A Survey and Classification of Software-Defined Storage Systems

RICARDO MACEDO, JOÃO PAULO, and JOSÉ PEREIRA, INESC TEC & University of Minho
ALYSSON BESSANI, LASIGE & Faculdade de Ciências da Universidade de Lisboa

The exponential growth of digital information is imposing increasing scale and efficiency demands on modern storage infrastructures. As infrastructures complexity increases, so does the difficulty in ensuring quality of service, maintainability, and resource fairness, raising unprecedented performance, scalability, and programmability challenges. Software-Defined Storage (SDS) addresses these challenges by cleanly disentangling control and data flows, easing management and improving control functionality of conventional storage systems. Despite its momentum in the research community, many aspects of the paradigm are still unclear, undefined, and unexplored, leading to misunderstandings that hamper the research and development of novel SDS technologies. In this article, we present an in-depth study of SDS systems, providing a thorough description and categorization of each plane of functionality. Further, we propose a taxonomy and classification of existing SDS solutions accordingly several different criteria. Finally, we provide key insights about the paradigm and discuss potential future research directions for the field.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Information systems** → **Storage architectures**; • **Computer systems organization** → *Distributed architectures.*

Additional Key Words and Phrases: Software-defined storage, distributed storage, storage infrastructures.

ACM Reference Format:
Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. 2019. A Survey and Classification of Software-Defined Storage Systems. *ACM Comput. Surv.* 9, 4, Article 39 (May 2019), 35 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Massive amounts of digital data served by a number of different sources are generated, processed, and stored every day in both public and private storage infrastructures. Recent reports predict that by 2025 the *global datasphere* will grow to 163 Zettabytes (ZiB), representing a tenfold increase from the data generated in 2016 [84]. Efficient large-scale storage systems will be essential for handling this proliferation of data, which must be persisted for future processing and backup purposes [110]. However, efficiently storing such deluge of data is a complex and resource-demanding task that raises unprecedented performance, scalability, reliability, and programmability challenges.

First, as the complexity of data center infrastructures increases, so does the difficulty in ensuring end-to-end quality of service (QoS), maintainability, and flexibility. Today's data centers are vertically designed and comprehend several layers along the I/O path providing compute, network, and

Authors' addresses: Ricardo Macedo, ricardo.g.macedo@inesctec.pt; João Paulo, joao.t.paulo@inesctec.pt; José Pereira, jop@di.uminho.pt, INESC TEC & University of Minho, Departamento de Informática, Universidade do Minho, Campus de Gualtar. 4710-057 Braga, Portugal; Alysson Bessani, anbessani@fc.ul.pt, LASIGE & Faculdade de Ciências da Universidade de Lisboa, Campo Grande, 1749-016 Lisboa, Portugal.

0360-0300/2019/5-ART39 $15.00
https://doi.org/0000001.0000001

storage functionalities, including operating systems (OSes), hypervisors, distributed storage, caches, I/O schedulers, file systems, and device drivers [105]. Each of these layers includes a predetermined set of services (*e.g.,* caching, queueing, data management) with strict interfaces and isolated procedures to employ over I/O requests, leveraging a complex, limited, and coarse-grained treatment of the I/O flow. Moreover, data-centric operations such as routing, processing, and management are, in most cases, blended as a single monolithic block, turning infeasible the ability to enforce end-to-end policies (*e.g.,* bandwidth aggregation, I/O prioritization), imposing major scalability, flexibility, and modularity limitations to storage infrastructures [102].

Second, efficiently managing system resources in multi-tenant environments becomes progressively harder as service demand increases, since not only fine-grained resources within a process are shared but also resources across multiple processes and nodes along the I/O path (*e.g.,* shared memory, storage devices, schedulers, caches, network) [63]. Further, since tenants comprehend different service requirements and workload profiles, traditional resource management mechanisms fall short to ensure performance isolation and resource fairness, due to their rigid and coarse-grained I/O management [63]. As a result, achieving QoS under multi-tenancy is infeasible while differentiated treatment of I/O flow, global knowledge of system resources, and end-to-end control and coordination of the infrastructure are not ensured [103, 125].

Third, storage infrastructures have become highly heterogeneous, and are used simultaneously by a myriad of applications with significant different behaviors and evolving requirements that fluctuate over time [21]. However, these infrastructures are frequently tuned with monolithic configuration setups that prevent the online tuning of the storage system [3]. As a result, these homogeneous setups have lead to applications running on a general-purpose I/O stack, competing for the system resources in a non-optimal fashion, and incapable of performing I/O differentiation and end-to-end system optimization [30, 102].

These pitfalls are inherent to the design of traditional large-scale storage infrastructures (*e.g.,* cloud computing, high-performance computing – HPC), and reflect the *absence of a true programmable I/O stack* and the *uncoordinated control of the distributed infrastructure* [30]. As the length of the I/O path increases, it becomes harder to efficiently control and maintain the infrastructure stack. Moreover, individually fine-tuning and optimizing each layer of the I/O stack (*i.e.,* in a non-holistic fashion) of a large-scale infrastructure increases the difficulty to scale to new levels of performance, concurrency, fairness, and resource capacity. Such outcomes result in lack of coordination and performance isolation, weak programmability and customization, strict configuration and adaptability, and waste of shared system resources.

To overcome the shortcomings of traditional storage infrastructures, the *Software-Defined Storage (SDS)* paradigm emerged as a compelling solution to ease data and configuration management, while improving end-to-end control functionality of conventional storage systems [105]. By decoupling the control and the data flows into two major components — *control* and *data planes* — it ensures improved modularity of the storage stack, enables dynamic end-to-end policy enforcement, and introduces differentiated I/O treatment on multi-tenant environments. SDS inherits legacy concepts from Software-Defined Networking (SDN) [54] and applies them to storage-oriented environments, bringing new insights to storage infrastructures, such as improved system programmability and extensibility [80, 95]; fine-grained resource orchestration (under single- and multi-tenancy) [63, 103]; end-to-end QoS, maintainability, and flexibility [46, 105]; and resource efficiency [68, 70]. Furthermore, by breaking the vertical alignment of conventional storage designs, SDS systems provide holistic orchestration of heterogeneous infrastructures, ensure system-wide visibility of storage components, and enable straightforward enforcement of several storage objectives.

In recent years, the SDS paradigm has gained significant traction in the research community, leading to a wide spectrum of both academic and commercial proposals to address the drawbacks

of traditional storage infrastructures. Despite this momentum, many aspects of the paradigm are still unclear, undefined, and unexplored, which may lead to an ambiguous conceptualization of the paradigm and a disparate formalization between current and forthcoming solutions. Essential aspects of SDS such as design principles and main challenges, as well as ambiguities of the field demand detailed clarification and standardization. There is, however, no comprehensive survey providing a complete view of the SDS paradigm. The closest related work [43], provides insightful details about the Software-Defined Cloud paradigm, and surveys other software-defined genres, such as networking, storage, systems, and security. However, this study investigates the possibility of a software-defined environment to handle the complexities of cloud computing systems, providing only a superficial view of each paradigm, not addressing the internals of each plane of functionality, nor existing limitations and open questions.

In this article, we present the first comprehensive literature survey of SDS, explaining and clarifying fundamental aspects of the field. We provide a thorough description of each plane of functionality, and survey and classify existing SDS technologies in both academia and industry regarding storage infrastructure type, namely cloud computing, HPC, and application-specific storage stacks. We define application-specific infrastructures as storage stacks built from the ground up, designed for specialized storage and processing purposes. While some of these stacks can be seen as a subfield of cloud infrastructures, for the purpose of this article and to provide a more granular classification of SDS systems, we classify these systems in a separate category. In more detail, this article provides the following contributions:

- ***Describe an abstract SDS architecture and identify its main design principles.*** The work presented in this article goes beyond reviewing existing literature and categorizes the SDS planes of functionality, regarding their designs. We surveyed existing work on SDS and distilled the key design concepts for both SDS controllers and data planes stages, where we consider most solutions fit.

- ***Propose a taxonomy and classification for SDS systems.*** We propose a taxonomy and classification of existing SDS solutions in order to organize the manifold approaches, bringing significant research directions into focus. Solutions are classified and analyzed based on the storage infrastructure – cloud, HPC, and application-specific – as well as other essences of the field.

- ***Draw lessons from and outline future directions for SDS research.*** We provide key insights about this survey and investigate the open research challenges for this field.

This survey is focused only on the SDS paradigm. Namely, we do not address the design nor limitations of either specialized storage systems (*e.g.,* file systems, block devices, object stores) or other fields of storage research (*e.g.,* deduplication [78], confidentiality [22], metadata management [100], device failures [91], non-volatile memory [49]). Autonomic computing systems are also out of the scope of this article [37]. Furthermore, even though other software-defined genres share similar design principles, they are out of the scope of this article. Such genres include but are not limited to networking [6, 54], operating systems [8, 79], data center [2, 88], cloud [43], key value stores [4, 51], flash [77, 93], security [53, 108], and Internet of Things (IoT) [9, 42].

The remainder of this article is structured as follows. We first present the fundamentals of the SDS paradigm (§2), by depicting the design principles and major characteristics of the field, and introduce a classification for SDS systems. §3 surveys existing SDS systems grouped by the storage infrastructure, namely cloud, HPC, or application-specific storage stacks. In §4, we discuss the current research focus of the paradigm, and investigate the open challenges and future directions of SDS systems. §5 presents the final remarks of the survey.

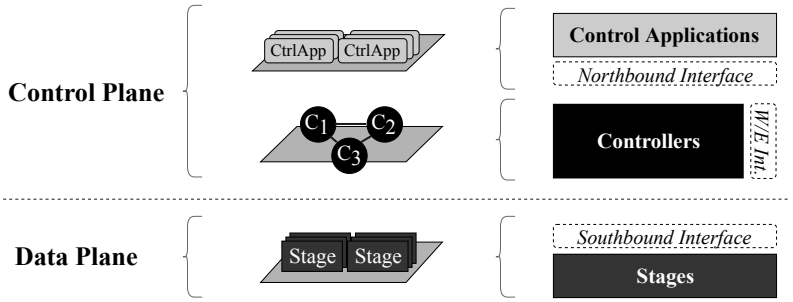

Fig. 1. Layered view of the SDS planes of functionality, comprehending both control and data tiers, as well as the corresponding communication interfaces.

2 SOFTWARE-DEFINED STORAGE

Software-Defined Storage is an emerging storage paradigm that breaks the vertical alignment of conventional storage infrastructures, by reorganizing the I/O stack to disentangle the control and data flows into two planes of functionality — *control* and *data*. Such separation of concerns breaks the storage control into tractable pieces, which *(i)* enables end-to-end policy enforcement [46, 63, 105]; *(ii)* grants programmable storage configurations and improved I/O stack flexibility [80, 95]; *(iii)* exposes internal storage services to new abstraction levels, easing application-building [82, 95]; *(iv)* adds support for isolated and dynamic configurations under multi-tenancy [30, 103, 105]; and *(v)* improves resource utilization [46, 63, 70].

Unlike traditional storage solutions, which require designing and implementing individual control tasks at each I/O layer, such as coordination, queueing, metadata management, and per-layer monitoring, SDS brings a general system abstraction where plain control primitives are implemented at the control platform and properly managed by applications built on top. As a result, a single control platform allows to implement a range of control features (*e.g.,* I/O prioritization, data placement strategies, rate limiting) over a wide spectrum of control granularities (*e.g.,* per-user, -tenant, -request) in a variety of contexts (*e.g.,* cloud, HPC, and application-specific storage stacks).

An SDS architecture comprises two planes of functionality. Figures 1 and 2 depict such an architecture, through a layered view of the SDS functionality and a SDS-enabled storage infrastructure, respectively. The control plane is twofold, comprehending the global control building blocks used for designing system-wide control applications. It holds the intelligence of the SDS system, and consists of a logically centralized controller (§2.2) that shares global system visibility and centralized control, and several control applications (§2.3) built on top [30, 46, 102, 103, 105]. Control applications are the entry point of the control environment (Figure 2: $CtrlApp_1$ and $CtrlApp_2$), and the *de facto* way of SDS users (*e.g.,* system designers, administrators) to express different storage policies to be enforced over the storage infrastructure. Policies are a set of rules that declare how the I/O flow is managed, being defined at control applications, disseminated by controllers, and installed at the data plane. For example, to ensure sustained performance, SDS users may define minimum bandwidth guarantees for a particular set of tenants [105], or define request prioritization to ensure X^{th} percentile latency [58]. Since controllers share a centralized view of the infrastructure, control applications resort to centralized algorithms to implement the control logic, which are simpler and less error prone than designing the corresponding decentralized versions [105]. The scope of control applications (and policies enforced by these) is broad, ranging from QoS provisioning and performance-related services [65, 125], and storage management [103, 115], to shared-logging [95], malware scanning [105], and energy efficiency functionalities [70]. These user-defined policies are

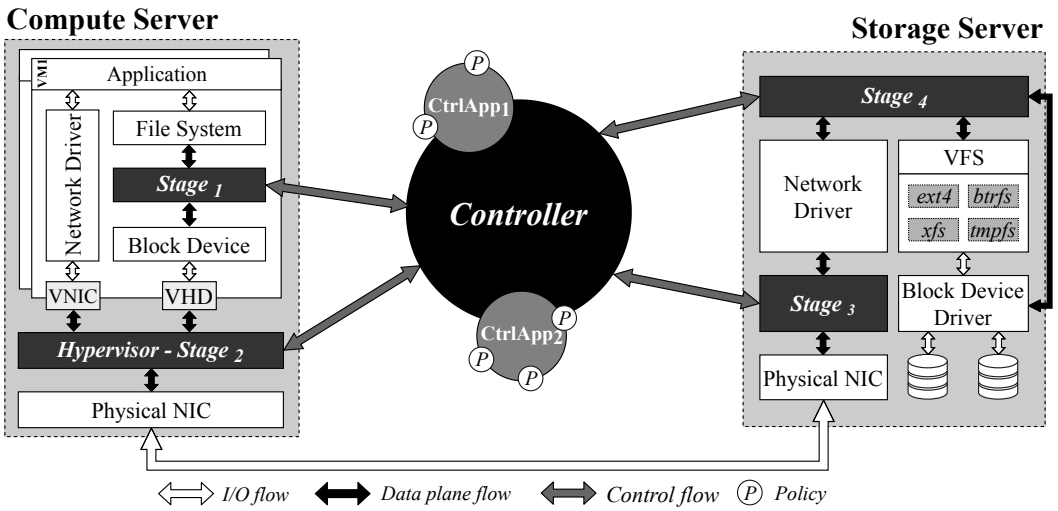

Fig. 2. *SDS*-enabled architecture materialized on top of a general-purpose multi-tenant storage infrastructure. *Compute* servers are virtualized and host virtual machines interconnected to the *hypervisor* by virtual devices, namely *virtual NIC* and *virtual hard disk*. *Storage* servers comprehend general network and storage elements.

shared between control applications and controllers through a well-defined *Northbound interface*, which defines the instruction set of the control tier while abstracting the distributed control environment into a centralized one. To preserve a common terminology between software-defined genres, we adopt the communication interfaces terminology from SDN, namely *Northbound*, *Southbound*, and *Westbound/Eastbound interfaces* [54].

Controllers track the status of the storage infrastructure, including data plane stages, storage devices, and other storage-related resources, and orchestrate the overall storage services present along the I/O stack in a holistic fashion. The control centralization enables an efficient enforcement of policies and simplifies storage configuration [105]. Policies submitted to controllers are handled by a planning engine that translates centralized policies into stage-specific rules and operation logic, which are disseminated to the targeted data plane stages and synchronized with the remainder controllers. Such rules can come directly from control applications or be dynamically generated by automation engines that enable controllers to respond to system variations at runtime [30]. Communication with the data plane is achieved through a *Southbound interface*, illustrated in Figure 2 as a communication channel with gray-toned arrows, that allows controllers to exercise direct control over data plane stages through policy dissemination (*e.g.,* create, update, remove policies) and data plane monitoring (*e.g.,* performance metrics, system statistics) operators. Moreover, a *Westbound/Eastbound interface* (*W/E int.* in Figure 1) establishes the communication protocols and instruction set between controllers to ensure coordination and agreement [30, 102].

The data plane (§2.1) is a programmable multi-stage component distributed along the I/O path (Figure 2: $Stage_1 \ldots Stage_4$), that holds fine-grained storage services dynamically adaptable to the infrastructure status. Each stage can be placed within or horizontally aligned with I/O layers, and respects to a distinct storage service to be employed over intercepted data flows, such as data management (*e.g.,* caching, rate limiting, I/O prioritization), data transformations (*e.g.,* compression, encryption, deduplication), and data routing activities (*e.g.,* flow customization, replica placement). For all intercepted I/O requests, any matching policy will have employed its associated storage

services over the filtered data, which will then be redirected to the original I/O flow (*e.g.,* Figure 2: data flow between *File System* ↔ $Stage_1$ ↔ *Block Device*).

The remainder of this section focus on the specificities of each plane of functionality in a bottom-up fashion, namely data plane stages (§2.1), controllers (§2.2), and control applications (§2.3), regarding their internals, main properties, and designs. Moreover, we introduce a taxonomy for classifying SDS systems based on the design principles and major properties of both control and data planes, which is used as classification criteria for the surveyed work in §3.

2.1 Data Plane

Data plane stages are multi-tiered components, distributed along the I/O path that employ storage-related operations over incoming I/O packets. Stages are the end-point of SDS systems and abstract complex storage services into a seamless design that allows user-defined policies to be enforced over I/O flows. Each of these can be placed throughout the I/O path, establishing flexible enforcement points for the control plane to specify fine-grained control instructions. As presented in Figure 2, stages can be transparently placed between two layers of the I/O stack, acting as a middleware ($Stage_1$, $Stage_3$, and $Stage_4$), or within an individual I/O layer ($Stage_2$). Moreover, stages comprehend *I/O interfaces* to handle I/O flows and a *core* that encompasses the storage primitives to be enforced over such flows. To preserve I/O stack semantics, an *input* and an *output* interfaces marshal and unmarshal data flows to be both enforced at the *core* and correctly forwarded to the next I/O layer. The stage *core* comprises a *(i) policy store*, to orchestrate enforcement rules installed by the control plane (similar to a SDN flow table [67]); an *(ii) I/O filter*, that matches incoming packets with the stage's installed policies; and *(iii)* an *enforcement structure*, that employs the respective storage features over filtered data (a thorough description of this component is presented in §2.1.2).

Albeit overlooked in current SDS literature, the *Southbound interface* is the *de facto* component that defines the separation of concerns in software-defined systems. This interface is the connecting bridge between control and data planes, and establishes the operators that can be used for direct control (*e.g.,* policy propagation, share monitoring information, fine-tune storage services), as well as the communication protocols between them. The control API exhibits to the control plane the instructions a stage understands, in order to configure it to perform local storage decisions (*e.g.,* manage storage policies, fine-tune data plane configurations) [105]. Such an API can be represented in a variety of formats, such as tuples [102] or domain-specific languages [30]. Further, the *Southbound interface* acts as a communication middleware, and defines the communication models between these two planes of functionality (*e.g.,* publish-subscribe [30, 63], RPC [63], REST [30], RDMA-enabled channels [105]). The design of such interface paves the path of interoperability between SDS technologies, *i.e.,* a controller may orchestrate different data plane solutions.

Despite the influence from SDN, the divergence between storage and networking areas has driven the SDS paradigm to comprehend fundamentally different design principles and system properties. First, each research field targets distinct stack components, leading to significantly different policy domains, services to employ over data, and data plane designs. Second, contrarily to the SDN data plane, whose stages are simple networking devices specialized in packet forwarding [54], such as switches, routers, and middleboxes, SDS-enabled stages hold a myriad of storage services, leading to a more comprehensive and complex design. Third, the simplicity of SDN stages eases the placement strategy when introducing new functionalities to be enforced [54]. Differently, SDS stages demand accurate enforcement points, otherwise it may disrupt the SDS environment, and introduce significant performance penalty to the overall storage system.

2.1.1 Properties. The disparity between SDN and SDS data planes has lead both fields to assume significantly different design principles and system properties. We now define the properties that

characterize SDS data tiers, namely programmability, extensibility, stage placement, transparency, and policy scope. The explored properties are contemplated as part of the taxonomy for classifying current SDS solutions (§3).

Programmability. Programmability refers to the ability of a data plane to reuse, extend, and configure existing storage abstractions provided by either specialized storage services or the underlying I/O stack, enabling the development of novel fine-tuned configurable storage services via composition [95]. Programmability in SDS data tiers is usually exploited to ensure I/O differentiation [30, 105], service isolation and customization [46, 102, 103], and development of new storage abstractions [26, 95]. Conventional storage infrastructures are typically tuned with monolithic setups to handle different types of applications with time-varying requirements, leading to all applications experience the same service level [3, 21]. SDS programmability prevents such an end by customizing and fine-tuning the stage *core* to provide differentiation of I/O requests and ensure service isolation (a thorough description of this process is presented in §2.2.2). Moreover, programmable storage systems can ease service development by re-purposing existing abstractions of the storage stack, and exporting the configurability aspect of specialized storage systems to a more generalized environment, *i.e.,* expose runtime configurations (*e.g.,* replication factor, cache size) and existing storage subsystems of specific storage services (*e.g.,* load balancing, durability, metadata management) to a more high-level and accessible environment [94, 95].

Extensibility. Extensibility refers to how easy is for data plane stages to support additional storage services or customize existing ones. An extensible data plane comprises a flexible and general-purpose design suitable for heterogeneous storage environments, and allows for a straightforward implementation of storage services. Such a property is key for achieving a comprehensive SDS environment, capable of attending significantly different requirements of a myriad of applications, as well as to broad the available policy spectrum supported by the SDS system. The extensibility of SDS data tiers strongly relies on the actual implementation of the data plane architecture. In fact, as presented in the SDS literature, highly-extensible data plane implementations are built atop flexible and *extensible by design* storage systems (*e.g.,* FUSE [60], OpenStack Swift [76]). However, behind this flexible and generic data plane design, lies a great deal of storage complexity that if not properly assessed, can introduce significant performance overhead. On another hand, an *inextensible* data plane is typically accompanied with a rigid implementation and hard-wired storage services, tailored for a predefined subset of storage policies. Such a design bears a more straightforward system implementation, fine-tuned for specific storage purposes, and thus comprehending a more strict policy domain only applicable to a limited set of storage scenarios.

Placement. The placement of data plane stages refers to the overall position on the I/O path a stage can be deployed. It defines the control granularity of SDS systems, and is the key enabler to ensure efficient policy enforcement. Each stage is considered as an enforcement point. Less enforcement points lead to a coarse-grained treatment of I/O flows, while more points allow for a fine-grained management of storage functionalities. Since the control plane comprehends system-wide visibility, broadening the enforcement domain allows controllers to accurately determine the most suitable place to enforce a specific storage policy [105]. Improper number and placement of stages may disrupt the control environment, and therefore, introduce significant performance and scalability penalties to the overall storage system.

Depending on the storage context and infrastructure volume, data plane stages are often deployed individually *i.e.,* presenting a single enforcement point to the SDS environment. This *single-point* placement is often associated to local storage environments and tightly coupled to a specific I/O layer or storage component (*e.g.,* file system [80], block layer [68]). Regardless the extensibility and

programmability properties of such data tier, the single enforcement point narrows the available enforcement strategies, which may lead to control inefficiencies and conflicting policies (*e.g.,* enforcing X^{th} percentile latency under throughput-oriented services). Further, a similar placement pattern can be applied in distributed storage settings. Distributed placement of data plane stages (*distributed single-points*) is associated to distributed storage components of an individual I/O layer (*e.g.,* distributed file systems [63], object stores [30, 70]). In this scenario, each enforcement point is a replica of a data plane stage and is deployed at the same I/O layer as the others. In contrast to the prior placement strategy, this design displays more enforcement points for the control plane to decide the enforcement strategies. However, it is still limited to a particular subset of storage components, and may suffer similar drawbacks as the *single-point* approach.

Another placement alternative is *multi-point* data plane stages, which can be placed at several points of the I/O path, regardless the I/O layer [102, 103, 105]. Such a design provides a fine-grained control over data plane stages and is key to achieve end-to-end policy enforcement. However, such a property can introduce significant design complexity in data plane development, and often requires to be directly implemented over I/O layers *i.e.,* following a more intrusive approach.

Transparency. The transparency of a data plane reflects on how seamless is its integration with the I/O stack layers. A *transparent* data plane is often placed between storage components, and preserves the original I/O flow semantics [80]. Yet, such an integration may require substantial marshaling and unmarshaling activities, which directly impacts the absolute latency of I/O operations. Contrarily, an *intrusive* stage implementation is tailored for specific storage contexts and can achieve higher levels of performance since it does not require unnecessary semantic conversions [30, 103, 105]. However, such an approach may entail significant changes to the original codebase, imposing major challenges in developing, deploying, and maintaining such stages, and ultimately reducing its flexibility and portability.

Scope. The policy scope of a data plane categorizes the different storage services and objectives employed over I/O requests. SDS systems can be applied in different storage infrastructures to achieve a number of different purposes, namely for performance [105], security [80], resource utilization [70], and data placement [115] optimizations. To cope with this diversity of objectives, SDS systems comprehend a large array of storage policies categorized in three main scopes, namely *data management*, *data transformations*, and *data routing*. Noticeably, the support for different scopes relies on the data plane implementation, as well as the storage purpose it is being employed.

Data management services are primarily associated to performance-related policies to ensure guaranteed QoS levels [46, 102, 103, 105] (*e.g.,* cache management, rate limiting, bandwidth aggregation, I/O prioritization). *Data transformations* assemble services that alter the original I/O content, such as compression [30], deduplication [70], encryption [30, 80], and erasure coding [80]. *Data routing* primitives encompass routing mechanisms that redefine the data flow, such as I/O path customization [102], replica placement strategies [115], and data staging [40]. Even though mainly applied in networking contexts [54], *data routing* services are now contemplated as another storage primitive, in order to dynamically define the path and destination of an I/O flow at runtime [102].

2.1.2 Stage design. We now categorize data plane stages in three main designs, regarding their internals (*enforcement structure*) and the organization of storage services. Figure 3 illustrates such designs, namely *(a) Stackable*, *(b) Queue-*, and *(c) Storlet-based* data plane stages. Similarly to §2.1.1, these designs are contemplated as part of the taxonomy for classifying SDS data elements.

Stackable stages. *Stackable* or *stack-based* data plane stages provide an interoperable layer design, where layers correspond to specific storage features and are *stacked* according stage's installed policies [80]. Such stacking mechanism, enables an independent and straightforward layer

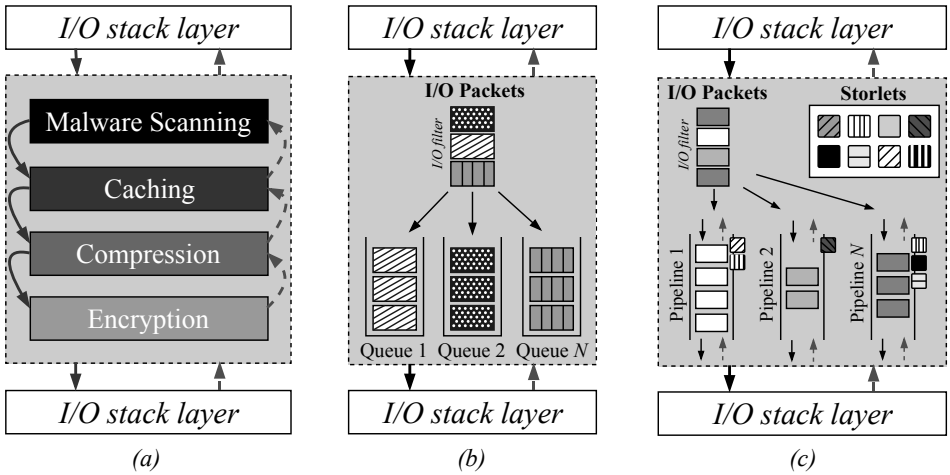

Fig. 3. SDS data plane stage architectures, namely *(a)* Stackable, *(b)* Queue-, and *(c)* Storlet-based designs.

development, introducing a modular and programmable storage environment. Layer organization is established by the control tier, and may result in a number of *stacking configurations* fine-tuned to satisfy the installed storage policies and attend I/O stack requirements. Figure 3-*(a)* depicts an abstract design of a *stackable* data plane stage. It comprehends a four-layer stacking configuration, each with a specific storage feature to be employed over I/O flows, namely *malware scanning*, *caching*, *compression*, and *encryption*. The data flow follows a pass-through layout, ensuring that all packets traverse all layers orderly, such that each layer only receives packets from the layer immediately on top, and only issues packets to the layer immediately bellow. This sequential organization simplifies the architecture design and eases metadata management.

Stacking flexibility is key to efficiently reuse layers and to adapt for different storage environments (*e.g.,* volatile workloads, heterogeneous hardware) [80]. However, this vertical alignment may limit the ability to enforce specific storage policies (*e.g.,* data routing), hampering the available policy spectrum exposed to the control plane. Current stackable data plane solutions are attached to specialized I/O layers, and usually deployed at lower levels of the I/O stack (such as file systems and block devices) closer to storage drives [70, 80].

Queue-based stages. *Queue-based* data plane stages provide a multi-queue storage environment, where queues respect to distinct storage functionalities to apply over I/O packets [105]. Each queue is a programmable storage element that comprehends a set of rules (derived from policies) to regulate traffic differentiation and define its storage properties. Such properties govern how fast queues are served, employ specific actions over data, and forward I/O packets to other point of the I/O path. Figure 3-*(b)* depicts the design of a *queue-based* data plane stage. Incoming packets are inspected, filtered, and assigned to the corresponding queue ($Queue_1 \ldots Queue_N$).

The flexible design of the queuing mechanism allows for simplified queue orchestration, and improved flexibility and modularity of data plane stages [105]. Although, as demonstrated by complementary research fields [32, 111], queue structures are essentially used to serve data management policies, such as performance isolation and QoS provisioning. This management-oriented flexibility trades stage design's customization, turning the integration and extension of alternative storage services, without conflicting with concurrent storage objectives, a challenging endeavor.

Storlet-based stages. *Storlet-based* data plane stages abstract storage functionalities into programmable storage objects (*storlets*) [30, 95]. Leveraging from the principles of active storage [87] and *OpenStack Swift Storlets* [76], a *storlet* is a piece of programming logic that can be injected into the data plane stage to perform custom storage services over incoming I/O requests. This design promotes a flexible and straightforward storage development, and improves the modularity and programmability of data plane stages, fostering the reutilization of existing programmable objects. Figure 3-*(c)* depicts the abstract architecture of a *storlet-based* data plane stage. Stages comprehend a set of pre-installed storlets that comply with initial storage policies. Policies and storlets are kept up-to-date to ensure consistent actions over I/O packets. Moreover, the data plane stage forms several storlet-made pipelines to efficiently enforce storage services over data flows. At runtime, I/O flows are intercepted, filtered, classified, and redirected to the respective pipeline.

The seamless development of programmable storage objects ensures a programmable, extensible, and reusable SDS environment. However, as the policy sphere increases, it becomes harder to efficiently manage the storlets mechanisms at data plane stages. Such an increase may introduce significant complexity in data plane organization and lead to performance penalties on pipeline construction, pipeline forwarding, and metadata and storlet management.

2.2 Control Plane — Controllers

Similarly to SDN, the SDS control plane provides a logically centralized controller with system-wide visibility that orchestrates a number of data plane instances. By sharing a unified control point with control applications, it eases both application-building and control logic development. However, even though identical in principles and system properties, the divergence between SDN and SDS research objectives may impact the entailed complexity on designing and implementing production-quality SDS systems. First, the introduction of a new storage functionality to employ over I/O flows cannot be arbitrarily assigned to a data plane stage, since it may introduce significant performance penalties, and even compromise the enforcement of other policies. For instance, SDN data planes are mainly composed with simple forwarding services, while SDS data planes may comprehend performance-oriented functionalities (*e.g.,* rate limiting, tail latency), which are sensitive to I/O processing and propagation along the I/O path, but also data transformations that entail additional computation, directly impacting the overall processing and propagation time of I/O flows. Thus, the controller requires to perform extra computations to ensure the efficient placement of storage features, preventing policy conflicts, and ensuring a correct execution of the SDS environment. Second, since the domain of both storage features and policy scope is broader than in SDN ensuring a transparent control logic and straightforward policy specification, introduces increased complexity to the design of SDS controllers (*e.g.,* decision making, storage service placement).

As depicted in Figure 1, controllers are placed at the mid-tier of the SDS logical view, and are the main building blocks of the SDS environment, which orchestrate a number of data plane stages according the actions of control applications built on top. Despite distributed, the control plane shares the control logic through a logically centralized controller that comprehends system-wide visibility, eases the design and development of general-purpose control applications, provides a simpler and less error-prone development of control algorithms, ensures an efficient distribution and enforcement of policies, and fine-tunes SDS storage artifacts in a holistic fashion (*i.e.,* encompassing the global storage environment) [30, 40, 102, 105, 115]. Unless otherwise stated, a *controller* defines a logically centralized component, even though physically distributed. To ensure such an end, the controller can be partitioned in three functional modules, namely a *metadata store*, a *system monitor*, and a *planning engine* (as illustrated in Figure 4-*(a)*). Each of these modules comprehends a particular set of control features shared between controllers or designed for a specific control

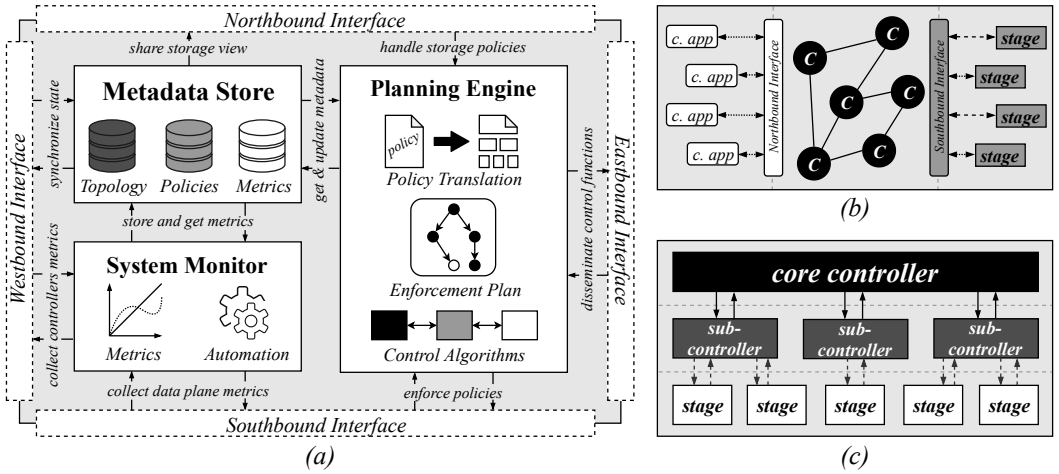

Fig. 4. SDS controllers architecture. *(a)* presents the organization of the internals of a SDS controller. *(b)* and *(c)* depict the design of SDS control plane controllers, namely *Flat* and *Hierarchical* designs.

device. Moreover, these modules are programmable and allow SDS users to install, at runtime, custom control actions to employ over the SDS system (*e.g.,* control algorithms for accurate policy enforcement, collection of monitoring metrics).

The *metadata store* comprehends the essential metadata of the storage environment and ensures a synchronized view of the storage infrastructure. As depicted in Figure 4-*(a)*, different types of metadata are stored in separate metadata instances, namely *topology*, *policies*, or *metrics*. The *topology* instance maintains a stage-level topology graph that comprehends the distribution of the SDS data plane stages, along the assigned storage services and information about the resources of each node a stage is deployed (*e.g.,* available storage space, CPU usage). The *policies* instance holds the policies submitted by control applications, as well as the ones installed at data plane stages. The *metrics* instance persists digested monitoring metrics and statistics of both control and data flows, to assist other modules define the enforcement plan and control logic. Further, to ensure a dependable SDS environment, metadata is usually synchronized among controllers [102, 105].

The *system monitor* collects, aggregates, and transforms unstructured storage metrics and statistics into general and valuable monitoring data [30, 36, 115]. It captures relevant properties of the physical storage environment, including both static and dynamic metrics (*e.g.,* maximum disk and network switch performance, available storage space, IOPS, bandwidth usage) from SDS controllers and data plane tiers, and collects samples of I/O workloads, in order to trace an up-to-date profile of the storage stack. Such metrics are then analyzed and correlated, bringing significant insights about the system status, allowing the controller to optimize the infrastructure by (re)configuring the SDS storage services (deployed at data plane stages), and assist other modules in policy enforcement and feature placement activities. Furthermore, the *system monitor* can also exercise automation operations without input of control applications, ranging from simple management activities (*e.g.,* increase or decrease number of controllers), to more complex tasks (*e.g.,* fine-tune storage policies, reconfigure data plane stages) [30]. Such mechanism is key to enable a self-adaptable SDS system, capable of attending the requirements of heterogeneous storage environments.

The *planning engine* implements the control logic responsible for policy translation, policy enforcement, and data plane configuration. Policies submitted by control applications are parsed, validated, and translated into stage-specific rules, that will be installed at the respective data plane

stage. Policy enforcement is achieved through control algorithms that specify how the data plane handles the I/O flow and define the most suitable place for policies to be enforced [105]. Such control algorithms can be defined *a priori* or installed at runtime by control applications. Both translation and enforcement operations may lead the controller to interact with a single data plane stage (*e.g.,* to install a particular rule), or include a number of stages to perform distributed enforcement (*e.g.,* bandwidth aggregation) [103, 105]. Leveraging from the deliverables of other modules, the *planning engine* fine-tunes data plane stages in a holistic fashion to efficiently attend the time-varying requirements of the storage infrastructure.

To provide a seamless integration with the remainder SDS layers, the controller connects to a *northbound* and a *southbound interfaces* to interact with control applications and data plane stages, respectively. Similar to other software-defined genres, SDS architectures comprehend a network of controllers connected through a *westbound/eastbound interface* (as illustrated in Figures 1 and 4-*(a)*). Such an interface establishes the instruction set and implements the communication protocols between SDS controllers, being used to exchange data; ensure state replication, synchronization, and fault tolerance; monitoring; and, depending on the control plane architecture, submit control operations and assign control tasks to other controllers (a thorough description of such property is presented in §2.2.2). Further, this interface aims at achieving interoperability between different controllers [54], however, despite its clear position in the SDS environment, current literature does not explore nor details about such an interface.

The depicted modules present the fundamental organization of SDS controllers. Depending on the design and storage context, additional modules such as *discovery components* [105], *schedulers* [115], and *estimators* [125] may be included on such architecture.

2.2.1 Properties. Similarly to other software-defined genres, designing and implementing production-quality SDS systems requires solving important challenges at the control plane [52]. We now define the properties that characterize SDS controllers, namely scalability, dependability, and adaptability, which are also contemplated as part of the taxonomy for classifying SDS systems (§3).

Scalability. Scalability refers to the ability of a control plane to efficiently orchestrate and monitor a number of data plane stages. Analogously to other software-defined fields, the control plane can either be physically centralized or distributed [6]. A physically centralized control plane consists of a single SDS controller to orchestrate the overall storage infrastructure, which is an attractive design choice in terms of simplicity [74, 98, 107]. However, control centralization of SDS ecosystems imposes severe scalability, performance, and dependability requirements that are likely to exhaust and saturate underlying system resources, largely dictating the end performance of the storage environment. As the amount of data planes stages increases so does the control traffic destined towards the centralized controller, bounding the overall system performance to the processing power of this single control unit. Hence, despite the obvious limitations in scale and reliability, such a design may be only suitable to orchestrate small-to-medium storage infrastructures [54].

Production-grade SDS controllers must be designed to attend the scalability, performance and dependability requirements of today's production storage systems, meaning that any scaling limitations should be inherent to the storage infrastructure and not from the actual SDS implementation. As such, physically distributed controllers can be scaled up to attend the requirements of large-scale storage infrastructures. While sharing a logically centralized service, multiple interconnected SDS controllers orchestrate the overall storage infrastructure by sharing control responsibility, and thus alleviating the control load imposed on the centralized controller. Leveraging from existing control classifications of other software-defined genres, distributed SDS controllers can be organized as a *flat* or a *hierarchical* distribution (§2.2.2) [6]. Flat designs (Figure 4-*(b)*) imply the horizontal

partitioning of the storage environment to provide a replicated control service, forming for a reliable, fault-tolerant, highly-available cluster of controllers [30, 63, 103, 105, 115]. On the other hand, hierarchy-based designs (Figure 4-*(c)*) imply the vertical partitioning of the storage environment to provide a scalable and highly-performing SDS control plane [40, 46, 102].

Dependability. Dependability refers to the ability of a control plane to ensure sustained availability, resiliency, and fault tolerance of the control service. Physically centralized controllers represent a single point of failure (SPOF) to the SDS environment, leading to the unavailability of the control service upon a failure. As a result, the SDS ecosystem becomes unsupervised, incapable of regulating incoming storage policies and orchestrate the data plane tier, consequently impacting the SDS enforcement capabilities and the overall infrastructure's performance. The system should handle failures gracefully, avoiding SPOF and enabling fault tolerance mechanisms. Thus, similarly to SDN [10, 17, 48, 52], physically distributed SDS solutions provide coordination facilities for detecting and recovering from control instance failures [30, 102, 115]. In this model, controllers are added to the system to form a replicated, fault-tolerant, and highly-available SDS environment. Moreover, existing distributed controllers consider the different trade-offs of performance, scalability, and state consistency (strong or eventual), and provide distinct mechanisms to meet fault-tolerance and reliability requirements. For instance, controllers may assume a clustered format to achieve fault-tolerance through active or passive replication strategies by resorting to Replicated State Machines built with Paxos-like techniques [55, 75], or simply implement a primary-backup approach where one main controller orchestrates all data plane elements while the remainder control instances are used for replication of the control service [16]. Further, while some solutions may comprehend a strong consistency model to ensure correctness and robustness of the control service, others may resort to relaxed models, where each controller is assigned to a subset of storage domain and holds a different view of the storage infrastructure. Regarding control distribution, flat control planes are designed to ensure sustained resilience and availability of the SDS environment [30, 105], while hierarchical control planes focus on the scalability challenges of the SDS environment [40, 102].

Adaptability. Adaptability refers to the ability of a control plane to respond, adapt, and fine-tune enforcement decisions under time-varying requirements of the storage infrastructure. The high demand for virtualized services has driven data centers to become extremely heterogeneous, leading storage components and data plane elements to experience volatile workloads, subject of workload bursts, diverse storage requests, and skewed access patterns [3, 21, 105]. Moreover, designing heterogeneity-oblivious SDS systems, with monolithic deployments and homogeneous configurations, can severely impact the global storage ecosystem, hindering the ability to accurately enforce storage policies throughout the I/O path [30]. Therefore, SDS controllers must comprehend a self-adaptive design, capable of dynamically adjust their storage artifacts (*e.g.,* policy values, data plane stage configurations) to the surrounding environment, in order to deliver responsive, accurate, and efficient enforcement decisions upon such evolving requirements [30]. While heuristic-based mechanisms are commonly used to dynamically adjust storage artifacts, existing literature resorts to storage automation mechanisms such as linear programming [70, 125] and machine learning (ML) techniques [46] to provide more accurate enforcement strategies and fine-grained control over both storage policies and data plane stage configurations. First, enforcement strategies directly impact I/O flows, dictating the overall performance and efficiency of SDS systems. Thus, by employing self-adaptive and autonomous mechanisms over controllers, an accurate, dynamic, and efficient enforcement service can be guaranteed. Second, due to the fast changing requirements of the storage environment, data plane configurations rapidly become subpar. Thus, automated optimization of data plane resources is key to ensure efficient policy enforcement and optimal resource usage.

Complementary properties, such as *generality*, *simplicity*, and *performance*, also portray interesting aspects of SDS control planes [52, 93, 105]. Moreover, controllers should provide a generic, simplified, extensible, and comprehensive control environment, broadening the spectrum of storage functionalities, simplifying application-building, and easing control service extension. At the same time, the control plane must not stall the overall system's performance. Data plane orchestration, state synchronization, enforcement planning, and policy handling should cause minimal performance degradation [102, 105].

2.2.2 Controller distribution. SDS literature classifies the distribution and design of controllers as logically centralized, despite physically distributed for the obvious reasons of scale and resilience [102, 105]. Though, beyond the design and implementation of these distributed elements lies a great deal of practical complexity. In this work, we categorize distributed control planes regarding controller organization. Figure 4 illustrates such designs, namely *(b) Flat* and *(c) Hierarchical* control planes. Similarly to the properties presented in §2.2.1, both designs are contemplated as part of the taxonomy for classifying SDS control elements.

Flat. Flat control planes provide a horizontally partitioned control environment, where a set of interconnected controllers act as a coordinated group to ensure a reliable and highly-available control service while preserving logical control centralization. In this model, depending on the control plane's implementation, controllers may hold different organizations, being designed accounting with the different trade-offs of performance and resiliency. For instance, some implementations may provide a cluster-like distribution, where a single controller orchestrates the overall storage domain, while others are used as backups that can take over in case the primary fails. In this scenario, the centralized controller orchestrates the global data plane environment, which involves handling all stage-related events (*e.g.,* collect reports and metrics), disseminate policies, generate comprehensive enforcement plans, and enforce policies. At the same time, the control plane provides the coordination facilities to ensure fault-tolerance and strong consistency by relying on Paxos-based mechanisms [102, 115] or simple primary-backup strategies [16]. Such a design vests distributed controllers with strong consistency properties, ensures high-availability of the control service, and eases control responsibility. However, since control remains centralized, this cluster-based distribution falls short when it comes to scalability, limiting its applicability to small-to-medium sized storage infrastructures [105].

Other solutions may provide a network-like flat control platform, where each controller is responsible for a subset of the data plane elements [30, 52]. In such a case, by partitioning control responsibility, each controller orchestrates a different part of the infrastructure, synchronizing its state with strong or eventual consistency mechanisms with remainder control peers. Upon the failure of a controller, another may assume its responsibilities until it becomes available. Such a design ensures an efficient and high-performance control service, and provides a flexible consistency model that allows the SDS system to scale to larger environments than cluster-based approaches. However, the shared control responsibilities of this network-like model hardens the control plane's ability to share a logical centralized setup with control applications, along continuous up-to-date system-wide visibility of the storage environment, hindering its applicability to large-scale production storage infrastructures. Further, with the emergence of novel computing paradigms composed by thousands of nodes, such a serverless cloud computing [45] and Exascale computing [24], this design may hold severe scalability and performance constraints.

Hierarchical. The constant dissemination of stage-related events, such as control enforcement and metrics collection, hinders the scalability of the control plane [46, 102]. To taper the imposed load from the centralized control, both control and management flows must be handled closer

to data plane resources, and minimized as possible without compromising system correctness. Thus, similarly to distributed SDN controllers [35, 41, 120], hierarchical control plane distributions address such a problem by organizing SDS controllers in a hierarchical disposition [30, 40, 46, 102]. Despite distributed across the storage infrastructure, controllers are hierarchically ranked and grouped by control levels, each of them respecting to a cumulative set of control services. Such an approach distributes control responsibility to alleviate the load imposed over centralized services, thus enabling a more scalable SDS control environment.

As depicted in Figure 4-*(c)*, the control plane is vertically partitioned, and distinguishes its elements between *core controllers* and *sub-controllers*. *Core controllers* are placed at the top-tier of the hierarchy, and comprehend overall control power and system-wide visibility. While maintaining a synchronized state of the overall SDS environment, *core controllers* manage control applications, and orchestrate both data plane and *sub-controller* elements. Moreover, *core controllers* share part of their control responsibilities with underlying control tiers, propagating the control fragments in a hierarchical fashion. *Sub-controllers* form the lower-tiers of the control hierarchy, placed closer to data sources, and comprehend a subset of control services. Each of these controllers manages a segment of the SDS data plane, and is used manage control activities that do not require global knowledge nor impact the overall state of the control environment. Since *core controllers* comprehend global visibility, they perform accurate and holistic control decisions over the SDS environment. However, maintaining a consistent view of the system is costly, causing significant performance overhead even when performing simple and local decisions [40, 102]. On the other hand, *sub-controllers* are tailored for control-specific operations, thus performing faster and fine-grained local decisions over data plane stages. In case a *sub-controller* cannot perform certain control actions over its elements, it passes such responsibility to higher ranking controllers.

Communication between control instances is achieved through the *westbound/eastbound interface*, and is used for establishing the control power and policy dissemination, periodic state propagation for synchronization, and health monitoring events.

2.3 Control Plane — Control Applications

Control applications are the entry point of the SDS environment and the *de facto* way of expressing the management requirements and control directives of I/O flows. Such applications exercise direct control over SDS controllers by defining their control logic through storage policies and control algorithms, which are ultimately translated into fine-grained stage-specific rules to be employed over I/O packets. Similar to other software-defined genres [43, 54], control applications introduce a *specification* abstraction into the SDS environment, allowing to express the desired storage behavior without being responsible for implementing the behavior itself. Moreover, the logical centralization of control services allows control applications to leverage from the same control base, leading to a correct, consistent, and efficient policy creation.

Existing control applications are designed for a variety of storage contexts, and cover a wide array of control functionalities. For instance, existing literature proposes *performance-oriented control applications*, addressing minimum performance guarantees, I/O prioritization, bandwidth aggregation, rate limiting, cache management, and tail latency [30, 46, 63, 65, 102, 103, 105, 124, 125]; *data placement applications* such as replica placement, I/O path customization, load balancing, and data staging [40, 94, 95, 102, 115]; *data transformation applications* such as encryption, deduplication, compression, and erasure coding [30, 80]; and also explore other storage aspects, such as reducing energy consumption, malware scanning, and distributed logging [70, 95, 105].

Analogously to the *Southbound interface*, the *Northbound interface* is the connecting bridge between control applications and controllers. It abstracts the distributed control environment into a language-specific communication interface, hiding unnecessary infrastructure details, while

allowing a straightforward application-building and policy specification. Moreover, such design fosters the integration and reutilization of different control applications between SDS technologies, enabling an interoperable control design. However, existing SDS proposals define their own *Northbound interface*, each with its own specific definitions, thus hindering the ability to combine different control applications throughout distinct storage contexts and SDS technologies.

3 SURVEY BY INFRASTRUCTURE

This section presents an overview of existing SDS systems, grouped by storage infrastructure type, namely *cloud*, *HPC*, and *application-specific* storage stacks. Moreover, each system is classified according the taxonomies described in §2.1 and §2.2. As each storage infrastructure has its own requirements and restrictions, the design and combination of SDS properties vary significantly with the storage type being targeted. Further, solutions from both academia and industry are considered for this survey.

3.1 Cloud Infrastructures

Cloud computing infrastructures host a myriad of public and private customers with significant different behaviors and evolving requirements that fluctuate over time. Commercial cloud systems such as Google Compute Engine, Amazon EC2, and Microsoft Azure, offer enterprise-grade computing and storage resources as public utilities so customers can deploy and execute general-purpose services in a flexible pay-as-you-go model. Such infrastructures consist of hundreds to thousands of compute and storage servers. Compute servers are virtualized and abstract multiple physical servers into an infinite pool of resources that is exposed through virtual machines (VMs) or containers. Resources are shared between tenants and mediated by hypervisors. Storage servers accommodate different storage systems (such as block devices, file systems, and object stores) and drives (*e.g.,* SATA HDD and SSD, NVMe SSD) with distinct levels of granularity and performance, respectively. Such servers are responsible for persisting all data, and are exposed to VMs as virtual devices. Compute and storage servers are connected through high-speed network links (1-100 Gbps [28]) that carry all infrastructure traffic. However, such virtualized environment has lead cloud infrastructures to comprehend complex and predefined I/O stacks unable to perform I/O differentiation, thus resulting in increased end-to-end complexity and difficulty in enforcing storage requirements and end-to-end control over storage resources [59, 105]. While several systems have been proposed to partially address this problem (*e.g.,* QoS provisioning [32, 34, 111], scheduling [12, 33, 118], elasticity [3, 19, 61]), neither have considered end-to-end enforcement nor holistic orchestration of infrastructure resources. To address such shortcomings, software-defined systems have moved towards cloud-based infrastructures.

The term Software-Defined Storage was first introduced by IOFlow [105]. Specifically, IOFlow is a SDS system that enables end-to-end policy enforcement under multi-tenant infrastructures. It comprehends a full-fledged SDS stack, consisting of a queue-based data plane and a flat control plane. Stages are designed for performance isolation and routing primitives, and are deployable throughout the I/O stack (*e.g.,* hypervisor, drivers), allowing differentiated I/O treatment and policy enforcement from VMs to shared storage, with minimal performance degradation. Even though transparent to VMs and guest applications, ensuring negligible overhead during policy enforcement comes at the cost of reduced extensibility, as turning conventional I/O layers into IOFlow-compliant ones requires significant code changes. Moreover, stages are soft-state. While not impacting system's correctness, the failure of a stage leads to temporary degradation in performance, until the enforcement environment stabilizes. At the control plane, a logically centralized controller discovers IOFlow-compliant stages and dynamically configures their storage services. It maintains an *up-to-date* topology graph with the SDS essentials and periodic trace profiles of the infrastructure's

hardware, in order to validate the I/O performance under variable scenarios and provide accurate, correct, and flexible enforcement rules. Moreover, the controller dynamically adapts and fine-tunes enforcement plans, stage configurations, and storage policies through heuristic-based mechanisms, which is key to ensure a comprehensive control service under heterogeneous and time-varying workloads. Furthermore, a controller's failure does not impact system correctness, but only leads to temporary performance degradation. Interestingly, IOFlow's control applications implement conservative policies in case the controller is unreachable, and perform a best-effort enforcement activity until the controller is available again.

While the original IOFlow design was focused on end-to-end policy enforcement over complex I/O stacks, it was later extended to support two fundamental data center resources, namely caching and networking [102, 103]. Moirai [103] extends IOFlow to exercise direct and coordinated control over the distributed caching infrastructure, improving overall resource utilization, and achieving performance isolation and QoS guarantees. Stages are deployed at the I/O path as stackable and programmable caching instances, that employ management storage features over incoming I/O requests. Since caches are flexible resources deployable throughout the I/O path, Moirai is able to perform end-to-end policy enforcement. However, while designed for performance-sensitive and QoS objectives, extending such design to support routing primitives may be infeasible since the I/O path is predetermined and static. Additionally, Moirai adds support for monitoring stages that maintain key performance metrics of each workload running on the system (*e.g.,* throughput, read-write proportion, *hit ratio* curves), which are periodically sent to the controller. At the control plane, a logically centralized controller, built on top IOFlow's traffic classification mechanism, orchestrates data plane stages in a holistic fashion, and maintains (cache) consistency and coherence across the I/O stack. By continuously monitoring the SDS infrastructure and leveraging from the collected metrics and statistics, it dynamically fine-tunes existing stages, regarding their installed rules, storage features (*e.g.,* caching policies, priority schemes), and placement.

To override the rigid and predefined I/O path, sRoute [102] goes towards combining storage and networking primitives. By extending IOFlow's architecture, it employs routing primitives to the storage stack and proposes an I/O routing abstraction, turning the storage stack more programmable and dynamic. The data plane is twofold, composed by programmable switches (*sSwitch* stages) deployed throughout the I/O path to provide flow regulation and routing properties, and *sSwitch-enabled* queue-based stages, that implement management features and I/O differentiation. Such stages provide per-I/O and per-flow routing capabilities with strong semantic guarantees, that allow I/O flows to be redirected to any I/O point (*e.g.,* other file destinations, controller, *sSwitch-enabled* stages), and to cope with the manifold QoS objectives. The control plane comprehends a hierarchical distribution, made of a centralized cluster of controllers, and several *control delegates*, which are restricted control plane daemons installed at *sSwitches* for control plane efficiency. The controller uses *delegates* to perform control decisions locally at the *sSwitches*, alleviating the load of the centralized component. Such a design, allows the control plane to scale as many existing *sSwitches* in the system. Moreover, control service is dependable, being replicated for availability with standard Paxos-like schemes [55]. While IOFlow only explored routing primitives between stages, sRoute considers the full I/O routing spectrum, where both I/O path and endpoint can also change, leading to a more comprehensive SDS system.

Similarly, JoiNS [113] holistically orchestrates both storage and network primitives to efficiently enforce performance-sensitive operators over networked storage environments. While similar to sRoute's control plane, its data plane differs by comprehending queue-based network and storage stages. For network stages, JoiNS leverages from existing SDN data planes [67] and programmable switches, to efficiently enforce routing primitives over the storage infrastructure, while storage stages implement predetermined management features over block device drivers. Moreover, despite

distributed, both stages are placed at fixed points of the I/O flow. Specifically, network stages are deployed over network devices, while the storage ones are implemented over storage drivers.

Differently, several systems have been proposed to address specific problems of cloud storage in a SDS-fashion (*e.g.,* performance isolation, tail latency, resource fairness), which are well-suited to be further integrated as a submodule, storage functionality, or control algorithm of full-fledged SDS systems [74, 82, 98, 104, 124, 125]. Pisces [98] introduced system-wide performance isolation and fair resource allocation at the service level in multi-tenant cloud environments, through a physically centralized controller and a twofold data plane setup. Similar to generally known SDS control primitives, the controller translates global storage objectives from different tenants into individual storage rules to employ over data plane stages, while monitoring performance metrics and periodically fine-tuning stages to ensure efficient performance isolation. However, control and data operations are not entirely decoupled, as a number of control activities are performed at data plane elements, thus not ensuring a comprehensive enforcement activity over the data plane. The data plane is composed by a number of request routers that orchestrate the resource sharing activity, and several queue-based stages, each installed at a distinct storage node. Such stages are implemented with built-in management features to provide fairness and performance isolation.

Despite its advantages, Pisces is limited to storage bandwidth, which is simpler to control and manage than tail latency as bandwidth is an average over time not affected by I/O path's cumulative interactions. This led to designs for efficient QoS provisioning, achieving end-to-end tail latency under multi-tenancy. PriorityMeister (PM) [125] is a proactive QoS system that combines prioritization and rate limiting mechanisms over network and storage resources to meet tail latency objectives. Similar to IOFlow, it comprehends a distributed controller that automatically orchestrates QoS-based parameters to be installed at data plane stages. At the data plane, queue-based stages are deployed at both network and storage elements of compute and storage servers, and provide latency differentiation per-workload basis, by analyzing the workload's burstiness and load at each stage. PM stages are independent enforcement modules vested with single-purpose management functionalities, that act upon QoS parameters received from the global controller. However, each of these stages comprehends several rate limiting queues, which may result in a manifold increase in computation time and number of required computing servers. WorkloadCompactor (WC) [124] addresses such a problem by consolidating multiple workloads onto a storage server. Similar to PM's design, WC enforces rate limits and priorities in storage and network to minimize the number of servers that cloud providers use to satisfy all workloads, while meeting its tail latency objectives. Its controller automatically selects rate limits and priority profiles through heuristic-based mechanisms, in order proactively compact workloads while enabling tail latencies at the 99.9th percentile. Unlike PM, data plane stages are deployed along storage devices (*e.g.,* SSD drives).

Cloud providers offer a wide-array of storage services with trade-offs in performance, cost, and durability, which lead to applications built on top to opt for simplicity instead of resorting to different services with conflicting properties. Tiera [82] is a programmable middleware that facilitates the use and specification of multi-tiered cloud storage. It shields applications from the complexity introduced by cloud providers by encapsulating and re-purposing existing storage services of cloud providers into an optimized interface that can be glued to comply with the requirements of cloud applications. Even though controllerless, Tiera comprehends a programmable data plane that separates storage policies from the tier mechanisms through storlet-based object store stages, each of these abstracting a subset of storage services to be employed over I/O requests of a selected application. Stages are transparently accommodated between applications and cloud services, and are built through a configuration file that specifies the storage tiers to use, along their capacities and set of storage services to be employed. However, this design can only extract the storage properties of a single cloud provider, not being able to combine existing features across data centers.

To address this, Wiera [74] extends Tiera's design to provide a control-based geo-distributed cloud storage system, that allows enforcing global storage policies across multi-tired data centers. The data plane is composed by Tiera stages, while the control plane comprehends an adaptable physically centralized controller with system-wide visibility that abstracts the complexities of accessing distinct Tiera-enabled data centers, such as scalability, fault tolerance, consistency, and load balancing. Such a design allows system designers to combine a wide-array of storage features available at different tiers of the cloud storage hierarchy, enabling the creation of new services via composition, thus providing a comprehensive storage environment suited to attend the needs of geo-distributed dependent applications.

Multi-tenant distributed systems composed by hundreds of small services (*e.g.,* Service-Oriented Architectures (SOAs), micro-services) are often used on cloud infrastructures to build large-scale web applications and infrastructure systems [104]. Such systems comprise fine-grained and loosely coupled services, each running inside a physical or virtual machine. However, its limited visibility hinders the ability to provide end-to-end enforcement and QoS provisioning. To address such challenges, Wisp [104] proposes a physically and logically distributed framework for building efficient and programmable SOAs that dynamically adapt storage resources under multi-tenancy. It comprehends a fully decentralized design, composed by a number of controllers and data plane stages. Each SOA service accommodates a SDS tuple, made of a controller and a number of stages. The control plane executes distributed control algorithms (*e.g.,* rate limiting, scheduling). Each controller gathers local information and propagates it with remote controllers, so when locally executing control algorithms it can cope with the required storage objectives. Moreover, while several systems provide per-tenant and per-workload enforcement [105, 125], Wisp coordinates per-request resources, meaning that even though decentralized, it does not trade performance nor decision accuracy. The data plane resorts to the building blocks provided by SOA resources, and abstracts them into queue-based stages able to enforce management-oriented storage features, such as rate limiting and performance isolation.

Other storage features besides QoS provisioning and performance isolation can be also explored on SDS systems. For instance, flexStore [70] investigates the possibility of using SDS to improve energy consumption of cloud-based data centers. Specifically, flexStore provides a storage framework for dynamically adapting a data center to cope with specific QoS and power objectives. The data plane is twofold and employs stackable stages over heterogeneous storage systems (*e.g.,* object stores) and hypervisors. Storage systems stages adjust the data layout of different storage devices, and collect performance and resource usage metrics (*e.g.,* latency, space utilization) to support the enforcement plan of energy-related policies (*e.g.,* data consolidation, reduce amount of storage drives). Hypervisor-based stages apply transformations and management features over multi-tenant I/O requests (*e.g.,* deduplication, adjust replication factor, caching). The control plane follows a flat distribution that enforces QoS and energy-related policies under multi-tenancy. To ensure isolation, the controller manages the life cycle of dedicated storage volumes (*i.e.,* creates and destroys) and allocates them to VMs/tenants. Each of these volumes complies with both tenants' objectives and overall energy requirements. Upon energy variations, flexStore dynamically adapts its energy requirements while complying with the performance guarantees.

Cloud-based SDS systems are currently an actively research topic on the storage field, mainly to improve overall performance, programmability, and resource efficiency of cloud storage stacks. Indeed, several enterprise systems such as VMware Cloud [109], Microsoft Windows Server [69], and NetApp ONTAP Select [71], have been paving the way of the SDS paradigm in industry, fostering its adoption at a global scale. Furthermore, to broad the current SDS environment, researchers have suggested to integrate on full-fledged SDS systems existing storage subsystems directly assessing storage-related resources in the cloud (*e.g.,* QoS provisioning [32, 47, 58, 111], I/O

scheduling [12, 33, 65, 118], consensus [66]), either as a storage functionality, a control algorithm, or a control application. Nevertheless, most of these systems assume a flat control plane and programmable queue-based data planes stages, tailored for management-based storage services. These assumptions may not hold true in HPC infrastructures, which are composed by thousands of nodes designed for specialized computationally intensive tasks.

3.2 High-Performance Computing Infrastructures

HPC infrastructures are composed by thousands of nodes capable of generating hundreds of PFLOPS (10^{15} floating point operations per second) at peak performance [106]. Indeed, supercomputers are the cornerstone of scientific computing and the *de facto* premises for running compute-intensive applications in several research fields (*e.g.,* aeronautics, quantum mechanics, weather forecasting, earth sciences), where large-scale simulations are conducted. Modern HPC infrastructures are mainly composed by compute and storage nodes [14]. Compute nodes are responsible for computational-related tasks, comprehending manycore processors that deliver massive parallelism and vectorization. While cloud premises virtualize heterogeneous commodity hardware to execute general-purpose tasks, high-performance compute nodes comprehend bleeding-edge homogeneous hardware, along fine-tuned libraries for parallel I/O communication (*e.g.,* MPI-I/O [31]) and I/O abstraction (*e.g.,* HDF5 [29], NetCDF [85]). Storage nodes are responsible for storing applications' results in a Petabyte-scale parallel file system (*e.g.,* Lustre [92], OrangeFS [18], GPFS [90]) that offers a global namespace and high-performance storage and archival access, on top of hundreds of storage drives (*e.g.,* SSD, HDD, RAID, tape). Communication across nodes is made available through specialized high-performance interconnects such as InfiniBand or Intel Omni-Path [13]. HPC applications are directly deployed at compute instances, and users reserve a number of cores for a given duration, contrarily to cloud where tenants rent a set of VMs. Furthermore, many of the current *Top500* supercomputers comprehend a third group of HPC nodes, namely I/O forwarding nodes (or I/O nodes) [14, 106]. Such nodes act as a middleware between compute and storage instances, and are responsible for receiving compute nodes' requests and forward them to storage ones. I/O nodes hold the intermediate results of applications, either in memory or high-speed SSDs, and enable several optimizations over I/O flows (*e.g.,* request ordering, aggregation, data staging).

However, the long I/O path of HPC infrastructures, combined with complex software stacks and hardware configurations, make performance isolation, end-to-end control of I/O flow, and I/O optimizations increasingly challenging [117]. The variety of access patterns exhibited by HPC applications has lead modern HPC clusters to observe high levels of I/O interference and performance degradation, inhibiting their ability to achieve predictable and controlled I/O performance [62, 121]. While several efforts were made to prevent I/O contention and performance degradation of HPC infrastructures (*e.g.,* QoS provisioning [116, 123], I/O flow, and job scheduling optimizations [44, 101]), neither have considered the path of end-to-end enforcement of storage policies nor system-wide flow optimizations. To this end, SDS systems have been recently introduced to HPC environments.

Clarisse [40] is a SDS system designed to improve the I/O stack performance of complex large-scale HPC infrastructures. It comprises a queue-based data plane and a hierarchically distributed control plane, and offers the building blocks for designing coordinated system-wide data staging optimizations. The data plane is responsible for staging data between applications and storage nodes, and implements the management and routing mechanisms for transferring data between compute and storage nodes at the middleware layer (*e.g.,* MPI-IO) of compute instances. To alleviate the number of calls and data exchange to storage servers, the data plane exposes the parallel data flows from a supercomputer to its control plane, allowing system designers to develop novel cross-layer mechanisms, such as parallel I/O scheduling, load balancing, and elastic collective I/O. The control plane offers the mechanisms for coordinating and controlling routing-related

activities. Controllers are deployed and hierarchically distributed over compute premises, and are classified in three different control types, namely node, application, and global controllers. Node controllers perform single node orchestration, enforcing control primitives at local data plane stages. Application controllers monitor the nodes an application is running. A single centralized global controller orchestrates the overall control environment, and monitors all running applications, enforces system-level decisions and adapts enforcement plans at runtime. Even though distributed, no assumptions are made about its dependability.

Similarly to Clarisse, SIREN [46] offers the building block for ensuring performance isolation and resource sharing guarantees in HPC infrastructures. It introduces the concept of SDS resource enclaves for resource management of HPC storage settings, allowing users to specify I/O requirements via reservation and sharing of compute and storage resources between applications. Moreover, SIREN enforces end-to-end storage policies for performance-sensitive objectives by dynamically allocating resources according applications' demands. The control plane comprehends several distributed controllers organized in a tree hierarchy. Each controller allocates a set of compute and storage nodes according the I/O requirements of applications, and efficiently enforces performance objectives over such resources. A root controller manages both HPC resources and SDS elements by dynamically adapting (through heuristic and ML techniques) sub-controllers organization, resource allocations, and stage configurations to time-varying workloads. Sub-controllers orchestrate a subset of resources and directly compute over SDS stages. Resources are recursively allocated, being controlled at a finer grain as control functionality descends. Moreover, even though distributed, SIREN's control plane does not ensure dependability of the control environment, and only provides recovery mechanisms, leading to temporary downtime of the controller and overall performance degradation of the HPC stack. Data plane stages are deployed throughout the I/O stack in user-defined enforcement points (*e.g.,* request schedulers, parallel file systems), and are responsible for dispatching I/O packets through a queue-based structure that implements management-based storage features to efficiently enforce the reservations and shares specified by control instances. Stages continuously monitor system performance and resource usage and periodically report them to the control plane. Similarly to Clarisse, SIREN's interfaces communicate through a standard publish/subscribe API to ease communication between SDS elements.

The recent efforts on designing and implementing SDS solutions for HPC environments have proven the utility and feasibility of the paradigm on high-performance technologies. Even though in its infancy, as we move closer to the Exascale era [24], the adoption of SDS-based platforms and services by the scientific community is key to ensure end-to-end enforcement, I/O differentiation, and performance isolation over large-scale supercomputers. When compared to cloud infrastructures, the radically different requirements in I/O performance and storage bandwidth imposed by supercomputers has driven HPC-based SDS systems to adopt hierarchically distributed controllers over flat-based ones, in order to efficiently enforce specialized storage policies and employ high-performance I/O functionalities.

3.3 Application-specific Infrastructures

Application-specific storage infrastructures are storage stacks built from the ground up, designed to serve specialized storage and processing purposes to achieve application-specific I/O optimizations [95]. Examples of real-world production clusters encompass multi-tenant distributed storage systems such as Hadoop [114], Ceph [112], and OpenStack Swift [76], being mainly composed by proxy and storage servers. Proxy servers are responsible for mapping application requests to the respective data location, and comprehend global visibility of the infrastructure (*e.g.,* metadata, namespace, network topology), system-wide management activities (*e.g.,* garbage collection, load balancing, lease management), and high-availability of the storage service. Storage servers are

user-space daemons responsible for storing, retrieving, and deleting application's data. Each of these systems exposes distinct abstractions to applications, and comprises different storage mechanisms to achieve application-specific I/O optimizations and meet the different requirements of performance, scalability, and dependability. While these systems are built to run on commodity hardware [99], enterprise-grade infrastructures may hold hundreds to thousands of storage servers interconnected with dedicated network links (10-56 Gib/s) [86]. Each of these servers accommodates several multi-core processors and a number of storage drives hierarchically organized, consisting of 64-512 GiB of memory and hundreds of GiB of high-speed storage devices for temporary files and hot data, and hundreds of TiB of HDD drives for the main data bulk. However, the specialized environment that surrounds modern application-specific stacks has lead to hard-coded designs and predefined I/O stacks, making the programmability of such systems challenging [95]. Moreover, the absence of performance guarantees and isolation, leads to greedy tenants and background tasks (*e.g.,* garbage collection, replication, load balancing) to consume a large quota of system resources, ultimately impacting overall system performance [63]. While several mechanisms have been proposed to address different system intricacies (*e.g.,* workload-awareness [19], availability [15]), neither have considered end-to-end enforcement of storage policies nor improve programmability of specialized storage stacks. As such, several application-specific SDS systems have been proposed to address such challenges. Even though application-specific infrastructures can be seen as a subfield of cloud computing, or even HPC, for the purpose of this article and to provide a more granular classification of SDS systems, we classify these in a separate category of cloud computing.

The ever-growing requirements in QoS provisioning and performance isolation of distributed storage ecosystems have lead researchers and practitioners to shift from hard-coded single-purpose implementations to software-defined approaches. Retro [63] provides efficient resource management in multi-tenant distributed storage systems, enabling system designers to orchestrate and fine-tune resource management policies independently from the underlying system implementation. A logically flat-based centralized controller provides global view of resources and orchestrates a number of queue-based data plane stages deployed throughout the I/O path. Implemented over the Hadoop stack, Retro provides dynamic and adaptive policy enforcement by collecting near real-time measurements, and continuously enforcing fine-tuned storage policies that react to volatile system requirements. Such a reactive approach allows the system to respond to real-time resource usage workflows, instead of relying on static models of future system requirements. Retro is system- and resource-agnostic, and its data plane abstracts arbitrary resources (*e.g.,* storage devices, CPU, thread pools, network) and implements management-based storage features (*e.g.,* I/O prioritization, rate limiting) at predefined points of the I/O path. Stages comprehend a queue-based design, tailored for management objectives to handle custom system resources and support novel I/O logic. Such a design is key to efficiently enforce end-to-end system objectives and achieve the desired performance guarantees and resource fairness.

While Retro focus on efficient resource management in the Hadoop distributed stack, other systems aim at ensuring sustained efficacy and performance in multi-tenant object stores. Crystal [30] extends the resource management capabilities of Retro by proposing an SDS architecture for object storage to ensure efficient resource sharing and performance isolation under multi-tenant heterogeneous workloads. At the control plane, a set of flat distributed controllers dynamically adapt storage artifacts to comply with evolving requirements of a number of tenants. Each controller is seen as an autonomous micro-service, deployable at runtime, that runs a separate control algorithm to enforce selected points of the storage stack. *Global controllers* comprehend system-wide visibility and continuous control of data plane stages, being responsible for distilling monitoring events and disseminating storage policies to both controllers and data stages, while *automation controllers* ingest predefined monitoring metrics and directly fine-tune stages, in a trigger-based fashion, to

employ specific actions over object requests. Since controllers are lightweight micro-services, fault tolerance is achieved by spawning a new control process if one fails. Further, the control plane exposes a domain-specific language to control applications, hiding the complexities of low-level policy enforcement, and simplifying storage administration. At the data plane, programmable storlet-based stages are deployed at *OpenStack Swift* instances [76] to serve management and transformation storage services. Each of these stages provides a flexible *filter* (storlet) framework that enables developers to deploy and run customized storage services on incoming object requests.

Commercial storage systems have also experienced a thrust towards the software-defined domain. For instance, Coho Data [20, 115] proposes a twofold SDS-based enterprise storage architecture that provides efficient, scalable, and highly-available control over high-performance storage devices (*e.g.,* PCIe storage drives). At the control plane, Mirador [115] provides a dynamic storage placement service to efficiently orchestrate heterogeneous scale-out storage systems. Controllers are organized through a flat distribution, providing efficient control centralization and highly-available control environment, where a coordination service [38] ensures strong resilience of the control environment. Moreover, it provides modular building blocks to continuous monitor and fine-tune storage artifacts, allowing the system to enforce dynamic and flexible data placement policies and to be adaptable under time-varying workloads. At the data plane, Strata [20] implements a stack-based network-attached object store that allows efficient management of high-performance storage devices under multi-tenancy. Moreover, its stacking configuration comprehends tightly coupled and predefined storage services to employ over I/O requests. Specifically, at the first layer, along SDN-enabled switches, Strata abstracts the underlying storage resources to efficiently balance requests without compromising the performance and scalability of the storage stack; at the mid-tier, a global address namespace preserves the centralized control over data placement, reconfiguration, and failure recovery services; at the ending layer, it implements an object store over PCIe flash devices, presenting an OSD-like storage interface. Besides routing and device management, Strata also implements transformation services over I/O requests (*e.g.,* striping, replication, deduplication).

As some storage systems may provide a rich spectrum of storage functionalities (*e.g.,* resource sharing, durability, load balancing), some SDS systems rely on these artifacts to improve control functionality of the storage environment [96, 107]. For instance, Mantle [94, 96] decouples management-based policies from the storage implementation, allowing users to fine-tune and adapt the storage environment as requirements vary over time (*e.g.,* workloads, resource usage). While originally implemented over the Ceph distributed storage system to let system administrators define metadata load balancing policies [96], Mantle is currently implemented as a library to be linked into a storage system service and provide general-purpose management capabilities. At the control plane, a policy engine enables users to inject management-oriented policies into running storage systems, which allows the system to be fine-tuned at runtime. At the data plane, Mantle abstracts underlying storage artifacts through a data management language, that allows users to build flexible and fine-grained policies to implement complex storage systems. On the other hand, SuperCell [107] relies on the flexibility and availability of the storage features of the Ceph distributed storage system. To respond to the fast changing requirements and time-varying workloads imposed over real-world production clusters, SuperCell proposes a SDS-based recommendation engine that measures and provides different configuration settings of existing Ceph deployments. This enables the runtime adaptation of storage artifacts to best meet the different QoS requirements under multi-tenancy. At the control plane, a centralized controller measures detailed workload characteristics of the storage environment (*e.g.,* I/O size, read/write proportion) and generates accurate enforcement strategies tailored to meet user's requirements in a cost-effective manner, while ensuring efficient resource sharing and performance isolation. System administrators will then select an enforcement plan and fine-tune the existing storage stack. At the data plane, SuperCell fine-tunes different

storage settings and configurations of Ceph deployments at runtime, to comply with the manifold management and transformation storage policies.

While previous systems are mainly focused on providing efficient management and fine-grained control over storage systems, others aim at providing novel abstractions and storage features [80, 83, 95]. Malacology [95] is a controllerless SDS system that provides novel storage abstractions by exposing and re-purposing code-hardened storage artifacts (*e.g.,* resources, services, abstractions) into a high-level programmable storage design. Rather than creating storage systems from scratch, Malacology encapsulates storage system functionality into reusable building blocks that enable general-purpose storage systems to be programmed and adapted into tailored storage applications via composition. Due to its wide spectrum of storage functionality, Malacology actuates over Ceph deployments, and decouples policies from the respective implementation mechanisms through a storlet-based data plane that exposes commonly used services as programmable interfaces that hold the main primitives for developing comprehensive storage applications, namely service metadata, data I/O, resource sharing, load balancing, and durability. Such a design allows administrators to have fine-grained control of the storage environment, while enabling efficient enforcement of management-, transformation-, and routing-oriented storage policies.

Following these same principles, SafeFS [80] aims at re-purposing existing FUSE-based file system implementations into stackable storage services to employ over I/O requests. In more detail, SafeFS is a flexible, extensible, and modular stackable data plane that abstracts the file system layer to enable the development of POSIX-compliant file systems atop FUSE. By following a stackable organization, SafeFS achieves layer interoperability and allows system operators to simply stack independent layers in any desired order to attend different management and transformation objectives (*e.g.,* encryption, replication, erasure coding, caching). Such a design reduces the cost of (re-)implementing new storage functionalities by resorting to self-contained, stackable, and reusable layers, and enables the development of tailored and fine-tuned data planes. Likewise, to bridge cluster and big data-oriented file systems, which comprehend significantly different design choices and workload profiles, Hadoop-GPFS [83] provides a stacking configuration between the HDFS [99] and the GPFS [90] distributed file systems. Rather than maintaining two separate storage silos, this stackable design unifies these two file systems, enabling the seamless execution of analytic-oriented applications over them while improving total cost of ownership. The Hadoop-GPFS system is implemented as part of the IBM Spectrum Scale SDS solution [39], which is not further detailed since its design is not publicly disclosed.

Similarly to previous infrastructures (§3.1 – §3.2), while SDS-based solutions received considerable attention from researchers and practitioners, previous works on application-specific storage had already crossed the path of software-defined principles [7, 68]. Specifically, as a first attempt towards SDS, PADS [7] introduced a policy-based architecture to ease the development of custom distributed storage systems. It decouples storage policies from the associated mechanisms by providing a controllerless control plane and a storlet-based data plane. At the control plane, control applications hold a set of routing and blocking policies to define the management and routing logic of the correspondent storage system. Routing policies define the data flows between nodes, while blocking policies specify the predicates for meeting consistency and durability objectives of distributed storage systems. At the data plane, stages accommodate a set of common storage services (*e.g.,* replication, consistency, storage interface) that allow system designers to develop tailored storage systems via composition, by simply defining a set of policies rather than implementing them from scratch. Such flexible and programmable design, enables the development of a wide-range of storage systems accounting with the different trade-offs of performance, reliability, and availability. Differently, Mesnier et al. [68] proposes a classification architecture to achieve I/O differentiation between storage requests. Since the performance of compute servers is often determined by the I/O

interference and performance degradation of storage servers, it proposes a classification framework that is able to classify I/O requests at compute instances, and differentiated them at storage servers according user-defined policies, thus ensuring performance isolation and resource fairness. A controllerless control plane installs management-oriented policies at a block-level data plane, that introduces programmability and extensibility to the formerly rigid block layer. Such a design, allows users to classify I/O requests according different criteria (*e.g.,* metadata, access pattern, file size) and ensure different objectives of performance, reliability, and security.

The introduction of software-defined principles into specialized distributed storage systems have lead to significant improvements in terms of programmability, performance isolation, and resource efficiency of application-specific storage infrastructures. Such a design allows users to experience sustained QoS provisioning and performance isolation in multi-tenant settings, instead of the formerly predefined and single-purposed approaches. Further, the broad applicability and increasing interest of these storage stacks in both academia and enterprise domains, alongside the overwhelming requirements of emerging storage paradigms (*e.g.,* serverless computing [45]), motivate the need to foster research and development of SDS-enabled application-specific technologies.

4 LESSONS LEARNED AND FUTURE DIRECTIONS

Table 1 classifies the surveyed SDS systems according the taxonomy described in §2, while grouping them by storage infrastructure, namely cloud, HPC, and application-specific storage stacks. This table highlights the design space of each storage infrastructure, and depicts current trends of the SDS field and unexplored aspects that require further investigation. The classification considers systems from both academia and industry.[1] Commercial solutions whose specification is not publicly disclosed are not considered for this survey. Systems that clearly define their solution as SDS targeting at least one of the planes of functionality, are in the scope of this classification. Likewise, systems that are not clearly declared as SDS but share similar design principles and storage functionalities are also contemplated in this classification.[2] We now present the key insights provided by this survey, grouped by storage infrastructure (***SI****x*), planes of functionality, namely data (***D****x*) and control planes (***C****x*), SDS interfaces (***I****x*), and other aspects of the field (***O****x*).

SI1: SDS research is widely explored over cloud and application-specific infrastructures. Several SDS systems have been proposed to attain the needs of both cloud and application-specific infrastructures. Cloud-based solutions are full-fledged, mainly composed by flat controllers and queue-based stages, while application-specific environments focus on data plane proposals. The advances of the SDS paradigm in both fields are justified by its direct applicability over storage solutions and the novel trade-offs that it brings to conventional infrastructures [1].

SI2: HPC-based systems are at an early research stage. Due to the increasing requirements of scale and performance of supercomputers, HPC-oriented solutions have just recently made the first advances towards software-defined, being composed by hierarchical control distributions backed by high-performance queue stages. Considering that only a few proposals address this challenge, novel contributions are expected to both advance the knowledge of the SDS-HPC field and to attend the requirements of incoming Exascale premises [1, 24].

SI3: SDS-enabled systems for emerging computing paradigms are unexplored. SDS-enabled systems have been employed over modern storage infrastructures to achieve different storage objectives. However, with the emergence of novel computing paradigms such as serverless cloud computing [45], IoT [5], and Exascale computing [24], a number of challenges (*e.g.,* scalability,

[1]Industrial SDS systems are marked with an ***i*** in the classification table.
[2]SDS-alike systems are marked with ⋆ in the classification table.

Table 1. Classification of Software-Defined Storage systems regarding storage infrastructure.

	Control Plane				Data Plane						
	Distribution	Scalability	Dependability	Adaptability	Design	Programmability	Extensibility	Placement	Transparency	Scope	**Interface**
IOFlow [i] [105]	*F*	◐	●	◐	*Q*	◐	◐	*MP*	○	*MR*	↑ ↓
Moirai [103]	*F*	◐	●	◐	*S*	◐	◐	*MP*	●	*M*	↓
sRoute [102]	*H*	●	●	◐	*Q*	●	◐	*MP*	○	*MR*	↑↔↓
JoiNS [113]	*H*	●	---	◐	*Q*	◐	○	*DSP*	○	*MR*	↔ ↓
Pisces⋆ [98]	*C*	○	○	◐	*Q*	○	○	*DSP*	○	*M*	↓
PM⋆ [125], *WC*⋆ [124]	*F*	◐	●	●	*Q*	○	○	*DSP*	○	*M*	↓
Tiera⋆ [82]	—	—	—	—	*St*	●	●	*SP*	●	*MT*	↓
Wiera⋆ [74]	*C*	○	○	◐	*St*	●	●	*DSP*	●	*MTR*	↓
Wisp⋆ [104]	*D*	●	---	◐	*Q*	◐	○	*DSP*	●	*M*	↔ ↓
flexStore [70]	*F*	◐	●	●	*S*	◐	○	*DSP*	◐	*MT*	↑ ↓
Clarisse [40]	*H*	●	◐	◐	*Q*	◐	○	*DSP*	○	*MR*	↑↔↓
SIREN [46]	*H*	●	◐	●	*Q*	◐	○	*MP*	○	*M*	↔ ↓
Retro [63]	*F*	◐	---	◐	*Q*	◐	◐	*MP*	○	*M*	↑ ↓
Crystal [30]	*F*	◐	●	●	*St*	●	●	*DSP*	◐	*MT*	↑↔↓
Coho Data [i] [20, 115]	*F*	◐	●	◐	*S*	◐	○	*DSP*	●	*MTR*	↑ ↓
Mantle [94]	---	---	---	◐	—	●	○	*DSP*	○	*M*	↑ ↓
SuperCell [107]	*C*	○	○	◐	—	○	○	*DSP*	●	*MT*	↓
Malacology [95]	—	—	—	—	*St*	●	◐	*DSP*	○	*MTR*	↓
Hadoop-GPFS [i] [83]	—	—	—	—	*S*	○	○	*DSP*	●	*T*	↓
SafeFS [80]	—	—	—	—	*S*	◐	●	*SP*	●	*MT*	↓
PADS ⋆ [7]	—	○	○	○	*St*	●	●	*DSP*	○	*MR*	↑ ↓
Mesnier et al.⋆ [68]	—	○	○	○	—	◐	◐	*SP*	○	*M*	↓

Properties
○ Absent
◐ Limited
● Manifested
--- Unspecified
— Not Applicable

Distribution
C-Centralized
F-Flat
H-Hierarchical
D-Decentralized

Design
S-Stackable
Q-Queue
St-Storlet

Placement
SP-Single-point
DSP-Distributed SP
MP-Multi-point

Scope
M-Management
T-Transformation
R-Routing

Interfaces
↑ Northbound
↔ West/Eastbound
↓ Southbound

performance, resiliency) need to be addressed to ensure sustained storage efficiency. As such, novel contributions in SDS architectures over such environments should be expected.

D1: Stage design impacts programmability and extensibility. Several SDS data planes rely on queue-based designs, trading customization and transparency for performance. This performance-focused development has lead queue-based solutions to experience limited programmability and extensibility. Storlet-based solutions however, comprehend a more programmable and extensible design, being able to serve general-purpose storage requirements.

D2: End-to-end enforcement is hard to ensure. Most SDS systems provide distributed enforcement points, bounded to a specific layer of the I/O stack (*e.g.,* hypervisor, file system). Systems that ensure efficient end-to-end policy enforcement comprehend specialized queue-based stages fine-tuned for specific storage services, which require significant code changes to the original codebase. As such, end-to-end enforcement is tightly coupled to the placement property, and directly influences the transparency of data plane stages.

D3: Data management services dominate SDS systems. Data management services have dominated the spectrum of storage policies and services supported by SDS systems. This management-oriented design has lead to a large research gap for the remainder scopes. Nevertheless, the advent of modern storage technologies, such as kernel-bypass [122], NVMe devices [49, 119], and storage disaggregation [50, 97], along the emergence of novel computing paradigms require significant attention and further investigation in order to adapt, extend, and implement novel storage features over SDS data planes [1]. As such, there is a great research opportunity to explore these new technologies in SDS architectures.

D4: End-to-end storlet data planes are unexplored. Despite the acknowledged programmability and extensibility benefits of storlet-based data planes, end-to-end enforcement has not yet been explored with such data types. In fact, there are few proposals on storlet data planes, and several contributions and combinations of storage spaces are possible, being of utmost interest for attaining the requirements of incoming serverless cloud computing [45] and IoT infrastructures [1, 5], as well as on the approximation of HPC and cloud ecosystems [72].

D5: Re-purposing of existing storage subsystems is overlooked. Existing storage services installed at data plane stages are mainly designed from the ground up and fine-tuned for a specific data plane solution. While some solutions already encapsulate existing storage systems as reusable building blocks [80, 83, 95], there is no SDS system that leverages from existing storage subsystems (*e.g.,* QoS provisioning, I/O scheduling) to be re-purposed as programmable storage objects and reused in different storage contexts throughout the I/O path. Such a design would open research opportunities towards programmable storage stacks, and foster reutilization of complementary works [47, 65, 81, 118] and existing storage subsystems [25, 26, 66].

D6: Heterogeneous data planes are unexplored. Despite the large-array of possible combinations and design flavors of SDS data planes, the combination of different stage designs has not yet been explored. Such a design turns the data plane domain mostly monolithic, tailored for specific storage objectives, and may lead to suboptimal performance and enforcement efficiency. As such, following the steps of the SDN paradigm [54], novel contributions towards heterogeneous data plane environments, that explore the different trade-offs of stackable, queue-, and storlet-based designs, should be expected.

C1: Current systems are unsuitable for larger environments. A large quota of SDS systems follow a flat setup, being able to scale for small-to-medium storage infrastructures, made of hundreds of compute servers and tens of storage servers [105]. However, as we move closer to larger storage environments such as serverless cloud computing [45] and IoT infrastructures [5], made of complex and highly-heterogeneous storage stacks, the control centralization assumptions of current SDS systems do not hold true. As such, leveraging from the initial efforts of decentralized controllers [104], it is fundamental to further investigate this topic and provide novel contributions towards control decentralization.

C2: Controllers lack programmability. Current hierarchical SDS controllers resort to delegate functions or micro-services to improve control scalability, providing limited control functionality to

the aforementioned control peers. Rather, it would be interesting to follow similar design principles as SDS data elements and turn control functionality more programmable. Researching different paths of scalability and programmability in SDS would bring major benefits for incoming storage infrastructures [1].

C3: Scalability and dependability are overlooked. The design of SDS control planes has been overlooked since early solutions, which simply define controllers as *"logically centralized"* [105]. However, behind this simple but ambiguous assumption lies a great deal of practical complexity that involves several aspects of SDS dependability, leaving no clear definitions on its practical challenges and actual impact in performance and scalability at the overall storage infrastructure. Moreover, few SDS systems discuss their control plane's dependability. Similarly to other software-defined genres [56], these assumptions leave several open questions regarding controllers dependability that require further investigation, such as fault tolerance and consistency [10, 11, 48, 52], load balancing and control dissemination [23], controller synchronization [89], and concurrency [27].

C4: Controllers are self-adaptable. Several controllers resort to heuristic-based mechanisms to dynamically adjust storage artifacts, while few proposals rely on linear programming and ML techniques to provide a more accurate and comprehensive automation model. However, the storage landscape is changing at a fast pace, with new computing paradigms and emerging hardware technologies vested with novel workload profiles, leaving heuristic-based strategies unsuitable to achieve higher-levels of accuracy and efficacy. As such, it is necessary to advance the research of autonomous mechanisms for supporting control decisions of SDS controllers. For instance, exploring distributed ML techniques [57] would be of great utility to attend the needs of both modern and emerging storage infrastructures, not only for the obvious reasons of scale but also for ensuring new levels of accuracy in such heterogeneous and volatile environments.

I1: Communication protocols are tightly coupled to planes of functionality. Current SDS interfaces are used as simple communication APIs, making the communication protocols to be tightly coupled to either control or data plane implementations. Such a design prevents the reutilization of alternative communication technologies and inhibits current SDS systems to be adaptable to other storage contexts without significant code changes at the communication codebase. As such, decoupling the communication from control and/or data plane implementations would improve the transparency between the two planes of functionality, foster reutilization of communication protocols, and open novel research opportunities to attain the communication challenges of novel storage paradigms [5, 45] and network fabrics [1].

I2: Interfaces lack standardization and interoperability. Contrarily to SDN [54], the SDS literature does not provide any standard interface to achieve interoperability between SDS technologies. Indeed, this lack of standardization leads researchers and practitioners to implement custom interfaces and communication protocols for each novel SDS proposal, tailored for specific software components and storage purposes. Such a design inhibits interoperability between control and data plane technologies, and hinders the independent development of each plane of functionality. As such, novel contributions towards standard and interoperable SDS interfaces should be expected.

O1: Black-box and end-to-end monitoring are unexplored. Monitoring mechanisms in SDS controllers are predefined and static. Integrating black-box [73] and end-to-end monitoring systems [64, 117] in SDS controllers would bring novel insights to the field, and reveal interesting (and unexpected) metrics and I/O behaviors of data flows that could assist SDS controllers to define more accurate enforcement strategies. Moreover, such an approach does not require *a priori* knowledge of the I/O stack and comprehends near-zero changes to the original codebase.

O2: SDS paradigm lacks proper methodologies and benchmarking platforms. Current evaluation methodology of SDS systems is essentially made through trace replaying and benchmarking of specific points of the I/O path, either with specialized or custom-made benchmarks. Thus, there is no comprehensive SDS benchmarking methodology that systematically characterizes the end-to-end performance and design trade-offs of general SDS technologies. As such, these considerations motivate for novel contributions in SDS evaluation.

5 CONCLUSION

In recent years, the Software-Defined Storage paradigm has gained significant traction in the research community, leading to a broad spectrum of academic and commercial proposals to address the shortcomings of conventional storage infrastructures. By reorganizing the I/O stack to disentangle control and data flows into two planes of functionality, SDS has proved to be a fundamental solution to enable end-to-end policy enforcement and holistic control, ensure performance isolation and QoS provisioning, and provide new levels of programmability to storage stacks.

To this end, and to the best of our knowledge, we present the first in-depth survey of the SDS paradigm. Specifically, we explain and clarify fundamental aspects of the field, and provide a comprehensive description of each plane of functionality, namely control and data planes, describing their design principles, internal organization, and storage properties. As a first contribution, we define the SDS design principles and categorize planes of functionality regarding internal organization and distribution. In more detail, we surveyed existing work on SDS and distilled a number of designs for data plane stages, being categorized as stackable, queue-, or storlet-based. At the control plane, we categorize distributed SDS control planes as being flat or hierarchically organized.

Then, as another contribution, we propose a taxonomy and classification of existing SDS systems to organize the manifold approaches according to their storage infrastructure, namely cloud, HPC, and application-specific storage infrastructures. Cloud-based solutions aim at addressing the requirements of general-purpose multi-tenant storage stacks, mainly through flat controllers and queue-based data plane stages. On the other hand, SDS-enabled application-specific systems target specialized storage stacks tailored for specific storage and processing purposes, being particularly focused on the SDS data elements. Lastly, even though at an early research stage, HPC-based solutions follow hierarchical control distributions backed by high-performance queue-based stages, in order to respond to the increasing requirements of scale and performance of supercomputers.

Finally, we provide key insights about this survey and discuss future efforts of the SDS field. Even though significant advances in SDS research have been made in both application-specific and cloud computing infrastructures, several issues can still be improved, namely scalability, control programmability, dependability, heterogeneous data planes, alternative data plane combinations, and novel storage policies and services of less explored data scopes. Noticeably, since HPC-oriented SDS systems are at an early research stage, novel contributions to improve performance variability and I/O contention can be expected. Likewise, novel SDS solutions capable of ensuring the several requirements of scale, performance, and dependability of incoming storage infrastructures, namely serverless cloud computing, IoT, and Exascale computing, can also be expected. To conclude, due to their well-acknowledged benefits, and the need for addressing emerging storage technologies challenges, we believe that SDS research will continue to grow at an accelerated pace in forthcoming years. Moreover, we believe the insights provided by this survey will be of great utility to guide researchers and practitioners to foster the SDS field.

REFERENCES

[1] George Amvrosiadis, Ali R. Butt, Vasily Tarasov, Erez Zadok, Ming Zhao, et al. 2018. *Data Storage Research Vision 2025: Report on NSF Visioning Workshop Held May 30–June 1, 2018.* Technical Report.

[2] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 233–248.

[3] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. 2016. MOS: Workload-aware Elasticity for Cloud Object Stores. In *25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 16)*. ACM, 177–188.

[4] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R. Butt. 2018. bespoKV: Application Tailored Scale-out Key-value Stores. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 18)*. IEEE, 2:1–2:16.

[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks* 54, 15 (2010), 2787–2805.

[6] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. 2018. Distributed SDN Control: Survey, Taxonomy, and Challenges. *IEEE Communications Surveys & Tutorials* 20, 1 (2018), 333–354.

[7] Nalini M. Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Michael Dahlin, and Robert Grimm. 2009. PADS: A Policy Architecture for Distributed Storage Systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*. USENIX Association, 59–73.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating System Design and Implementation (OSDI 14)*. USENIX Association, 49–65.

[9] Samaresh Bera, Sudip Misra, and Athanasios V. Vasilakos. 2017. Software-Defined Networking for Internet of Things: A Survey. *IEEE Internet of Things Journal* 4, 6 (2017), 1994–2008.

[10] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *3rd Workshop on Hot Topics in Software Defined Networking (HotSDN 14)*. ACM, 1–6.

[11] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 14)*. IEEE, 355–362.

[12] Jean-Pascal Billaud and Ajay Gulati. 2013. hClock: Hierarchical QoS for Packet Scheduling in a Hypervisor. In *8th ACM European Conference on Computer Systems (EuroSys 13)*. ACM, 309–322.

[13] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. 2015. Intel® Omni-Path Architecture: Enabling Scalable, High Performance Fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 1–9.

[14] Francieli Zanon Boito, Eduardo C. Inacio, Jean Luca Bez, Philippe O.A. Navaux, Mario A.R. Dantas, and Yves Denneulin. 2018. A Checkpoint of Research on Parallel I/O for High-Performance Computing. *ACM Computing Surveys* 51, 2 (2018), 23:1–23:35.

[15] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. 2011. Apache Hadoop Goes Realtime at Facebook. In *2011 ACM SIGMOD International Conference on Management of Data (SIGMOD 11)*. ACM, 1071–1080.

[16] Fábio Botelho, Alysson Bessani, Fernando M.V. Ramos, and Paulo Ferreira. 2014. On the Design of Practical Fault-Tolerant SDN Controllers. In *2014 3rd European Workshop on Software Defined Networks (EWSDN 14)*. IEEE, 73–78.

[17] Fábio Botelho, Tulio A. Ribeiro, Paulo Ferreira, Fernando M.V. Ramos, and Alysson Bessani. 2016. Design and Implementation of a Consistent Data Store for a Distributed SDN Control Plane. In *2016 12th European Dependable Computing Conference (EDCC 16)*. IEEE, 169–180.

[18] Philip H. Carns, Walter B. Ligon III, Robert Ross, and Rajeev Thakur. 2000. PVFS: A Parallel File System for Linux Clusters. In *4th Annual Linux Showcase and Conference*. 391–430.

[19] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. 2013. MeT: Workload Aware Elasticity for NoSQL. In *8th ACM European Conference on Computer Systems (EuroSys 13)*. ACM, 183–196.

[20] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. 2014. Strata: High-Performance Scalable Storage on Virtualized Non-volatile Memory. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX Association, 17–31.

[21] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 13)*. ACM, 77–88.

[22] Sarah M. Diesburg and An-I Andy Wang. 2010. A Survey of Confidential Data Storage and Deletion Methods. *ACM Computing Surveys* 43, 1 (2010), 2:1–2:37.

[23] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. 2013. Towards an Elastic Distributed SDN Controller. In *2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 13)*. ACM, 7–12.

[24] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, et al. 2011. The International Exascale Software Project Roadmap. *The International Journal of High Performance Computing Applications* 25, 1 (2011), 3–60.

[25] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. 2014. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS 14)*. IEEE, 155–164.

[26] Matthieu Dorier, Matthieu Dreher, Tom Peterka, and Robert Ross. 2017. CoSS: Proposing a Contract-based Storage System for HPC. In *2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS@HPC 17)*. ACM, 13–18.

[27] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. 2016. SDNRacer: Concurrency Analysis for Software-defined Networks. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 16)*. ACM, 402–415.

[28] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid H. Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2010. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *ACM SIGCOMM 2010 Conference (SIGCOMM 10)*. ACM, 339–350.

[29] Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A File Format and I/O Library for High Performance Computing Applications. In *Proceedings of Supercomputing*, Vol. 99. 5–33.

[30] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. 2017. Crystal: Software-Defined Storage for Multi-tenant Object Stores. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 243–256.

[31] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing* 22, 6 (1996), 789–828.

[32] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2007. *p*Clock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 07)*. ACM, 13–24.

[33] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2010. *m*Clock: Handling Throughput Variability for Hypervisor IO Scheduling. In *9th USENIX Conference on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, 437–450.

[34] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter Varman. 2012. Demand Based Hierarchical QoS Using Storage Resource Pools. In *2012 USENIX Annual Technical Conference (ATC 12)*. USENIX Association, 1–13.

[35] Soheil Hassas Yeganeh and Yashar Ganjali. 2012. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *1st Workshop on Hot Topics in Software Defined Networks (HotSDN 12)*. ACM, 19–24.

[36] Chin-Jung Hsu, Rajesh K. Panta, Moo-Ryong Ra, and Vincent W. Freeh. 2016. Inside-Out: Reliable Performance Prediction for Distributed Storage Systems in the Cloud. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS 16)*. IEEE, 127–136.

[37] Markus C. Huebscher and Julie A. McCann. 2008. A Survey of Autonomic Computing — Degrees, Models, and Applications. *ACM Computing Surveys* 40, 3 (2008), 7:1–7:28.

[38] Patrick Hunt, Mahadev Konar, Flavio Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (ATC 10)*. USENIX Association.

[39] IBM. 2019. IBM Spectrum Storage for data-driven architecture. Retrieved April 8, 2019 from https://www.ibm.com/it-infrastructure/storage/spectrum.

[40] Florin Isaila, Jesus Carretero, and Rob Ross. 2016. Clarisse: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 16)*. IEEE, 346–355.

[41] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined Wan. In *2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 13)*. ACM, 3–14.

[42] Yaser Jararweh, Mahmoud Al-Ayyoub, Elhadj Benkhelifa, Mladen Vouk, Andy Rindos, et al. 2015. SDIoT: a software defined based internet of things framework. *Journal of Ambient Intelligence and Humanized Computing* 6, 4 (2015), 453–461.

[43] Yaser Jararweh, Mahmoud Al-Ayyoub, Ala' Darabseh, Elhadj Benkhelifa, Mladen Vouk, and Andy Rindos. 2016. Software defined cloud: Survey, system and evaluation. *Future Generation Computer Systems* 58 (2016), 56–74.

[44] Xu Ji, Bin Yang, Tianyu Zhang, Xiaosong Ma, Xiupeng Zhu, Xiyang Wang, Nosayba El-Sayed, Jidong Zhai, Weiguo Liu, and Wei Xue. 2019. Automatic, Application-Aware I/O Forwarding Resource Allocation. In *17th USENIX Conference*

on File and Storage Technologies (FAST 19). USENIX Association, 265–279.

[45] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, et al. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley.

[46] Suman Karki, Bao Nguyen, and Xuechen Zhang. 2018. QoS Support for Scientific Workflows Using Software-Defined Storage Resource Enclaves. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS 18)*. IEEE, 95–104.

[47] Ian A. Kash, Greg O'Shea, and Stavros Volos. 2018. DC-DRF: Adaptive Multi-Resource Sharing at Public Cloud Scale. In *9th ACM Symposium on Cloud Computing (SoCC 18)*. ACM, 374–385.

[48] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. 2015. Ravana: Controller Fault-tolerance in Software-defined Networking. In *1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR 15)*. ACM, 4:1–4:12.

[49] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association.

[50] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *11th European Conference on Computer Systems (EuroSys 16)*. ACM, 29:1–29:15.

[51] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 427–444.

[52] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. 2010. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, 25:1–25:14.

[53] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *12th European Conference on Computer Systems (EuroSys 18)*. ACM, 25:1–25:17.

[54] Diego Kreutz, Fernando M.V. Ramos, Paulo E. Verissimo, Christian E. Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE* 103, 1 (2015), 14–76.

[55] Leslie Lamport. 1998. The Part-time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[56] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. 2012. Logically Centralized?: State Distribution Trade-offs in Software Defined Networks. In *1st Workshop on Hot Topics in Software Defined Networks (HotSDN 12)*. ACM, 1–6.

[57] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 583–598.

[58] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. 2016. PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *11th European Conference on Computer Systems (EuroSys 16)*. ACM, 28:1–28:14.

[59] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. 2017. Customizable SLO and Its Near-Precise Enforcement for Storage Bandwidth. *ACM Transactions on Storage* 13, 1 (2017), 6:1–6:25.

[60] libfuse. 2001. libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. Retrieved Jan. 28, 2019 from https://github.com/libfuse/libfuse.

[61] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. 2010. Automated Control for Elastic Storage. In *7th International Conference on Autonomic Computing (ICAC 10)*. ACM, 1–10.

[62] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 2010. Managing Variability in the IO Performance of Petascale Storage Systems. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10)*. IEEE, 1–12.

[63] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, 589–603.

[64] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 11:1–11:28.

[65] Adam Manzanares, Filip Blagojević, and Cyril Guyot. 2017. IOPriority: To The Device and Beyond. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association.

[66] Parisa Jalili Marandi, Christos Gkantsidis, Flavio Junqueira, and Dushyanth Narayanan. 2016. Filo: Consolidated Consensus as a Cloud Service. In *2016 USENIX Annual Technical Conference (ATC 16)*. USENIX Association, 237–249.

[67] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.

[68] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. 2011. Differentiated Storage Services. In *23rd ACM Symposium on Operating Systems Principles (SOSP 11)*. ACM, 57–70.

[69] Microsoft. 2019. Microsoft Windows Server. Retrieved April 4, 2019 from https://www.microsoft.com/en-us/cloud-platform/software-defined-storage.

[70] Muthukumar Murugan, Krishna Kant, Ajaykrishna Raghavan, and David H.C. Du. 2014. flexStore: A Software Defined, Energy Adaptive Distributed Storage Framework. In *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS 14)*. IEEE, 81–90.

[71] NetApp. 2019. NetApp ONTAP Select. Retrieved April 4, 2019 from https://www.netapp.com/us/products/data-management-software/ontap-select-sds.aspx.

[72] Marco A.S. Netto, Rodrigo N. Calheiros, Eduardo R. Rodrigues, Renato L.F. Cunha, and Rajkumar Buyya. 2018. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. *ACM Computing Surveys* 51, 1 (2018), 8:1–8:29.

[73] Francisco Neves, Nuno Machado, and José Pereira. 2018. Falcon: A Practical Log-Based Analysis Tool for Distributed Systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 18)*. IEEE, 534–541.

[74] Kwangsung Oh, Abhishek Chandra, and Jon Weissman. 2016. Wiera: Towards Flexible Multi-Tiered Geo-Distributed Cloud Storage Instances. In *25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 16)*. ACM, 165–176.

[75] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (ATC 14)*. USENIX Association, 305–319.

[76] OpenStack. 2018. OpenStack Documentation: Storlets. Retrieved Jan. 25, 2019 from https://docs.openstack.org/storlets/latest/.

[77] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 14)*. ACM, 471–484.

[78] João Paulo and José Pereira. 2014. A Survey and Classification of Storage Deduplication Systems. *ACM Computing Surveys* 47, 1 (2014), 11:1–11:30.

[79] Simon Peter, Jialin Li, Irene Zhang, Dan R.K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 1–16.

[80] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. 2017. SafeFS: A Modular Architecture for Secure User-space File Systems: One FUSE to Rule Them All. In *10th ACM International Systems and Storage Conference (SYSTOR 17)*. ACM, 9:1–9:12.

[81] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. 2017. A Configurable Rule Based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 17)*. ACM, 6:1–6:12.

[82] Ajaykrishna Raghavan, Abhishek Chandra, and Jon B. Weissman. 2014. Tiera: Towards Flexible Multi-tiered Cloud Storage Instances. In *15th International Middleware Conference (Middleware 14)*. ACM, 1–12.

[83] Ramya Raghavendra, Pranita Dewan, and Mudhakar Srivatsa. 2016. Unifying HDFS and GPFS: Enabling Analytics on Software-Defined Storage. In *17th International Middleware Conference (Middleware 16)*. ACM.

[84] David Reinsel, John Gantz, and John Rydning. 2017. Data Age 2025: The Evolution of Data to Life-Critical. Don't Focus on Big Data; Focus on the Data That's Big. *International Data Corporation (IDC) White Paper* (2017).

[85] Russ Rew and Glenn Davis. 1990. NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications* 10, 4 (1990), 76–82.

[86] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:: The Magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[87] Erik Riedel, Garth Gibson, and Christos Faloutsos. 1998. Active Storage for Large-Scale Data Mining and Multimedia Applications. In *24th Conference on Very Large Databases*. Citeseer, 62–73.

[88] Eric W.D. Rozier, Pin Zhou, and Dwight Divine. 2013. Building Intelligence for Software Defined Data Centers: Modeling Usage Patterns. In *6th International Systems and Storage Conference (SYSTOR 13)*. ACM, 20:1–20:10.

[89] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. 2016. In-Band Synchronization for Distributed SDN Control Planes. *SIGCOMM Computer Communication Review* 46, 1 (2016), 37–43.

[90] Frank B. Schmuck and Roger L. Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*. 231–244.

[91] Bianca Schroeder and Garth A. Gibson. 2007. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?. In *5th USENIX Conference of File and Storage Technologies (FAST 12)*. USENIX Association, 1–16.

[92] Philip Schwan. 2003. Lustre: Building a File System for 1000-node Clusters. In *Proceedings of the 2003 Linux Symposium*, Vol. 2003. 380–386.

[93] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 67–80.

[94] Michael A. Sevilla, Carlos Maltzahn, Peter Alvaro, Reza Nasirigerdeh, Bradley W. Settlemyer, Danny Perez, David Rich, and Galen M. Shipman. 2018. Programmable Caches with a Data Management Language and Policy Engine. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 18)*. IEEE/ACM, 203–212.

[95] Michael A. Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. 2017. Malacology: A Programmable Storage System. In *12th European Conference on Computer Systems (EuroSys 17)*. ACM, 175–190.

[96] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. 2015. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 15)*. IEEE, 1–12.

[97] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 69–87.

[98] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, 349–362.

[99] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10.

[100] Harcharan Jit Singh and Seema Bawa. 2018. Scalable Metadata Management Techniques for Ultra-Large Distributed Storage Systems – A Systematic Review. *ACM Computing Surveys* 51, 4 (2018), 82:1–82:37.

[101] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. 2011. Server-side I/O Coordination for Parallel File Systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 11)*. ACM, 17:1–17:11.

[102] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. 2016. sRoute: Treating the Storage Stack Like a Network. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, 197–212.

[103] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. 2015. Software-Defined Caching: Managing Caches in Multi-Tenant Data Centers. In *6th ACM Symposium on Cloud Computing (SoCC 15)*. ACM, 174–181.

[104] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed Resource Management Across Process Boundaries. In *10th ACM Symposium on Cloud Computing (SoCC 17)*. ACM, 611–623.

[105] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: A Software-Defined Storage Architecture. In *24th ACM Symposium on Operating Systems Principles (SOSP 13)*. ACM, 182–196.

[106] Top500. 2019. Top 500: Supercomputers. Retrieved April 4, 2019 from https://www.top500.org/.

[107] Keitaro Uehara, Yu Xiang, Yih-Farn Robin Chen, Matti Hiltunen, Kaustubh Joshi, and Richard Schlichting. 2018. SuperCell: Adaptive Software-Defined Storage for Cloud Storage Workloads. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 18)*. IEEE, 103–112.

[108] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. 2015. Guardat: Enforcing data policies at the storage layer. In *10th European Conference on Computer Systems (EuroSys 10)*. ACM, 13:16–13:16.

[109] VMware. 2019. VMware Cloud. Retrieved April 4, 2019 from https://cloud.vmware.com/.

[110] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. 2012. Characteristics of Backup Workloads in Production Systems. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. USENIX Association, 33–48.

[111] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. 2012. Cake: Enabling High-level SLOs on Shared Storage Systems. In *3rd ACM Symposium on Cloud Computing (SoCC 12)*. ACM, 14:1–14:14.

[112] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D.E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, 307–320.

[113] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. 2018. JoiNS: Meeting Latency SLO with Integrated Control for Networked Storage. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 18)*. IEEE, 194–200.

[114] Tom White. 2012. *Hadoop: The definitive guide*. O'Reilly Media, Inc.

[115] Jake Wires and Andrew Warfield. 2017. Mirador: An Active Control Plane for Datacenter Storage. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 213–228.

[116] Yiqi Xu, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato Figueiredo, and Seetharami Seelam. 2012. vPFS: Bandwidth Virtualization of Parallel Storage Systems. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST 12)*. IEEE, 1–12.

[117] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, Weiguo Liu, and Wei Xue. 2019. End-to-end I/O Monitoring on a Leading Supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 379–394.

[118] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Split-level I/O Scheduling. In *25th Symposium on Operating Systems Principles (SOSP 15)*. ACM, 474–489.

[119] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom 17)*. IEEE, 154–161.

[120] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2017. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *2017 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 17)*. ACM, 432–445.

[121] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. 2016. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS 16)*. IEEE, 750–759.

[122] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. 2019. I'm Not Dead Yet!: The Role of the Operating System in a Kernel-Bypass Era. In *17th Workshop on Hot Topics in Operating Systems (HotOS 19)*. ACM, 73–80.

[123] Xuechen Zhang, Kei Davis, and Song Jiang. 2011. QoS Support for End Users of I/O-intensive Applications Using Shared Storage Systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 11)*. ACM, 18:1–18:12.

[124] Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. 2017. WorkloadCompactor: Reducing Datacenter Cost While Providing Tail Latency SLO Guarantees. In *8th ACM Symposium on Cloud Computing (SoCC 17)*. ACM, 598–610.

[125] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2014. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *5th ACM Symposium on Cloud Computing (SoCC 14)*. ACM, 29:1–29:14.

Received May 2019; revised March 2009; accepted June 2009