Skadi: Building a Distributed Runtime for Data Systems in Disaggregated Data Centers

Cunchen Hu¹,² *, Chenxi Wang¹,², Sa Wang¹,², Ninghui Sun¹,², Yungang Bao¹,², Jieru Zhao³, Sanidhya Kashyap⁴, Pengfei Zuo⁵, Xusheng Chen⁵, Liangliang Xu⁵, Qin Zhang⁵, Hao Feng⁵, Yizhou Shan⁵

¹University of Chinese Academy of Sciences, ²State Key Lab of Processors, ICT, CAS
³Shanghai Jiao Tong University, ⁴EPFL, ⁵Huawei Cloud

ABSTRACT

Data-intensive systems are the backbone of today’s computing and are responsible for shaping data centers. Over the years, cloud providers have relied on three principles to maintain cost-effective data systems: use disaggregation to decouple scaling, use domain-specific computing to battle waning laws, and use serverless to lower costs. Although they work well individually, they fail to work in harmony: an issue amplified by emerging data system workloads. In this paper, we envision a distributed runtime to mitigate current shortcomings. The distributed runtime has a tiered access layer exposing declarative APIs, underpinned by a stateful serverless runtime with a distributed task execution model. It will be the narrow waist between data systems and hardware. Users are oblivious to data location, concurrency, disaggregation style, or even the hardware to do the computing. The underlying stateful serverless runtime transparently evolves with novel data-center architectures, such as disaggregation and tightly-coupled clusters. We prototype Skadi to showcase that the distributed runtime is practical.

ACM Reference Format:


1 INTRODUCTION

Motivation. Today’s cloud has been deeply shaped by data-intensive systems since the early 2000s when GFS [22] was built to cope with the boom of big data. The volume of data and the variety of data-intensive systems in the cloud increased exponentially in the past two decades [1]. Many data-intensive systems migrate to the cloud for instant infrastructure availability. Nevertheless, it is incredibly challenging to design, deploy, and run data systems fast enough at a reasonable cost, as they commonly use deep storage and can be both compute-intensive and memory-intensive [12, 67]. Over the years, users, data system designers, and cloud vendors have worked closely to keep running data systems cost-effective using three principles: (1) use disaggregation to enable independent resource scaling; (2) use domain-specific accelerators to battle waning laws and improve computing efficiency; (3) use serverless to lower costs and boost productivity.

These principles work well individually, but they fail to work in harmony as each was proposed to solve a different problem (explained next), an issue amplified by two recent trends. The first trend is data systems integration in which multiple data systems are deployed onto one pipeline that jointly runs business logic, data management, HPC, and ML [1, 4, 14, 26, 40, 46]. For example, BigQuery can run data ingestion, extraction, analytics, and ML in one job [26]. The second trend is that giant model training has evolved from using SPMD to MPMD over multiple highly-specialized clusters [7, 12, 79]. Both workloads are commonly written in high-level, declarative languages (e.g., SQL), and developers of these workloads are mostly oblivious to how the underlying computation is carried out. For the best cluster
cost-efficiency, both trends call for a close co-design of programming models, runtime, networking, data-center hardware, etc. To motivate this paper, we now discuss what the above principles are, how they conflict, and why they fail to accommodate emerging workloads.

First, cloud vendors adopt the disaggregation principle at two levels in unison for a higher cluster efficiency. At the logical level, vendors build separate resource pools using regular servers [6, 44, 58, 67]. This decouples compute from storage and enables each to scale independently to best fit data system needs. However, the unit for scaling and running tasks is limited by the rigid server box [59]. Physical disaggregation emerged to break servers into customized devices, which often consist of a dominant resource like DRAM or GPU along with ancillaries like DPU for networking and control [28, 37, 45, 59, 60]. It can greatly improve resource utilization. To date, co-designing data systems with logical-disaggregation is battle-tested [6, 13, 44, 65, 67]. But adapting data systems with physical-disaggregation is still in its infancy [21, 37, 78].

The second principle is that vendors seek domain-specific accelerators (DSA) to overcome limitations imposed by conventional computing power such as CPUs [30] to keep up with data systems’ increasing computing need [6, 10, 18, 33]. Unlike CPUs, DSAs have intrinsically different usage models. Hence using a high-level framework [19, 53] is crucial to reap their benefits. However, integrating DSAs with data systems is a demanding but mostly one-time effort [6, 18, 33, 42, 56], often creating “computing silos” in which DSAs are exclusively owned by a data system or a service [6, 32]. Such computing silos can be tightly-coupled clusters in which DSAs are interconnected via high-speed interconnect, essentially trading the scale of the cluster for the best performance. This can result in suboptimal cluster utilization, which conflicts with the disaggregation and pooling principle. It also makes sharing DSAs across distinct data systems more difficult as the DSAs are usually not exposed as-is but with added-values.

Finally, the serverless principle further lowers cost by offering a pay-as-you-go cost model over a reservation-based one and boosts productivity by hiding distributed clusters. However, existing commercial serverless lacks key features that hinder its broad adoption [57]. In particular, compute and storage are separated due to logical disaggregation. As a result, functions usually bounce data via durable cloud storage [36, 55]. Unfortunately, this is detrimental to data systems that heavily rely on a fast caching layer for storing states and ephemeral data exchanged across functions. This issue is more prominent with data systems integration. In addition, existing serverless uses a CPU-centric model in which users run general-purpose code on CPUs and orchestrate DSAs from there, and the auto-scaling of DSAs is almost non-existent [54]. Due to these limitations, users and even cloud services only partially migrate to commercial serverless and still exchange data via slow durable storage, as reflected in Figure 1b.

We believe the current ecosystem is not optimal for modern workloads written in high-level, declarative languages and running multiple demanding data systems in one pipeline. Ideally, we need a solution that simultaneously provides all the benefits enabled by the above three principles. The said solution must meet the following requirements: (a) has an easy programming model that enjoys the pay-as-you-go model for all the computing power used, (b) optimizes for cluster cost-efficiency such as reducing data movement whenever possible, and (c) transparently evolves with data-center infrastructure. In this paper, we propose a distributed runtime to close the gap.

Our work. We envision that the distributed runtime has a tiered access layer that exposes declarative data-centric and functionality-centric APIs, underpinned by a stateful serverless runtime with a distributed task execution model and flexible state management. It is the narrow waist between data systems and hardware (Figure 1c). As such, it achieves the separation of concerns at scale: users apply domain-specific declarative computation on data, oblivious...
of data location, concurrency, consistency, disaggregation style, or even the hardware used to do the compute. The tiered access layer will map declarations onto a universal shared graph that dictates how data flow through computing tasks. On the other hand, the underpinning stateful serverless runtime will run the graph and ensure efficient task scheduling and data movement, all the while transparently evolving with novel data-center architecture such as disaggregation [59] and tightly-coupled clusters [23, 32].

We believe it will be in the cloud vendors’ best interest to exploit the co-design of data systems, the distributed runtime, and the data center infrastructure in order to achieve the best cost-efficiency. As such, more user data systems and cloud services would migrate to it. This necessitates the distributed runtime to host data systems with varied execution models such as BSP [16, 31, 44, 75], task-parallel [35, 46, 49], streaming [48, 76], graph [24, 25, 50], ML [2, 7, 11, 79], etc. A flexible access layer and a stateful serverless runtime are key to realizing this.

The access layer is proposed based on decades of research and industry best practices on data flow. It consolidates fragmented domain-specific declarations onto one execution graph. It does so using three tiers: a domain-specific declarative tier, a logical graph tier, and a physical shared graph tier. Domain-specific ones include SQL [5], Graph [24], MapReduce [16], ML [2, 11], etc. They are collectively lowered onto one logical graph that hosts data-parallel, task-parallel, and iterative patterns at once, using hardware-agnostic vertices connected by directed edges. Lowering a logical graph to a physical graph means possibly creating shared vertices along keyed edges and then mapping vertices to hardware operators. Both the generated logical and physical graph dictate how data flows through vertices. Crucially, they do not specify when and who should execute the vertices, a task delegated to Skadi’s stateful runtime.

The tiering model mirrors Dryad [31, 73] and Naiad [48], but differs in how vertices are built. We propose to use IR-based primitives, in addition to predefined operators (e.g., handcraft kernels [31, 75] or wrapped utilities [34, 48]) to build the vertices in the logical graph, akin to [14, 34]. A common IR enables graph-level optimizations such as op-fusing across application domains, in contrast to being confined within one domain [11]. This IR is data-centric, focusing on expressing the flow of data and the scheduling of compute. It is also functionality-centric: distributed runtime developers focus on functionalities intended, free from selecting hardware or porting the same operator to multiple hardware platforms, a task complicated by the emerging disaggregated [28, 37] and heterogeneous devices [8, 10, 33, 56] with diverging hardware characteristics.

The stateful serverless runtime will execute the physical graph from the access layer. It resembles a task-parallel system with a universal dynamic task execution API [35, 46, 49], based on which many distributed computing patterns can be built, e.g., generic data-parallel and task-parallel patterns, or the specialized MPMD pattern in giant model training [7]. The stateful serverless runtime has a flexible control plane and a fast data plane.

Its control plane is responsible for resource management, task dispatching, auto-scaling, etc. For workloads with long-running operators (e.g., analytics), it has a minor impact on performance but determines utilization. For workloads with frequent short operators (e.g., ML), it determines performance. To this end, the control plane embraces data-centric scheduling [27] for higher utilization, and forgoes the CPU-centric model to better support short-lived operators on heterogeneous hardware. If necessary, it could also integrate gang-scheduling to support SPMD-style sub-graph [7].

The stateful serverless runtime’s data plane is critical as data systems normally move a sizable amount of data to finish a job, reflected in the data size per transfer, the cost paid per transfer, and the number of transfers. One can certainly use direct hardware channels [15, 52] or novel networks [23, 43] for good performance. Above all, we believe a fast caching layer with a standard format [3] is the bedrock of our data plane. It enables stateful functions because it can store states, external storage’s input/output, and ephemeral results exchanged by functions within a job or across systems. Crucially, it has four benefits: (1) It decouples compute from states so compute (i.e., vertices) can be opportunistically migrated to where data reside to reduce data transfer. (2) A shared format such as Arrow [3] enables functions running on heterogeneous devices to exchange data without costly data marshalling, hence reducing the cost paid per transfer. (3) It unites data systems within an integrated pipeline using futures [7, 46, 63], thus enabling pipeline parallelism across system boundaries. Also, it can reduce the number of transfers to durable storage. (4) An optional highly-available caching can replace lineage [46], adding another design trade-off.

We prototype Skadi to showcase that the distributed runtime vision is practical. For the access layer, we build FlowGraph as the logical graph. Its vertices use a multi-level IR [41] to express hardware-agnostic computation. Skadi will lower the logical graph to a physical one. The physical graph is a set of linked Python objects in our current implementation. Skadi will parse the graph and then launch tasks accordingly using stateful serverless runtime’s task API. Skadi’s stateful serverless runtime is built based on Ray. We first offload Ray’s control and data plane to DPUs and extend its ownership table to enable Ray on physically-disaggregated devices. To support short-lived ops, we eschew Ray’s CPU-centric model and offload part of its control plane (i.e., raylet) to heterogeneous devices. In addition, we add another push-based future resolution scheme.
2 DESIGN AND RESEARCH AGENDA

We prototype Skadi to demonstrate that the distributed runtime idea is viable. This section will discuss Skadi’s initial design, research agenda, and open questions.

2.1 Skadi Overview

We present Skadi’s architecture in Figure 2. Skadi enables users to use only one runtime to express all of their programs as a part of this runtime. The input consists of several domain-specific declarations like SQL statements and ML training (e.g., a python script). Skadi maps the program onto a logical graph in two steps: (1) invokes domain-specific parsers to translate declarations onto a common graph called FlowGraph, whose edges dictate how data flow and vertices are built with either handcraft operators (ops in short) or hardware-agnostic ops using MLIR [41]; (2) optimizes the graph using predefined rules.

Skadi lowers the logical FlowGraph to a physical sharded graph in two steps: (1) selects hardware backends for MLIR-based ops using predefined rules; (b) decides a default degree of parallelism for each vertex (subscripts in Figure 2), and keyed edges with a default or user-supplied hashing scheme.

Skadi launches functions according to a given physical graph using stateful serverless runtime’s task APIs (pseudo-code in Figure 2). Functions exchange data either by value or by reference. Before scheduling a function, the runtime decides the preferred hardware based on memory locality, device availability, network topology, etc. Once dispatched, the function either runs immediately regardless of whether the input is available or not, or waits until the input is ready. The wait mode is possible because (a) Skadi’s control plane supports data-centric scheduling [27] and (b) Skadi supports pass-by-reference (i.e., futures) and has a flexible caching layer. The caching layer has a simple KV API for memory on regular servers, memory on heterogeneous devices, and disaggregated memory. Crucially, the caching layer can hide the location and movement of data.

Skadi handles failures in two ways: (1) re-executes the graph using lineage [46], or (2) uses a reliable caching layer with data replication [61] or EC [80]. Most existing data systems use lineage since replication is costly [35, 46, 49, 75]. However, a reliable caching layer could be beneficial as it helps reduce tail latency and potentially cost since the cost of restarting jobs may offset the cost of extra storage.

2.2 Access Layer

The access layer, as its name suggests, grants data systems access to the distributed runtime. Hence its top-level APIs must be compatible with existing ecosystems. Fortunately, it can reuse mature and open-source definitions, tools, and compilers [14, 19, 34, 53, 75]. For example, FlowGraph is a classical data flow graph, similar to the ones in [31, 48]. We also reuse cudf [52] ops, arrow ops [3], etc.

In building the access layer, we find a key challenge is defining the IR to build hardware-agnostic computations and further building FlowGraph with it. On the one hand, it should be generic enough to build computing patterns data systems commonly use. On the other hand, we should be able to lower it onto multiple hardware backends (e.g., CPU, FPGA, GPU, RMT [8]). No such IR exists today.

In response, we use MLIR [41], a compiler infrastructure for creating domain-specific compilers. Recent works [7, 14, 34] showcase that MLIR is viable to build data systems. We plan to use the open-source Daphne project [14] to build our IR because it is the closest to an ideal access layer (see Table 1): it has tiered declarative APIs, an MLIR-based DSL, and abstractions like data frames, and matrix operators. Although Daphne is versatile, its IR cannot be lowered onto distinct hardware since it only supports a single LLVM backend. In order to build hardware-agnostic IR, we must add more MLIR backends such as CIRCT for FPGA, or SPIR-V for...
GPU. This is an incredibly challenging task, which remains an open research question.

A key benefit of using hardware-agnostic IR is that we can lower a single piece of code to multiple hardware backends, based on a set of predefined policies. For example, in order to compare how an op performs on two platforms, the MLIR-based vertex D in Figure 2 is lowered onto a GPU version (D1) and an FPGA version (D2) for a direct comparison. This certainly alleviates developers’ chore of implementing and selecting the best hardware platform. In addition, should we finalize the degree of parallelism during the compilation time [75], or allow tuning [5] or graph reshaping [31] during runtime is an open question. We leave such policy explorations to future work.

2.3 Stateful Serverless Runtime

Skadi’s stateful serverless runtime is built based on Ray. Given a physical graph description, we will launch tasks (or actors) and pass futures using Ray’s vanilla distributed task and KV APIs (code snippet in Figure 2).

2.3.1 Ray Primer. Ray is a task-parallel system originally designed as a glue system to run RL workloads [46]. It has task APIs for users to launch functions using stateless tasks or stateful actors. Each Ray node runs a daemon process called raylet which is responsible for running tasks and managing a distributed object store called plasma. Functions exchange stateful actors. Each Ray node runs a daemon process called raylet which is responsible for running tasks and managing a distributed object store called plasma. Functions exchange data either by value or by futures. The future refers to data residing in the distributed object store. Future resolution uses an ownership protocol [24]. Due to the limited space, we refer readers to [46, 68, 69, 81] for more details.

2.3.2 Design. We now describe how we overhaul Ray to meet our goals described in Sec §1. We first offload raylet to disaggregated devices (recall that such a device consists of a DPU along with other dominant resources like GPU or DRAM [28, 37, 45]). To access memory on heterogeneous devices from Ray’s code using regular opaque pointers, we modify Ray’s ownership table [69] also to include a device ID and a handle for the device communication driver. Consequently, the raylet on DPU also manages memory on its companion devices. Combined, this results in the first generation of our stateful serverless runtime, capable of managing physically-disaggregated devices and their memory (Gen-1 in Figure 3). We are building it using an in-house card that has a BlueField DPU [51] and FPGAs.

The first generation can run most use cases efficiently. But it is inefficient in running short-lived ML ops for two reasons. First, it continues to use the CPU-centric model in which the DPU orchestrates all resources of a device. The management of tasks and pointers must go through the centralized DPU. For instance, if two chained ops from the same physical graph are deployed to two different FPGAs in Figure 3, their communication (e.g., future resolution) must go through the DPU. For short-lived ML ops, frequent trips to the DPU are too costly. Second, Ray’s future resolution uses a pull-based model in which the consumer pulls data from the producer on demand [69]. This creates long stalls for short-lived ops. Pathways [7] made a similar observation.

We propose a second generation (Gen-2 in Figure 3) to solve these issues. It adopts three key changes. First, we eschew the CPU-centric model by deploying a device-specific raylet to each heterogeneous device. Second, we add another push-based model for future resolution, in which the producer pushes data to the consumer proactively. Third, to resolve potential out-of-memory and to increase availability, we extend the caching layer to include disaggregated memory. This generation embraces a device-centric model and can run ops that call Ray APIs everywhere (e.g., ops written in Figure 3’s style-2). We are still in the early stage of designing a CUDA-based raylet for GPU. It’s intrinsically more difficult to build a similar raylet for FPGA despite its recent breakthroughs [38, 77].

3 RELATED WORK

Numerous data systems have been built over the years, e.g., parallel processing [16, 31, 75], task-parallel [35, 46, 49], large-scale ML [2, 7, 11, 79], OLAP [6, 44, 71, 72], graph processing [24, 25, 50], streaming [9, 48, 76], etc. As we mentioned earlier, cloud providers have relied on three principles to maintain cost-effective data systems. We compare these efforts in Table 1 across five dimensions.

(1) API and IR. We categorize the system APIs into POSIX, imperative, and declarative. Most emerging operating systems offer POSIX except FractOS [66], which exposes an imperative abstraction for users to write DAG manually. A similar approach is taken by several serverless frameworks [17, 36, 63] and task-parallel systems like CIEL [49], MODC [35], and Ray [46].
<table>
<thead>
<tr>
<th></th>
<th>API</th>
<th>IR</th>
<th>Serverless</th>
<th>PhysDisagg.</th>
<th>Integr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dist. OS [47]</td>
<td>POSIX</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>LegoOS [59]</td>
<td>POSIX</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>FractOS [66]</td>
<td>I-API</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Molecule [17]</td>
<td>I-API</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Cloudburst [63]</td>
<td>I-API</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Pocket [36]</td>
<td>I-API</td>
<td>×</td>
<td>stateful</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CIEL [49]</td>
<td>I-API</td>
<td>×</td>
<td>stateful</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Ray [46]</td>
<td>I-API</td>
<td>×</td>
<td>stateful</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>MODC [35]</td>
<td>I-API</td>
<td>×</td>
<td>stateful</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Pathways [7]</td>
<td>D-API</td>
<td>MLIR</td>
<td>stateful</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>OneFlow [74]</td>
<td>D-API</td>
<td>IR</td>
<td>actor</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Dryad [31]</td>
<td>D-API</td>
<td>×</td>
<td>stateless</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Naiad [48]</td>
<td>D-API</td>
<td>×</td>
<td>stateful</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DBOS [40, 62]</td>
<td>D-API</td>
<td>×</td>
<td>stateful</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TCR [20, 29]</td>
<td>D-API</td>
<td>IR</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>DAPHNE [14]</td>
<td>D-API</td>
<td>MLIR</td>
<td>stateless</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Skadi</td>
<td>D-API</td>
<td>MLIR</td>
<td>stateful</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Related work comparisons. (1) I-API: imperative APIs. D-API: declarative API. All non-POSIX systems support DAG. (2) IR: the system uses an IR for hardware-agnostic computation. (3) PhysDisagg: whether the system utilizes physically-disaggregated devices for higher cluster efficiency. (4) Integr.: whether the system runs integrated data system pipelines.

Skadi’s declarative API is inspired by early pioneers such as Dryad [31], Naiad [48], and DAPHNE [14]. Compared to DAPHNE, Skadi differs in using MLIR with multiple hardware backends to build hardware-agnostic ops [14].

(2) Serverless Runtime. If a system supports stateless functions, we believe it can support the serverless paradigm regardless of whether it was explicitly designed so. Dryad [31] is a prominent example. We categorize systems capable of storing function states as stateful serverless runtime. Note that we explicitly differentiate actor-based systems [39, 74], although generally they are also stateful systems. Typical stateful runtime include Cloudburst [63], Pocket [36, 64], etc. Pathways [7] has a data-flow system called Plaque which has a data store similar to Ray’s plasma, so we also mark it as a stateful runtime.

(3) Physical Disaggregation. Not many systems are capable of managing physically-disaggregated devices. LegoOS [59] and FractOS [66] are two seminal works in this space. LegoOS logically groups physically disaggregated devices and exposes a single-system image to run unmodified applications written for monolithic kernels. FractOS eschews the CPU-centric model and allows developers to write imperative DAGs to use disaggregated devices. However, neither of them is designed for running data-intensive systems. Some work optimizes data systems running on disaggregated devices [37, 78] but mostly targets a single system. Skadi extends the Ray runtime to disaggregated devices by offloading key components to DPUs.

(4) Data Systems Integration. In general, systems that aim to run mixed workloads need to provide an abstraction and a runtime. In Skadi’s case, we have the tiered access layer and the stateful serverless runtime, respectively. Prior works like DPA [39], DBOS [40, 62], and DAPHNE [14] also target integrated data systems. For example, DBOS can run serverless data system workloads on top of a distributed database. DPA provides an actor-based programming model for building distributed query serving systems and a shared runtime for running and scheduling actors. DAPHNE builds its abstraction layer based on MLIR and develops a runtime across heterogeneous devices from scratch.

4 CONCLUSION Over the last two decades, cloud vendors have used three principles to keep running demanding data systems cost-effective: resource disaggregation, domain-specific computing, and serverless computing. Since each was proposed to solve a different problem, these principles fail to work harmoniously and oftentimes result in conflicts in practical deployments. This paper proposes a distributed runtime to allow them to work in concert. The distributed runtime will be the narrow waist between data systems and the datacenter infrastructure. It achieves the separation of concerns at scale: users apply domain-specific declarative computation on data, oblivious of data location, concurrency, consistency, disaggregation style, or even the hardware used to do the compute. Indeed, the envisioned distributed runtime and our implementation Skadi is an overly ambitious take. We’ve only scratched the surface in this paper and still have a long way to achieve our ideals.

ACKNOWLEDGEMENT We thank anonymous reviewers for their valuable feedback. We also thank Xiaoyang Deng, Rongfeng He, Zeke Wang, and Xiaosong Lei for their insightful discussions. This work is supported in part by the National Key Research and Development Plan of China (No. 2022YFB4500400), the Strategic Priority Research Program of Chinese Academy of Sciences under grant number XDA0320000 and XDA0320300, the National Natural Science Foundation of China (Grant No. 62090022 and 62172388), and Youth Innovation Promotion Association of Chinese Academy of Sciences (2020105).
REFERENCES


Skadi: Building a Distributed Runtime for Data Systems in Disaggregated Data Centers


Shan, Y. Distributing and Disaggregating Hardware Resources in Data Centers. University of California, San Diego, 2022.


