-8 encoding causes some characters to change length after case conversion, preventing reuse of even the offset buffers. Thus, DeDuce performs identically to the baseline.

### 5.1.3 DeCache: Input-Input Deduplication

DeAnon makes data referable across DAG nodes without copying, and DeDuce eliminates duplication between the input and output of a node. We now evaluate DeCache, which avoids redundant deserialization when multiple DAGs load the same input data.

Figure 12 evaluates a workload where many DAGs run in parallel and share the same Parquet input file. Each DAG loads a 1.5 GB dataset and then performs a filter operation. Admission control and eviction are disabled so that the experiment isolates the impact of DeCache.

Figure 12a shows that Kelvin without DeCache performs slightly better than the baseline, but both systems scale only to about 5–10 concurrent DAGs before throughput declines

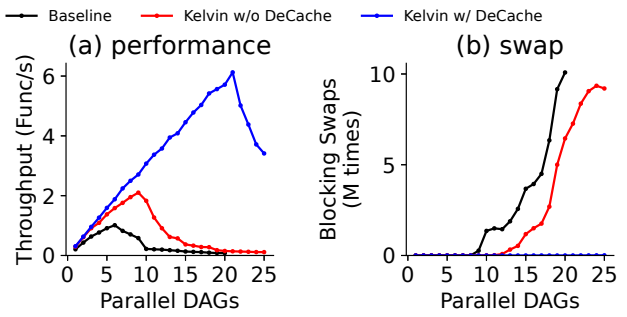



Figure 12: **DAGs with the same Inputs.** In (b), the y-axis is the number of foreground swap-in events in millions. Baseline crashes at x>20 because of OOM.

due to swapping. DeCache detects identical input loads across DAGs, performs the deserialization once, and shares the resulting Arrow data. This optimization improves performance in two ways. First, it avoids redundant loads, reducing execution time even with a small number of concurrent DAGs. Second, by eliminating duplicate in-memory copies, it substantially lowers memory consumption and allows many more DAGs to run concurrently. As a result, DeCache both reduces swap activity (Figure 12b) and dramatically improves throughput. With 20 parallel DAGs, DeCache achieves up to 38× the throughput of the baseline.

#### 5.1.4 DeLimit: Eviction Strategies

We evaluate the eviction strategies DeLimit uses to reclaim memory from unfinished DAG nodes. Without eviction of active nodes, deadlock could occur if all DAGs wait to proceed until others complete and release memory.

We compare three strategies. *Rollback* deletes a node and its outputs, recomputing them later if needed. *Limit dropping* lowers the node's cgroup limit to force its data to swap to disk. *Adaptive eviction* chooses between these approaches based on the ratio of node runtime to output size: if runtime/output is less than 5.25 s/GB, rollback is used; otherwise limit dropping is used. The threshold was selected from offline measurements of swapping out Arrow tables on our hardware; DeLimit is not highly sensitive to the exact value: thresholds from 3 to 7 s/GB change performance by less than 5%. We compare these techniques to a *Kswap* baseline that relies solely on the kernel's default swapping behavior; because the kernel is unaware of DAG dependencies and recomputation costs, it may swap out data that will soon be needed again, leading to unnecessary I/O and recomputation.

Figure 13a evaluates a workload of 15 sequential-chain DAGs, each containing 10 functions. The first function loads a 2 GB table, and each subsequent function appends a new 1 GB column, increasing memory consumption at every stage. The x-axis varies the computational work of each function: for $x = N$, the function performs columnar additions over $N$ 1 GB columns to produce one output column. The y-axis shows the normalized throughput.

Rollback consistently outperforms the *kswap* baseline by

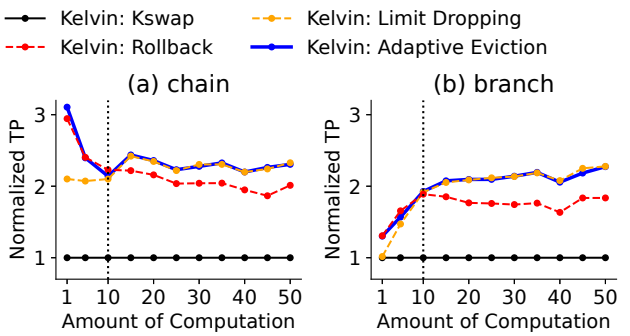



Figure 13: **Performance of Different Eviction Strategies.** The y-axis is the throughput normalized to *kswap*.

| Name | Library | Memory | Input-Output Copying |
|---|---|---|---|
| Matrix Mult | Numpy | variable | No |
| Linear Regr | Scikit | decreasing | No |
| Col Append 1 | PyArrow | increasing | Yes |
| Col Append 2 | DuckDB | increasing | No |

Table 2: **Ecosystem Benchmark DAGs**

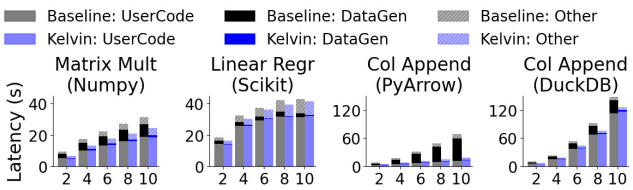
2.0–2.2×. Rollback works best when functions perform little computation ($x < 10$), since recomputing the output is inexpensive. For more compute-intensive functions ($x > 10$), limit dropping becomes preferable because swapping data is cheaper than expensive recomputation. Adaptive eviction selects the appropriate strategy and achieves the best performance across all configurations.

Figure 13b evaluates a different workload containing 15 concurrent DAGs with branching structure. Each DAG forms a tree of depth four with 15 nodes total. The same trends hold: rollback outperforms *kswap* by 1.3–1.8×, and adaptive eviction consistently delivers the best performance.

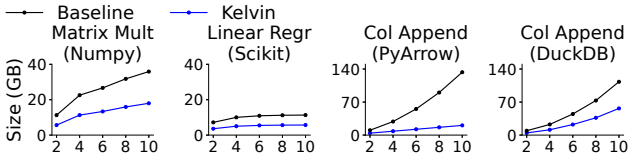
### 5.2 Ecosystem Pipelines

Our previous experiments used simple workloads to isolate the benefits of Kelvin's four components: DeAnon, DeDuce, DeCache, and DeLimit. We now evaluate Kelvin on a broader set of ecosystem pipelines built using common analytics libraries. Table 2 summarizes the four benchmark DAGs. These pipelines differ in the libraries used, memory usage patterns, and opportunities for input-output deduplication. All DAGs are sequential chains (without branches) so that the impact of intermediate data growth can be observed clearly. The DAG sizes are chosen such that even the baseline system fits within the 50 GB memory limit for the experiments in §5.2.1 and §5.2.2.

*Matrix Mult*: NumPy multiplies the input table by a matrix to produce an output with 1 to 10 columns randomly, leading to fluctuating memory usage across DAG nodes. *Linear Regr*: Scikit-learn performs linear regression where half the columns are randomly selected as features and the other half as labels; the table is split into equal-sized train/test sets, and each node predicts on the test set. Memory usage gradually




(a) performance




(b) intermediate data size (anonymous + referable)

Figure 14: **Ecosystem Benchmark on DeDuce.** The x-axis is the DAG length. In (a), latencies are divided into three parts: UserCode for running user functions where Kelvin does not help, DataGen for generating intermediate data where DeDuce takes effect, and other latencies such as container invocation and library loading.

decreases as the DAG progresses. *Col Append 1*: PyArrow computes four columns, concatenates them with the input, and returns the result. Appending columns increases intermediate data size and creates opportunities for input-output deduplication. *Col Append 2*: Similar to Col Append 1, but implemented with DuckDB SQL. DuckDB converts its internal representation to Arrow by copying in user code, preventing input-output deduplication.

### 5.2.1 DeAnon and DeDuce: Deduplication

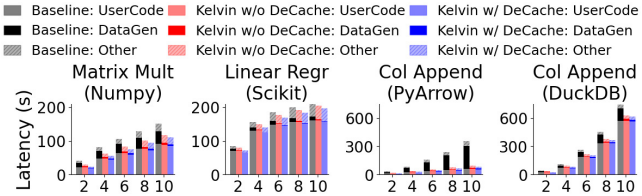
We run the ecosystem benchmarks on Kelvin to evaluate the combined benefits of DeAnon and DeDuce. Because Arrow IPC does not expose primitives for de-anonymizing memory, DeAnon cannot be evaluated independently of DeDuce.

Figure 14 shows that Kelvin consistently reduces intermediate data size across all workloads. For the baseline, intermediate data includes both the anonymous copy produced by the application and the referable copy written to tmpfs, since both contribute to peak memory usage. Kelvin therefore reduces intermediate data size by roughly 2×, and by more when input–output deduplication is possible.

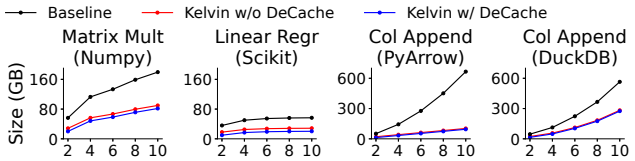
Latency improvements vary across workloads. Kelvin always accelerates intermediate data generation, providing larger benefits when data-copying cost dominates (e.g., Matrix Mult), but smaller benefits when user computation dominates (e.g., Linear Regr). In addition, DeDuce further helps in the presence of input-output duplication, providing the largest performance gain for Col Append (PyArrow). But copies performed internally by libraries are beyond Kelvin's control (e.g., DuckDB), resulting in a much smaller gain.

### 5.2.2 DeCache: Input-Input Deduplication

DeCache eliminates repeated deserialization across DAGs. We evaluate DeCache using the ecosystem benchmarks while varying DAG length from 2 to 10. For each configuration, we execute five DAGs sequentially. After the first DAG finishes, subsequent DAGs can reuse the cached deserialized input data produced by the loader node.




(a) performance




(b) intermediate data size (anonymous + referable)

Figure 15: **Ecosystem Benchmark on DeCache.** The x-axis is the DAG Length. Latencies division similar to Figure 14.

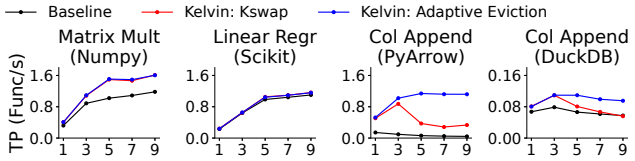



Figure 16: **Ecosystem Benchmark on Eviction Strategies.** The x-axis is the number of concurrent DAGs.

Figure 15 shows the results. DeCache eliminates repeated input loading and lets later DAGs reuse previously deserialized data, thus removing the cost of executing the loader function entirely and producing a consistent performance improvement across all DAG lengths. This benefit appears in both the UserCode and DataGen components of latency. Because all ecosystem workloads begin with a similar input-loading stage, the impact of DeCache is consistent across benchmarks. In each case, avoiding repeated deserialization reduces both computation and intermediate data generation.

### 5.2.3 DeLimit: Eviction Strategies

We evaluate eviction in the ecosystem benchmarks by running Kelvin with and without the adaptive eviction policy. The DAG length is fixed at 10, and we vary the number of concurrent DAGs from 1 to 9. Figure 16 shows the results. When multiple DAGs run concurrently and memory consumption grows at each stage (the *Col Append* workloads), enabling eviction substantially improves throughput: up to 4× compared to Kelvin without eviction and up to 28× compared to the baseline. The systems without eviction rely solely on generic kernel swapping. For workloads where memory usage does not steadily grow (Matrix Mult and Linear Regr), eviction provides little benefit. In these cases, upstream outputs are naturally released as the DAG progresses, freeing memory for later nodes.

| | DAG 1 | DAG 2 | DAG 3 | DAG 4 | DAG 5 |
|---|---|---|---|---|---|
| Load | 3 | 3 | 3 | 3 | 3 |
| Preproc. | 3 | 7 | 2 | 3 | 2 |
| Compute | 24 | 10 | 12 | 24 | 12 |
| Total | 30 | 20 | 15 | 30 | 17 |

Table 3: **DABstep Workloads:** Node count by type.

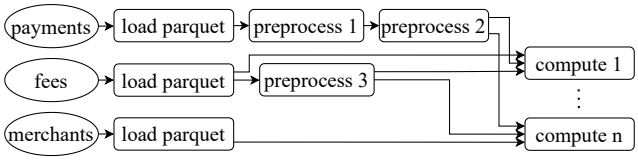



Figure 17: **Sample DABstep DAG**. Circles are parquet inputs. Boxes are functions. The load phase contains 3 nodes loading 3 tables. The preprocess phase contains 3 nodes, appending data to and filtering the input tables. The final computation phase involves tens of nodes in parallel.

## 5.3 End-to-End Evaluation with Real DAGs

To evaluate Kelvin end-to-end on realistic analytics pipelines, we use the Data Agent Benchmark for Multi-step Reasoning (DABstep) [3, 4]. DABstep contains analysis questions derived from real payment workloads at Adyen [1]. Each question corresponds to a multi-step data-processing DAG operating on transaction data. These pipelines exercise Kelvin's optimizations simultaneously.

We obtained solutions to five difficult questions and adapted them to run as Kelvin DAGs. Table 3 summarizes the structure of these DAGs, including the number of nodes responsible for loading data, preprocessing, and computing results. Loading nodes deserialize Parquet files to Arrow format. Preprocessing nodes filter and augment the data and partition it to enable parallel computation. Compute nodes perform the final task-specific calculations. All DAGs load the same three source tables: a 5 GB *payments* table, a 20 MB *fees* table, and a 5 MB *merchants* table. Figure 17 shows a representative DAG structure.

**Original DAGs:** Figure 18 (i) shows the performance and intermediate data size when running each DAG individually, as well as a workload executing all five DAGs concurrently. The load and preprocess phases dominate execution time, with relatively short compute phases due to high parallelism.

For the five individual DAGs, only DeAnon and DeDuce are active because neither input reuse nor memory pressure occurs. During load, Kelvin eliminates anonymous-referable copying, performing 1.4× faster while reducing intermediate data by roughly 50%. During preprocessing, DeDuce avoids input-output copying, performing 1.5× faster with 80% less intermediate data. The compute phase changes little because of little data generation. Overall, Kelvin runs the pipelines 1.2–1.6× faster while reducing intermediate data by 60%.

When all five DAGs execute concurrently, DeCache and DeLimit also become active because the DAGs share input tables and place greater pressure on memory. Kelvin achieves a 2.6× speedup over baseline and 1.8× over Kelvin

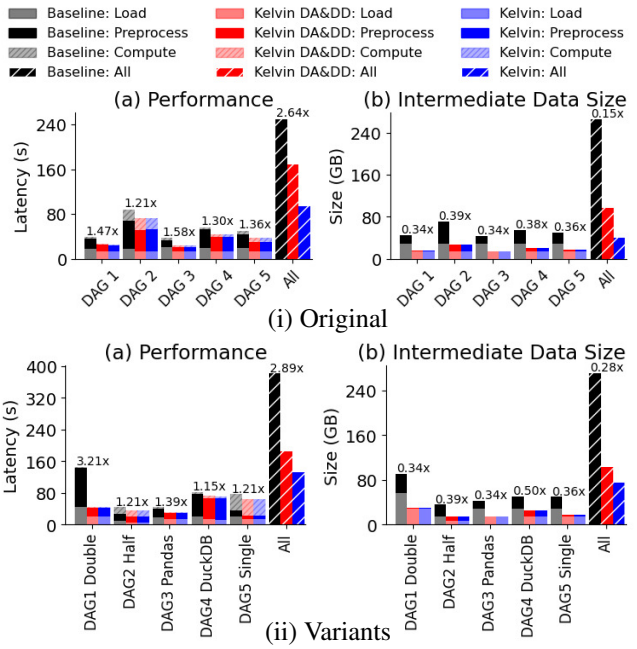



Figure 18: **Performance of DABstep DAGs.** Kelvin DA&DD stands for Kelvin with only DeAnon and DeDuce enabled. The numbers above the bars are comparisons between baseline and full Kelvin. Except for *All*, the latency and the intermediate data size are distributed to the load, preprocess, and compute phase. In (b), the intermediate data size includes anonymous and referable memory.

with only DeAnon and DeDuce. DeCache contributes most: loading the shared input tables only once reduces both computation and memory usage and enables greater parallelism.

**Variant DAGs:** We next modify each DAG to exercise additional scenarios, as shown in Figure 18 (ii). The five variants examine three factors: dataset size (*DAG1 Double* and *DAG2 Half*), processing libraries (*DAG3 Pandas* and *DAG4 DuckDB*), and computation intensity (*DAG5 Single*).

DAG1 Double doubles the data size, while DAG2 Half halves it. When the data size doubles, Kelvin's execution time increases only proportionally. In contrast, the baseline performs much worse because its working set exceeds the memory limit. Halving the data reduces both data generation and computation proportionally for both systems, so Kelvin's relative gain remains similar for smaller datasets.

For DAG3 Pandas and DAG4 DuckDB, we replace PyArrow with different data processing libraries, which copy data when converting to and from the Arrow format. Pandas can conditionally operate on Arrow data without copying, preserving some input-output sharing opportunities; consequently, Kelvin's gain decreases only slightly (from 1.6× to 1.4×). In contrast, DuckDB performs a full copy when converting back to Arrow, eliminating DeDuce's input-output deduplication and halving Kelvin's gain. Kelvin still benefits from DeAnon, but achieves larger gains when internal data-sharing opportunities are preserved.

DAG5 Single executes the computation in a single node

rather than in parallel. The compute phase is thus substantially longer and dominates execution time, roughly halving Kelvin's gain. This result illustrates Kelvin's least favorable case: compute-bound workloads with little data movement.

When all five variants execute concurrently, Kelvin's gain is higher than that for the original DAGs, because reducing redundant copies brings the combined working set back under the memory limit, outweighing the additional overheads.

Overall, Kelvin's components are effective in different regimes. DeAnon provides the largest benefit when data generation and copying dominate execution time, while DeDuce is most effective when transformations reuse input buffers. DeCache and DeLimit become important as pipelines share inputs and memory pressure grows. Consequently, Kelvin delivers its largest gains for workloads with substantial data movement and large intermediate results, while its benefit is naturally smaller for compute-bound workloads.

## 6 Related Work

Prior work on data pipelines, zero-copy communication, and kernel memory mechanisms has explored ways to reduce data movement or improve resource management. Kelvin differs by enabling copy avoidance for containerized DAG pipelines using existing Arrow-based applications without modifying user code.

**Data Pipelines:** Classic data-processing platforms such as MapReduce [11, 29] and Spark [41] impose a computational model and an intermediate data format, enabling system-level optimizations. In contrast, orchestration tools such as Apache Airflow [6], Luigi [30], and Metaflow [34] allow arbitrary code and data passing but provide little support for avoiding copying between pipeline stages. Kelvin takes a middle ground: it allows arbitrary node code while standardizing intermediate data on Arrow. Kelvin targets workloads similar to those supported by an existing commercial Arrow-based pipeline platform [35], but uses DeAnon and DeDuce to eliminate copying that occurs in that system when relying on a standard Linux kernel and Arrow IPC.

**Storage-Based Communication:** Many data-processing platforms exchange intermediate data through distributed storage systems such as HDFS or cloud object stores (e.g., S3). However, storage-based communication often introduces latency and variability [24]. Systems such as SONIC [19] and SAND [5] combine remote storage with local file systems based on online profiling and function placement. In contrast, Kelvin focuses on single-machine execution, which is increasingly practical as hardware capacity grows relative to workload sizes [26]. Modern systems can therefore run many pipelines entirely in memory and take advantage of faster local communication [20, 33, 36].

**Zero-Copy I/O Stacks:** Several systems reduce copying in I/O stacks. PASTE [14] and Fastmove [32] use DMA to minimize data movement, while Arrakis [25] redesigns the kernel to allow applications direct access to I/O. zIO [31] deduplicates buffers transparently when applications modify only small portions of data. Kelvin instead focuses on eliminating redundant copying between stages of data pipelines.

**Zero-Copy IPC:** Page-table manipulation has long been used to avoid copying during inter-process communication. Fbufs [13] uses page remapping but requires applications to explicitly identify shared buffers. IO-Lite [23] extends this idea with a *buffer aggregate* abstraction that allows outputs to reference inputs. Kelvin applies similar ideas within the Arrow ecosystem, adding mechanisms such as dictionary sharing. RMMap [17] provides RDMA-based shared memory across machines, whereas Kelvin focuses on efficient sharing across processes and containers on a single machine. Nightcore [15] also mounts a shared tmpfs across containers, but targets low-latency RPC between microservices; Kelvin instead optimizes for DAG pipelines where large outputs may be consumed by multiple downstream nodes. Arrow's experimental Dissociated IPC Protocol [8] proposes tagging message contents to enable reference-based communication, which could provide a standardized way to implement mechanisms similar to DeDuce.

**Linux Kernel Mechanisms:** Several kernel features relate to Kelvin's design. Kernel Samepage Merging (KSM) [12] scans memory and deduplicates identical pages after they are created, whereas DeCache avoids duplication when data is first produced. Senpai [39] introduced the limit-dropping technique for controlling swapping; although abandoned for general workloads, it is well suited to Kelvin, where it is applied only to intermediate pipeline data. Proposed APIs such as *process_vm_mmap* [37] and *msharefs* [9] would allow processes to share VMAs or page tables more directly. Kelvin instead exposes shared data through in-memory files via de-anonymization, providing explicit control over visibility and lifetime of shared memory.

## 7 Conclusion

Zero degrees Celsius is not the coldest theoretical temperature, and the term "zero-copy," as commonly used, does not actually mean no copying. We identified three types of copying that are often overlooked in data pipelines. Pursuing an "absolute zero" ideal, we introduced Kelvin, a data pipeline execution tool that avoids copying with new kernel support, an improved communication protocol, and a deserialization cache. Kelvin avoids copies across DAGs, between nodes in a DAG, and within individual nodes, and thus improves throughput for complex workloads by 1.2-28×.

## 8 Acknowledgement

We thank Mingxing Zhang (our shepherd), the anonymous reviewers, and the ADSL group for their valuable feedback. This material was supported by Bauplan and NSF grant CNS-2402859. Any opinions, findings, conclusions, or recommendations expressed in this material do not reflect the views of NSF or other institutions.

## References


[1] Adyen. https://www.adyen.com/.

[2] New Orleans, Louisiana, February 1999.

[3] Adyen and Hugging Face. Dabstep. https://huggingface.co/datasets/adyen/DABstep.

[4] Adyen and Hugging Face. Data agent benchmark for multi-step reasoning. https://medium.com/adyen/data-agent-benchmark-for-multi-step-reasoning-dabstep-70e913c339dc.

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX '18)*, Boston, MA, July 2018.

[6] Apache. Apache Airflow. https://airflow.apache.org/.

[7] Apache. Arrow. https://github.com/apache/arrow.

[8] Apache. Dissociated ipc protocol. https://arrow.apache.org/docs/format/DissociatedIPC.html.

[9] Khalid Aziz. memshare. https://lore.kernel.org/lkml/a1d6a3de-502e-4114-a603-01710e30428e@oracle.com/T/.

[10] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)* [2].

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, December 2004.

[12] Linux Kernel Developer. Kernel samepage merging. https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

[13] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. 27(5):189–202, December 1993.

[14] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, April 2018.

[15] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, Virtual Event, April 2021.

[16] Inc. Linux Kernel Organization. Tmpfs. https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html.

[17] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. Serialization/deserialization-free state transfer in serverless workflows. In *Proceedings of the EuroSys Conference (EuroSys '24)*, Athens, Greece, April 2024.

[18] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An Extensible Shuffle Architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 564–577, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX '21)*, Virtual Conference, July 2021.

[20] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In *USENIX Workshop on Hot Topics in Operating Systems*, 2015.

[21] Hannes Mühleisen and Mark Raasveldt. *duckdb: DBI Package for the DuckDB Database Management System*, 2026. R package version 1.5.4.3.

[22] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[23] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified IO buffering and caching system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)* [2].

[24] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, Portland, OR, June 2020.

[25] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[26] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. In *Proceedings of the 50th International Conference on Very Large Databases (VLDB 50)*, Guangzhou, China, August 2024.

[27] Amazon Web Services. Amazon ec2 high memory u7i instances. https://aws.amazon.com/ec2/instance-types/u7i/, 2026.

[28] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. Operationalizing machine learning: An interview study, 2022.

[29] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[30] Spotify. Luigi. https://github.com/spotify/luigi.

[31] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '22)*, Carlsbad, CA, July 2022.

[32] Jingbo Su, Jiahao Li, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, and Yinlong Xu. Revitalizing the forgotten On-Chip DMA to expedite data movement in NVM-based storage systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST '23)*, Santa Clara, CA, February 2023.

[33] Jacopo Tagliabue. You do not need a bigger boat: Recommendations at reasonable scale in a (mostly) serverless and open stack. In *Fifteenth ACM Conference on Recommender Systems*, RecSys '21, page 598–600, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Jacopo Tagliabue, Hugo Bowne-Anderson, Ville Tuulos, Savin Goyal, Romain Cledat, and David Berg. Reasonable scale machine learning with open-source metaflow. *ArXiv*, abs/2303.11761, 2023.

[35] Jacopo Tagliabue, Tyler Caraza-Harter, and Ciro Greco. Bauplan: Zero-copy, scale-up faas for data pipelines. In *Proceedings of the 10th International Workshop on Serverless Computing*, WoSC10 '24, page 31–36, 2024.

[36] Jordan Tigani. Big data is dead. https://motherduck.com/blog/big-data-is-dead/, 2023.

[37] Kirill Tkhai. process_vm_mmap. https://lore.kernel.org/linux-mm/155836082337.2441.15115541609966690918.stgit@localhost.localdomain/T/.

[38] Matthew Topol. *In-Memory Analytics with Apache Arrow: Perform fast and efficient data analytics on both flat and hierarchical structured data*. 2022.

[39] Johannes Weiner and Dan Schatzberg. Transparent memory offloading: more memory at a fraction of the cost and power. 2022.

[40] Doris Xin, Litian Ma, Shuchen Song, and Aditya G. Parameswaran. How developers iterate on machine learning workflows - a survey of the applied machine learning literature. *ArXiv*, abs/1803.10311, 2018.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.