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 infrastructure 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 according to 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*; • **Computer systems organization** → *Distributed architectures*; • **Information systems** → *Storage 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. 2020. A Survey and Classification of Software-Defined Storage Systems. *ACM Comput. Surv.* 53, 3, Article 48 (May 2020), 38 pages.
https://doi.org/10.1145/3385896

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 [78]. Efficient large-scale storage systems will be essential for handling this proliferation of data, which must be persisted for future processing and backup purposes. However, efficiently storing such a deluge of data is a complex and resource-demanding

This work was financed by the Portuguese funding agency FCT—Fundação para a Ciência e a Tecnologia through national funds, the PhD grant SFRH/BD/146059/2019, the project ThreatAdapt (FCT-FNR/0002/2018), the LASIGE Research Unit (UIDB/00408/2020), and cofunded by the FEDER, where applicable.
Authors' addresses: R. Macedo, J. Paulo, and J. Pereira, INESC TEC & University of Minho, Departamento de Informática, Universidade do Minho, Campus de Gualtar. 4710-057 Braga, Portugal; emails: {ricardo.g.macedo, joao.t.paulo}@inesctec.pt, jop@di.uminho.pt; A. Bessani, LASIGE & Faculdade de Ciências da Universidade de Lisboa, Campo Grande, 1749-016 Lisboa, Portugal; email: anbessani@fc.ul.pt.

0360-0300/2020/05-ART48 $15.00
https://doi.org/10.1145/3385896

task that raises unprecedented performance, scalability, reliability, and programmability challenges.

First, as the complexity of 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 feature several layers along the I/O path providing compute, network, and storage functionalities, including operating systems, hypervisors, caches, schedulers, file systems, and device drivers [99]. Each of these layers includes a predetermined set of services (e.g., caching, queueing, data placement) with strict interfaces and isolated procedures to employ over requests, leading to a complex, limited, and coarse-grained treatment of I/O flows. Moreover, data-centric operations such as routing, processing, and management are, in most cases, blended as a monolithic block hindering the enforcement of end-to-end storage policies (e.g., bandwidth aggregation, I/O prioritization), thus limiting the scalability and flexibility of infrastructures [96].

Second, efficiently managing system resources in multi-tenancy environments becomes progressively harder as resources are shared across multiple processes and nodes along the I/O path (e.g., shared memory, schedulers, devices). Further, as tenants have different service requirements and workload profiles, traditional resource management mechanisms fail to ensure performance isolation and resource fairness due to their rigid and coarse-grained I/O management [60]. As a result, enforcing QoS under multi-tenancy is infeasible as differentiated treatment of I/O flows, global knowledge of system resources, and end-to-end control and coordination of the infrastructure are not ensured [62, 97, 120].

Third, storage infrastructures have become highly heterogeneous, being subject to volatile and bursty workloads over long periods of time [19]. Additionally, infrastructures are frequently tuned offline with monolithic configuration setups [3]. As a result, this homogeneity has led to applications running on general-purpose I/O stacks, competing for system resources in non-optimal fashion and incapable of performing I/O differentiation and end-to-end system optimization [26, 96].

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 [26]. Individually fine-tuning and optimizing each layer of the I/O stack (i.e., in a non-holistic fashion) of large-scale infrastructures 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, 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 [99]. 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 under multi-tenancy. SDS inherits legacy concepts from Software-Defined Networking (SDN) [50] and applies them to storage-oriented environments, bringing new insights to storage stacks, such as improved system programmability and extensibility [75, 87], fine-grained resource orchestration [60, 66], and end-to-end QoS, maintainability, and flexibility [42, 99]. Furthermore, by breaking the vertical alignment of conventional designs, SDS systems provide holistic orchestration of heterogeneous infrastructures, ensure system-wide visibility of storage components, and enable straightforward enforcement of storage objectives.

Recently, SDS 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, leading to an ambiguous conceptualization 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 describes the Software-Defined Cloud paradigm and surveys other software-defined approaches, such as networking, storage, systems, and security [38]. However, it 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 its internals or existing limitations and open questions.

In this article, we present the first comprehensive literature survey of large-scale data center SDS systems, 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, as well as internal control and enforcement strategies. 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 in a separate category. In more detail, this article provides the following contributions:

- ***Provides a definition of SDS.*** We present a definition of the SDS paradigm and outline the distinctive characteristics of an SDS-enabled infrastructure.
- ***Describes an abstract SDS architecture and identifies its main design principles.*** The work presented in this article goes beyond reviewing existing literature and categorizes SDS planes of functionality regarding their designs. We surveyed existing work on SDS and distilled the key design concepts for both controllers and data plane stages.
- ***Proposes a taxonomy and classification for SDS systems.*** We propose a taxonomy and classification of existing SDS solutions in order to organize the many approaches, bringing significant research directions into focus. Solutions are classified and analyzed regarding storage infrastructure, control strategy, and enforcement strategy.
- ***Draws lessons from and outlines future directions for SDS research.*** We provide key insights about this survey and investigate the open research challenges for this field.

This survey focuses on programmable and adaptable SDS systems. Namely, we do not address the design and limitations of either specialized storage systems (e.g., file systems, block devices, object stores) or other fields of storage research (e.g., deduplication [73], confidentiality [20], metadata management [94], device failures [83], non-volatile memory [45]). Autonomic computing systems are also out of the scope of this article [33]. Furthermore, even though other software-defined approaches share similar design principles, they are out of the scope of this article, including but not limited to networking [7, 50], operating systems [9, 74], data center [2, 80], cloud [38], key-value stores [4, 47], flash [72, 85], security [49, 102], and Internet of Things (IoT) [10, 37].

The remainder of this article is structured as follows. Section 2 presents the fundamentals of the SDS paradigm by outlining its distinctive characteristics and introduces a classification for SDS systems. Section 3 surveys existing SDS systems grouped by storage infrastructure, control strategy, and enforcement strategy. In Section 4, we discuss the current research focus and investigate the open challenges and future directions of SDS systems. Section 5 presents the final remarks of the survey.

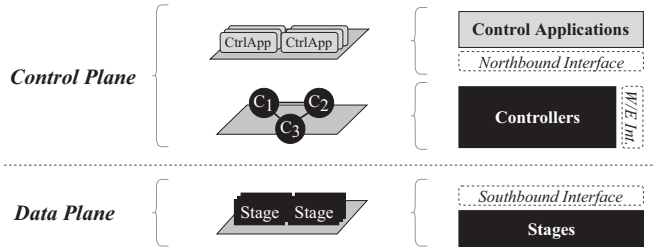

Fig. 1. Layered view of the SDS planes of functionality, comprehending both control and data tiers.

2 SOFTWARE-DEFINED STORAGE

SDS 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*. While the storage industry defines SDS as a storage architecture that simply separates software from vendor lock-in hardware [30, 103], existing SDS solutions are more comprehensive than that. As such, we define an SDS-enabled system as a storage architecture with four main principles.

- ***Storage mechanisms are decoupled from the policies that govern them.*** Instead of designing monolithic custom-made services at each I/O layer, SDS decouples the control functionality from the storage service to be employed over data.
- ***Storage service is moved to a programmable data plane.*** Services to be employed over I/O flows are implemented over programmable structures and fine-tuned to meet user-defined requirements.
- ***Control logic is moved to an external control plane.*** Control logic is implemented at a logically or physically decoupled control plane and properly managed by applications built on top.
- ***Storage policies are enforced over I/O flows.*** Service enforcement is data centric rather than system centric, being employed over arbitrary layers and resources along the I/O path.

Unlike traditional storage solutions, which require designing and implementing individual control tasks at each I/O layer, such as coordination, metadata management, and monitoring, SDS brings a general system abstraction where control primitives are implemented at the control platform. This separation of concerns breaks the storage control into tractable pieces, offering the possibility to program I/O resources and provide end-to-end adaptive control over large-scale infrastructures.

An SDS architecture comprises two planes of functionality. Figure 1 depicts a layered view of such an architecture. The control plane comprehends 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 (Section 2.2) that shares global system visibility and centralized control, and several control applications (Section 2.3) built on top [26, 42, 96, 99]. The data plane (Section 2.1) is composed by several stages that employ controller-defined storage operations over I/O flows [18, 60, 99]. Communication between components is established through specialized interfaces. To preserve a common terminology between software-defined approaches, we adopt the terminology from SDN, namely *Northbound*, *Southbound*, and *Westbound/Eastbound interfaces* [50].

In an SDS-enabled architecture, such as the one depicted in Figure 2, 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

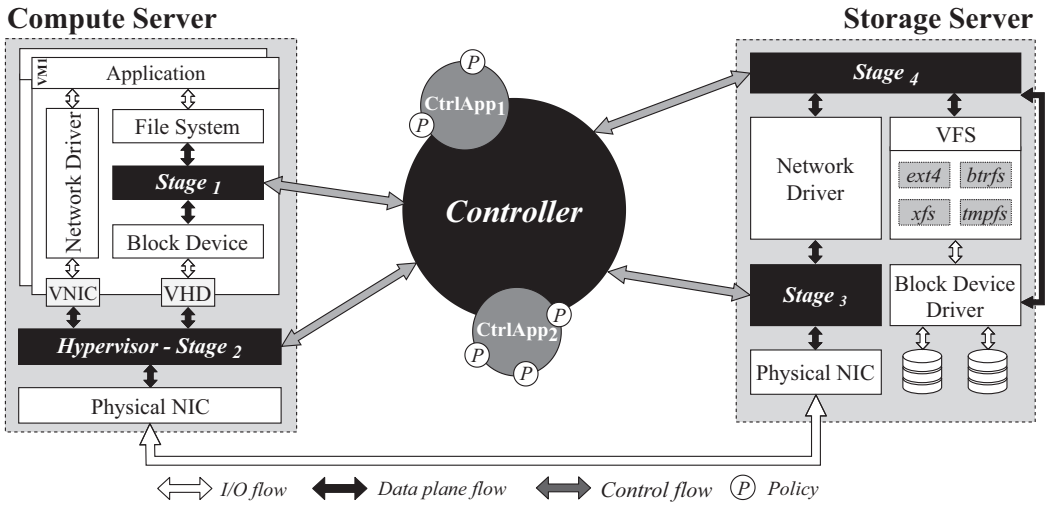

Fig. 2. *SDS*-enabled architecture materialized on top of a general-purpose multi-tenant storage infrastructure. *Compute* servers are virtualized and host virtual machines interconnected to the *hypervisor* by virtual devices, namely *virtual NIC* and *virtual hard disk*. *Storage* servers comprehend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

be enforced over the storage infrastructure. Policies are sets 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 [99] or define request prioritization to ensure X^{th} percentile latency [54]. As controllers share a centralized view, applications resort to centralized algorithms to implement the control logic, which are simpler and less error prone than designing the corresponding decentralized versions [99]. The scope of applications (and policies enforced by these) is broad, ranging from performance- and security-related services [59, 75, 99] to resource and data management functionalities [60, 87, 91]. These user-defined policies are shared between applications and controllers through a *Northbound interface*, which defines the instruction set of the control tier while abstracting the distributed control environment into a centralized one.

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 holistically. Centralized control enables an efficient enforcement of policies and simplifies storage configuration [99]. Policies are handled by a planning engine that translates centralized policies into stage-specific rules and operation logic, which are disseminated to targeted data plane stages and synchronized with other controllers. Communication with the data plane is achieved through a *Southbound interface* (illustrated in Figure 2 with gray-toned arrows), which allows controllers to exercise direct control over data plane stages through policy dissemination and data plane monitoring. Moreover, a *Westbound/Eastbound interface* establishes the communication between controllers to ensure coordination and agreement [26, 96].

The data plane (Section 2.1) is a multi-stage component distributed along the I/O path (Figure 2: $Stage_1 \cdots Stage_4$) that holds fine-grained storage services dynamically adaptable to the infrastructure status. Each stage employs a distinct storage service over intercepted data flows, such as performance management (e.g., prioritization) [54, 97], data management (e.g., compression, encryption) [75, 77], and data routing (e.g., flow customization) [35, 96]. For all intercepted requests, any matching policy will employ the respective service over filtered data and redirect to the corresponding data flow (e.g., Figure 2: data flow between *File System* $\leftrightarrow$ $Stage_1$ $\leftrightarrow$ *Block Device*).

The remainder of this section characterizes planes of functionality in a bottom-up fashion, namely data plane (Section 2.1), controllers (Section 2.2), and control applications (Section 2.3). Moreover, we introduce a taxonomy for classifying SDS systems based on the design principles and features of both control and data planes, which is used as classification criteria for the surveyed work in Section 3.

2.1 Data Plane

Data plane stages are multi-tiered components distributed along the I/O path that perform storage-related operations over incoming I/O requests. Stages are the endpoint 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 establishes a flexible enforcement point 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, *input* and *output* interfaces marshal and unmarshal data flows to be both enforced at the *core* and correctly forwarded to the next I/O layer. For instance, SafeFS exposes a POSIX-compliant interface to enable the transparent enforcement of policies over file systems [75]. The stage *core* comprises (1) a *policy store*, to orchestrate enforcement rules installed by the control plane (similar to an SDN flow table [63]); (2) an *I/O filter*, which matches incoming requests with installed policies; and (3) an *enforcement structure*, which employs the respective storage features over filtered data. A thorough description of this component is presented in Section 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 bridge between control and data planes and establishes the operators that can be used for direct control (e.g., policy propagation, sharing monitoring information, fine-tuning storage services). The control API exhibits to the control plane the instructions a stage understands in order to configure it to perform local decisions (e.g., manage storage policies, fine-tune configurations). Such an API can be represented in a variety of formats. For example, IOFlow [99] and sRoute [96] use tuples comprising human-friendly identifiers to control stages, while Crystal [26] resorts to a system-agnostic domain-specific language to simplify stage administration. Further, the *Southbound interface* acts as a communication middleware and defines the communication models between these two planes of functionality (e.g., publish-subscribe, RPC, REST, RDMA-enabled channels).

Despite the SDN influence, the divergence between storage and networking areas has driven SDS to comprehend fundamentally different design principles and system properties. First, each field targets distinct stack components, leading to significantly different policy domains, services, and data plane designs. Second, contrarily to an SDN data plane, whose stages are simple networking devices specialized in packet forwarding [50], such as switches, routers, and middleboxes, SDS-enabled stages hold a variety 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 [50], while SDS ones demand accurate enforcement points; otherwise, it may disrupt the SDS environment and introduce a significant performance penalty.

2.1.1 Properties. We now define the properties that characterize SDS data tiers, namely programmability, extensibility, stage placement, transparency, and policy scope. These properties are not mutually exclusive (i.e., a data plane can comprise several of them) and are contemplated as part of the taxonomy for classifying SDS solutions (Section 3).

Programmability. Programmability refers to the ability of a data plane to adapt and program existing storage services provided by the stage's enforcement structure (e.g., stacks, queues, storlets) to develop fine-tuned and configurable storage services [87]. In SDS data tiers, programmability is usually exploited to ensure I/O differentiation [26, 99], service isolation and customization [42, 96, 97], and development of new storage abstractions [87]. Conventional storage infrastructures are typically tuned with monolithic setups to handle different applications with time-varying requirements, leading them to experience the same service level [3, 19]. SDS programmability prevents such an end by adapting the stage *core* to provide differentiation of I/O requests and ensure service isolation (further description of this process in Section 2.2.2). Moreover, programmable storage systems can ease service development by repurposing 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 services (e.g., load balancing, durability, metadata management) to a more accessible environment [87]. For example, Mantle [86] decouples cache management services of storage systems into independent policies so they can be dynamically adapted and repurposed.

Extensibility. Extensibility refers to how easy it is for stages to support additional storage services or customize existing ones. An *extensible* data plane consists of 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 to different requirements of a variety of applications, as well as to broaden the 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 literature, highly extensible data plane implementations are built atop flexible and *extensible by design* storage systems (e.g., FUSE [56], OpenStack Swift [71]). For instance, SafeFS [75] allows developers to extend its design with new self-contained storage services (e.g., encryption, replication, erasure coding) without requiring changing its core codebase. However, behind this flexible and generic design lies a great deal of storage complexity that if not properly assessed can introduce significant performance overhead. On the other hand, an *inextensible* data plane typically holds a rigid implementation and hard-wired storage services, tailored for a predefined subset of storage policies. Such a design bears a more straightforward and fine-tuned system implementation, and thus comprehends a more strict policy domain only applicable to a limited set of scenarios.

Placement. The placement of stages refers to the overall position on the I/O path on which a stage can be deployed. It defines the control granularity of SDS systems and is a key enabler to ensure efficient policy enforcement. Each stage is considered as an enforcement point. Fewer enforcement points lead to a coarse-grained treatment of I/O flows, while more points allow for a fine-grained management. Since the control plane has system-wide visibility, broadening the enforcement domain allows controllers to accurately determine the most suitable place to enforce a specific storage policy [99]. An improper number and placement of stages may disrupt the control environment, and therefore introduce significant performance and scalability penalties to the overall infrastructure.

Depending on the storage context and cluster size, 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, being tightly coupled to a specific I/O layer or storage component, as in SafeFS [75] (file system) and Mesnier et al. [64] (block layer). However, this setting 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 settings. Distributed placement of stages (*i.e., distributed single-points*) is associated to distributed storage components of an individual I/O layer (e.g., distributed file systems [60], object stores [26, 66]). In this scenario, each enforcement point is a data plane stage 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. It is, however, 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 of the I/O layer [96, 97, 99]. This design provides a fine-grained control over stages and is key to achieve end-to-end policy enforcement. However, it can introduce significant complexity in data plane development and often requires direct implementation over I/O layers, i.e., following a more intrusive approach.

Transparency. The transparency of a data plane reflects on how seamless its integration is with I/O layers. A *transparent* stage is often placed between storage components and preserves the original I/O flow semantics [75]. For instance, Moirai [97] provides direct control over the distributed caching infrastructure, being applicable across different layers of the I/O path. Such an integration, however, may require substantial marshaling and unmarshaling activities, directly impacting the latency of I/O requests. Contrarily, an *intrusive* stage implementation is tailored for specific storage contexts and can achieve higher levels of performance since it does not require semantic conversions [26, 99]. However, this may entail significant changes to the original codebase, imposing major challenges in developing, deploying, and maintaining such stages, 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 several purposes, namely performance, security, and resource and data management optimizations. To cope with these objectives, SDS systems comprehend a large array of storage policies categorized in three main scopes, namely *performance management*, *data management*, and *data routing*. Noticeably, the support for different scopes relies on the data plane implementation and storage context. *Performance management* services are associated to performance-related policies to ensure isolation and QoS provisioning [42, 96, 97, 99] (e.g., cache management, bandwidth aggregation, I/O prioritization). *Data management* assembles services oriented to the management of data requests, such as data reduction [26, 66], security [75], and redundancy [8, 18]. *Data routing* primitives encompass routing mechanisms that redefine the data flow, such as I/O path customization [96], replica placement strategies [109], and data staging [35]. Even though mainly applied in networking contexts [50], *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 [96].

As SDS systems are employed over different storage scenarios, data planes can include additional properties (e.g., dependability, simplicity, generality) that portray other aspects of SDS [48, 99]. However, as they are not covered by the majority of systems, they are not contemplated as part of this survey's taxonomy. One such property is dependability, which refers to the ability of a data plane to ensure availability and tolerate faults of the storage services implemented at stages, regardless of employing performance management, data management, or data routing objectives [26, 99].

2.1.2 Stage Design. We now categorize data plane stages in three main designs. Each design respects the internal organization of a stage's enforcement structure, regardless of being applicable at different points of the I/O path. Figure 3 illustrates such designs, namely *(a) Stack-*, *(b) Queue-*,

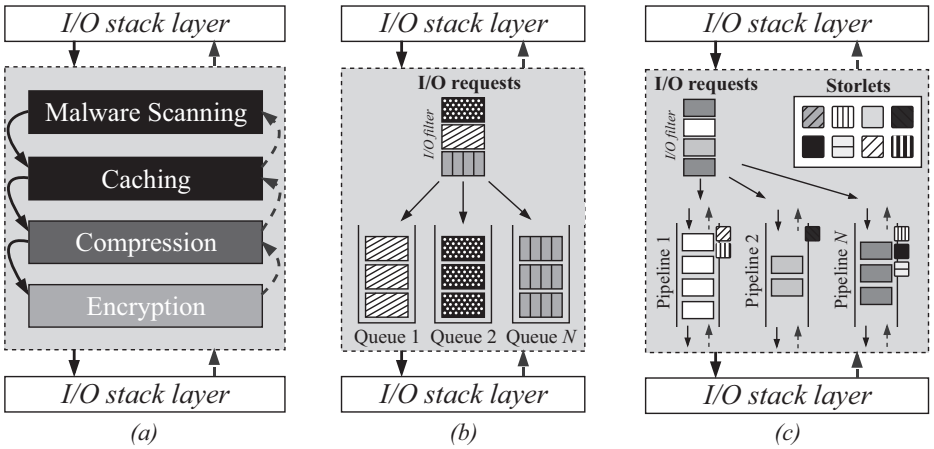

Fig. 3. SDS data plane stage architectures, namely *(a)* Stack-, *(b)* Queue-, and *(c)* Storlet-based designs.

and *(c) Storlet-based* data plane stages. Similarly to Section 2.1.1, these designs are contemplated as part of the taxonomy for classifying SDS data elements.

Stack-based stages. *Stack-based* data plane stages provide an interoperable layer design where layers correspond to specific storage features and are *stacked* according to installed policies [75]. This design enables an independent and straightforward layer development, introducing a modular and programmable storage environment. Layer organization is established by the control plane and may result in a number of stacking configurations tuned to satisfy the installed policies and attend the I/O requirements. Figure 3*(a)* depicts an abstract design of a *stack-based* data plane stage. It comprehends a four-layer stacking configuration, each with a specific storage service 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 requests traverse all layers in an orderly way, such that each layer only receives requests from the layer immediately on top and only issues requests to the layer immediately below. SafeFS [75], for example, provides a framework for developing stackable storage services by repurposing existing FUSE-based file system implementations to employ different storage objectives over I/O flows, such as encryption and erasure coding.

Stacking flexibility is key to efficiently reusing layers and adapting to different storage environments. However, this vertical alignment may limit the ability to enforce specific storage policies (e.g., data routing), limiting the available policy spectrum exposed to the control plane. Current stack-based 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 [66, 75].

Queue-based stages. *Queue-based* data plane stages provide a multi-queue storage environment, where queues are organized to employ distinct storage functionalities over I/O requests [99]. Each queue is a programmable storage element that comprehends a set of rules 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 requests to other points of the I/O path. Examples of such properties include the use of token-buckets, priority queues, and scheduling mechanisms (further detailed in Section 3.3). Figure 3*(b)* depicts the design of a *queue-based* data plane stage. Incoming requests are inspected, filtered, and assigned to the corresponding queue. For instance, IOFlow [99] provides programmable queues to employ performance and routing services over virtual instances and storage servers, enabling end-to-end differentiation and prioritization of I/O flows.

The flexible design of the queuing mechanism allows for simplified orchestration, flexibility, and modularity of data plane stages [99], although, as demonstrated by complementary research fields [27, 29], queue structures are primarily used to serve performance-oriented policies. As such, trading customization over a more tailored design turns the integration and extension of alternative storage services into a challenging endeavor.

Storlet-based stages. *Storlet-based* data plane stages abstract storage functionalities into programmable storage objects (*storlets*) [26, 87]. Leveraging from the principles of active storage [79] and *OpenStack Swift Storlets* [71], 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 reutilization of existing programmable objects. Figure 3*(c)* depicts the abstract architecture of a *storlet-based* data plane stage. Stages comprehend a set of preinstalled storlets that comply with initial storage policies. Policies and storlets are kept up-to-date to ensure consistent actions over requests. 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. Crystal [26], for example, provides a storlet-enabled data plane that allows system designers to deploy and run customized performance and data management services over I/O requests.

The seamless development of storlets ensures a programmable, extensible, and reusable SDS environment. However, as the policy scope increases, it becomes harder to efficiently manage the storlets' mechanisms at 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, SDS control planes provide a logically centralized controller with system-wide visibility that orchestrates a number of data plane instances. It shares a unified point of control, easing both application building and control logic development. However, even though identical in principle, 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 novel functionality to employ over I/O flows cannot be arbitrarily assigned to stages, since it may introduce significant performance penalties and 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 functionalities, which are sensitive to I/O processing along the I/O path, but also data management ones, which entail additional computation directly impacting processing and propagation time of I/O flows. Thus, controllers require performing 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 services and policies is broader than in SDN, ensuring transparent control and policy specification introduces increased complexity to the design of controllers (e.g., decision making, service placement).

As depicted in Figure 1, controllers are the midtier of an SDS system and provide the building blocks for orchestrating data plane stages according to the actions of control applications built on top. Despite being 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 holistically (i.e., encompassing the global storage environment) [26, 35, 96, 99, 109]. Unless otherwise stated, a *controller* defines a logically centralized component, even though

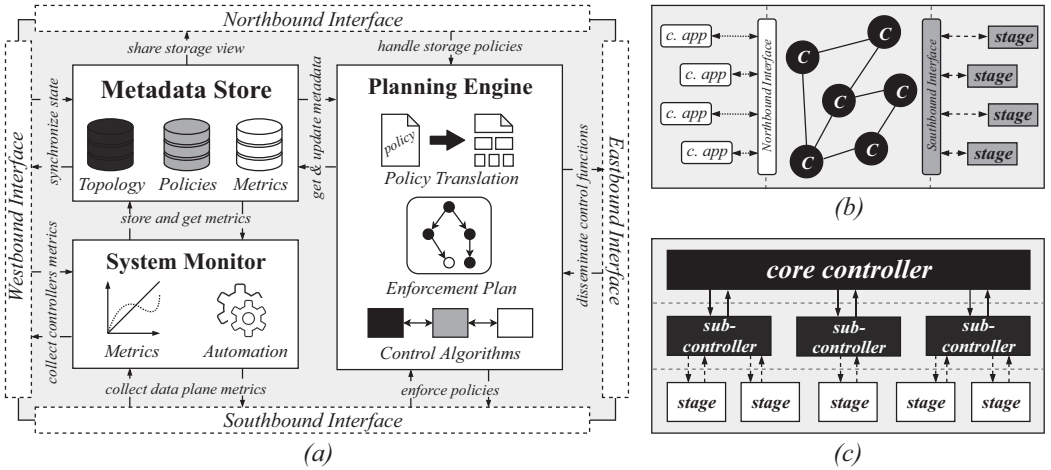

Fig. 4. SDS controllers' architecture: *(a)* presents the organization of the internals of an SDS controller, and *(b)* and *(c)* depict the design of SDS control plane controllers, namely *Flat* and *Hierarchical* designs.

it is physically distributed. A controller can be partitioned into three functional modules, namely a *metadata store*, a *system monitor*, and a *planning engine*, as illustrated in Figure 4*(a)*. Each of these modules consists of a particular set of control features shared between controllers or designed for a specific control device. Moreover, these modules are programmable and allow SDS users to install, at runtime, custom control actions to employ over the system (e.g., control algorithms for accurate policy enforcement, collection of monitoring metrics).

The *metadata store* holds 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 instances, namely *topology*, *policies*, or *metrics*. The *topology* instance maintains a stage-level topology graph that comprehends the distribution of SDS stages, along the assigned storage services, and information about the resources of each node in which a stage is deployed (e.g., available storage space, CPU usage). For instance, sRoute's [96] controller maintains an up-to-date topology graph with the capacity of physical resources and shares it with control applications to define accurate storage policies. The *policies* instance holds storage policies submitted by 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, which are used to adapt data plane stages to meet applications' requirements. Further, to ensure a dependable SDS environment, metadata is usually synchronized among controllers [96, 99].

The *system monitor* collects, aggregates, and transforms unstructured storage metrics and statistics into general and valuable monitoring data [26, 32, 109]. It captures relevant properties of the physical storage environment (e.g., device and network 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 that allow the controller to optimize the infrastructure by (re)configuring SDS stages and assist other modules in policy enforcement and feature placement activities. For example, Mirador [109] collects device and network load and traces workload profiles to build an accurate model of the infrastructure, in order to assist in flow customization and data placement enforcement. Furthermore, the *system monitor* can also exercise automation operations without input of applications, ranging from simple management activities

such as increasing or decreasing the number of controllers to more complex tasks (e.g., fine-tuning storage policies, reconfiguring data plane stages) [26].

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 different control algorithms and strategies that specify how the data plane handles I/O flows and define the most suitable place for policies to be enforced [99]. Examples of such control algorithms and control strategies include proportional sharing, isolation and priority, feedback control, and performance modeling (further detail in Section 3.2). 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 a number of stages to perform distributed enforcement, such as bandwidth aggregation and I/O prioritization [97, 99].

To provide a seamless integration with the remaining SDS layers, the controller connects to a *northbound* and a *southbound interface* to interact with control applications and data plane stages, respectively. Similar to other software-defined approaches, SDS architectures comprehend a network of controllers connected through a *westbound/eastbound interface*, as illustrated in Figures 1 and 4*(a)*. This interface defines the instruction set and communication protocols between controllers being used to exchange data, synchronization, and fault tolerance; monitor; and, depending on the control plane architecture, assign control tasks to other controllers (further description of this property is presented in Section 2.2.2). Further, this interface aims at achieving interoperability between different controllers [50]; however, despite its clear position in the SDS environment, current literature does not explore nor provide details about such an interface.

2.2.1 Properties. Similarly to other software-defined approaches, designing and implementing production-quality SDS systems requires solving important challenges at the control plane [48]. 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 (Section 3).

Scalability. Scalability refers to the ability of a control plane to efficiently orchestrate and monitor a number of data plane stages. Similarly to SDN, the control plane can be either physically centralized or distributed [7]. A physically centralized control plane consists of a single SDS controller that orchestrates the overall storage infrastructure, which is an attractive design choice in terms of simplicity [69, 92, 101]. However, physical control centralization 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 stages increases, so does the control traffic destined toward the centralized controller, bounding the 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 [50].

Production-grade SDS controllers must be designed to attend to the scalability, performance, and dependability requirements of today's production storage systems, meaning that any limitations should be inherent to the storage infrastructure and not from the actual SDS implementation. Thus, physically distributed controllers can be scaled up to attend to such requirements. While sharing a logically centralized service, multiple interconnected SDS controllers orchestrate the storage infrastructure by sharing control responsibility, and thus alleviating overall control load. Leveraging from existing classifications [7], distributed SDS controllers can follow a *flat* or a *hierarchical* distribution (Section 2.2.2). Flat designs (Figure 4*(b)*) imply horizontal control partitioning to provide a replicated control service, forming a reliable, fault-tolerant, highly available

cluster of controllers [26, 99, 109]. On the other hand, hierarchy-based designs (Figure 4*(c)*) imply the vertical control partitioning to provide a scalable and highly performing SDS control plane [42, 96].

Dependability. Dependability refers to the ability of a control plane to ensure sustained availability, resiliency, and fault tolerance of the control service [6]. Physically centralized controllers represent a single point of failure (SPOF), 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 orchestrating the data plane tier. The system should handle failures gracefully, avoiding SPOF and enabling fault tolerance mechanisms. Thus, similarly to SDN [11, 16, 48], physically distributed SDS solutions provide coordination facilities for detecting and recovering from control instance failures [26, 96, 109]. 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 tradeoffs of performance, scalability, and state consistency 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 [51, 70], or simply implement a primary backup approach where one main controller orchestrates all data plane elements while remaining control instances are used for replication of the control service [15].

While some solutions comprehend a strong consistency model to ensure correctness and robustness of the control service, others resort to relaxed models where each controller is assigned to a subset of the storage domain and holds a different view of the infrastructure. Regarding control distribution, flat control planes are designed to ensure sustained resilience and availability [26, 99], while hierarchical control planes focus on the scalability challenges of the SDS environment [35, 96].

Depending on the storage context, the dependability offered by the control plane can be coupled to a specific I/O layer. Specifically, as some SDS systems are directly implemented over existing storage systems, such as Ceph (e.g., Mantle [86], SuperCell [101]) and OpenStack (e.g., Crystal [26]), the control plane's dependability is bounded by the dependability of the respective storage system.

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 stages to experience volatile workloads [3, 19, 99]. Moreover, designing heterogeneity-oblivious SDS systems with monolithic and homogeneous configurations can severely impact the storage ecosystem, hindering the ability to accurately enforce policies [26].

Therefore, SDS controllers must comprehend a self-adaptive design, capable of dynamically adjusting their storage artifacts (e.g., policy values, data plane stage configurations) to the surrounding environment, in order to deliver responsive and accurate enforcement decisions [26]. As enforcement strategies directly impact I/O flows, employing self-adaptive and autonomous mechanisms over SDS controllers brings a more accurate and dynamic enforcement service. Moreover, due to the fast changing requirements of the storage environment, data plane configurations rapidly become subpar, and thus, automated optimizations of data plane resources are key to ensure efficient policy enforcement and resource usage.

Current strategies to provide adaptable SDS control include control-theoretic mechanisms, such as *feedback controllers* that orchestrate system state based on continuous monitoring and data plane tuning [43, 59, 99], and *performance modeling*, such as heuristics [108], linear programming [66,

120], and machine learning [42] techniques (further detail on these strategies is presented in Section 3.2).

2.2.2 Controller Distribution. SDS literature classifies the distribution of controllers as logically centralized, despite being physically distributed for the obvious reasons of scale and resilience [96, 99]. We now categorize distributed control planes regarding controller distribution. Figure 4 illustrates such designs, namely *(b) Flat* and *(c) Hierarchical* controllers. Similarly to the properties presented in Section 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. Depending on the control plane's implementation, controllers may hold different organizations, being designed to account for the different tradeoffs 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 handles all stage-related events (e.g., collect reports and metrics), disseminates policies, generates comprehensive enforcement plans, and enforces policies. Moreover, the control plane provides the coordination facilities to ensure fault tolerance and strong consistency by relying on Paxos-based mechanisms [96, 109] or simple primary backup strategies [15]. Mirador [109] follows such an approach by resorting to a coordination service to ensure a highly available control environment [34]. This design allows distributed controllers to have strong consistency properties, ensure high availability of the control service, and ease 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 [99].

Other solutions, as depicted in Figure 4*(b)*, may provide a network-like flat control platform, where each controller is responsible for a subset of the data plane elements [26, 48]. Namely, each controller orchestrates a different part of the infrastructure, synchronizing its state with the remaining controllers with strong or eventual consistency mechanisms. Upon the failure of a controller, another may assume its responsibilities until it becomes available. For instance, Crystal [26] holds a set of autonomous controllers, each running a separate control algorithm to enforce different points of the storage stack. This network-like 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, this control model hardens the control plane's ability to share a logical centralized setup and up-to-date visibility to control applications, hindering its applicability to large-scale production storage infrastructures. Further, with the emergence of novel computing paradigms composed by thousands of nodes, such as serverless cloud computing [40] and Exascale computing [22], this design may have 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 [42, 96]. To limit the load of the centralized controller, both control and management flows must be handled closer to data plane resources and minimized as much as possible without compromising system correctness. Thus, similarly to distributed SDN controllers [36, 113, 114], hierarchical control plane distributions address such a problem by organizing SDS controllers in a hierarchical disposition [35, 42, 96]. Controllers are hierarchically ranked and grouped by control levels, each of them with respect to a cumulative set of control services. This approach distributes the control

responsibility to alleviate the load imposed over centralized services, 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 *subcontrollers*. *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 SDS environment, *core controllers* manage control applications and orchestrate both data plane and *subcontroller* elements. Moreover, *core controllers* share part of their responsibilities with underlying control tiers, propagating the control fragments hierarchically. *Subcontrollers* 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 data plane, as well as the control activities that do not require global knowledge or impact the overall state of the control environment. For example, Clarisse [35] implements a hierarchical control plane over HPC infrastructures that groups control activity through global, application, and node controllers. Since *core controllers* hold global visibility, they perform accurate and holistic control decisions over the SDS environment. However, maintaining a consistent view is costly, causing significant performance overhead even when performing simple and local decisions [35, 96]. On the other hand, *subcontrollers* are tailored for control-specific operations, providing faster and fine-grained local decisions over SDS stages. In case a *subcontroller* 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 control directives of I/O flows. Applications exercise direct control over controllers by defining the control logic through policies and control algorithms, which are further translated into fine-grained stage-specific rules to be employed over I/O requests. Examples of control algorithms include proportional sharing, prioritization and isolation, and shares and reservations, each of which is further detailed in Section 3.2. Similar to other software-defined approaches [38, 50], control applications introduce a *specification* abstraction into the SDS environment, 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 an accurate, consistent, and efficient policy creation.

Existing control applications are designed for a variety of storage contexts and cover a wide array of functionalities, including *performance*, *resource* and *data management*, *security*, and *other storage objectives*. *Performance* objectives aim at enforcing performance guarantees (e.g., throughput and latency SLOs [54]), prioritization (e.g., bandwidth allocation according to applications' priority [99]), and performance control (e.g., I/O isolation). On the management side, resource-centric objectives enforce fairness between applications accessing shared storage systems, as well as caching and device management policies (e.g., caching schemes [97], storage quotas [18]) and I/O flow customization (e.g., modify the I/O endpoints of a layer [96]). Data-centric objectives, on the other hand, enforce objectives directly applicable over data and metadata such as data redundancy, data reduction, data placement, and (meta)data organization [88]. *Security*-based objectives enforce encryption and malware scanning rules to ensure privacy and confidentiality of sensitive data. *Other storage objectives* such as energy efficiency and elasticity control seek to provide additional properties to storage systems. Table 1 classifies existing SDS control applications regarding storage objectives, organized by storage infrastructure.

Table 1. Classification of SDS Control Applications Regarding Storage Objectives

		Cloud	HPC	Application-Specific
Performance	*Performance guarantees*	[54, 92, 96, 97, 99, 117, 120]	[42]	[26, 43, 60, 64, 105]
	Prioritization	[54, 97–99, 108, 117, 119, 120]	[35, 42]	[60, 64, 105]
	Performance control	[59, 66, 91, 97–99, 119]	[42]	[26, 60, 87, 101, 105]
Resource Management	*Fairness*	[92]		[60, 64]
	Cache management	[96, 97]		[26, 86]
	Device management	[66]		[18, 101]
	Flow customization	[59, 96, 108]	[35]	[8, 109]
Data Management	*Data redundancy*	[77]		[8, 18, 75]
	Data reduction	[66, 77]		[18, 26]
	Data placement	[66, 69, 77, 96]	[35]	[8, 86, 87, 101, 109]
	(Meta)Data organization	[69, 77]		[8, 86, 87]
Security	*Encryption*	[77]		[75]
	Malware scanning	[99]		
Other Storage Objectives		[66, 69, 77]	[35]	[8, 75, 87]

Finally, a *Northbound interface* connects control applications and controllers by abstracting the distributed control environment into a language-specific communication interface, hiding unnecessary infrastructure details while allowing straightforward application building and policy specification. Such a design fosters the integration and reutilization of different control applications between SDS technologies, enabling an interoperable control design. However, current work on SDS lacks a standard *Northbound interface*, which limits the ability to combine different control applications throughout distinct storage contexts and SDS technologies.

3 SURVEY OF SOFTWARE-DEFINED STORAGE SYSTEMS

This section presents an overview of existing SDS systems regarding storage infrastructure (Section 3.1), control strategy (Section 3.2), and enforcement strategy (Section 3.3). Section 3.4 discusses key differences between SDS systems. Systems are classified according to the taxonomies described in Section 2.1 and Section 2.2.

3.1 Survey by Infrastructure

Storage infrastructures have different requirements and restrictions, and thus, the design and combination of SDS properties may vary significantly with the storage type being targeted. To provide a comprehensive survey of SDS systems, we describe them in a twofold way. Table 2 classifies SDS systems according to the taxonomy described in Section 2 while grouping them by storage infrastructure, namely cloud, HPC, and application-specific storage stacks. This table highlights the design space of each infrastructure and depicts current trends and unexplored aspects of the paradigm that require further investigation. Then, the textual description presented in each section (Sections 3.1.1 through Section 3.1.3) draws focus on the environment and context where each system is applied, as well as the enforced storage objectives and other aspects that differentiate these solutions. The classification considers systems from both academia and industry.[1] Commercial solutions whose specification is not publicly disclosed are not considered. Systems that follow the

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

Table 2. 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] [99]	*F*	◐	●	◐	*Q*	◐	◐	*MP*	○	*PR*	↑↓
Moirai [97]	*F*	◐	●	◐	*S*	◐	◐	*MP*	●	*P*	↓
sRoute [96]	*H*	●	●	◐	*Q*	●	◐	*MP*	○	*PR*	↑↔↓
JoiNS [108]	*H*	●	-·-	◐	*Q*	◐	○	*DSP*	○	*PR*	↔↓
Façade⋆ [59], *AVATAR*⋆ [117]	*C*	○	○	◐	*Q*	◐	○	*SP*	●	*P*	↓
Pisces⋆ [92], *Libra*⋆ [91]	*C*	○	○	◐	*Q*	○	○	*DSP*	○	*P*	↓
PM⋆ [120], *WC*⋆ [119]	*F*	◐	●	●	*Q*	○	○	*DSP*	○	*P*	↓
PSLO⋆ [54]	*C*	○	○	●	*Q*	◐	○	*DSP*	○	*P*	↑↓
Wisp⋆ [98]	*D*	●	-·-	◐	*Q*	◐	○	*DSP*	●	*P*	↔↓
Wiera-Tiera⋆ [69, 77]	*C*	○	○	◐	*St*	●	●	*DSP*	●	*PDR*	↓
flexStore [66]	*F*	◐	●	◐	*S*	◐	○	*DSP*	◐	*PD*	↑↓
Clarisse [35]	*H*	●	◐	○	*Q*	◐	○	*DSP*	○	*PR*	↑↔↓
SIREN [42]	*H*	●	◐	◐	*Q*	◐	○	*MP*	○	*P*	↔↓
Retro [60]	*F*	◐	●	◐	*Q*	◐	◐	*MP*	○	*P*	↑↓
Cake⋆ [105]	*C*	○	○	◐	*Q*	◐	○	*MP*	●	*P*	↑↓
Crystal [26]	*F*	◐	●	◐	*St*	●	●	*DSP*	◐	*PD*	↑↔↓
Coho Data [i] [18, 109]	*F*	◐	●	◐	*S*	◐	○	*DSP*	●	*PDR*	↑↓
Mantle [86]	*C*	○	○	◐	*St*	●	○	*DSP*	○	*P*	↑↓
SuperCell [101]	*C*	○	○	◐	*St*	○	○	*DSP*	●	*PD*	↓
Malacology [87]	—	—	—	—	*St*	●	◐	*DSP*	○	*PDR*	↓
SafeFS [75]	—	—	—	—	*S*	◐	●	*SP*	●	*PD*	↓
Triage⋆ [43]	*F*	◐	●	●	*Q*	○	○	*DSP*	●	*P*	↓
PADS ⋆ [8]	—	○	○	○	*St*	●	●	*DSP*	○	*PR*	↑↓
Mesnier et al.⋆ [64]	—	○	○	○	—	◐	◐	*SP*	○	*P*	↓

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

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

Design
S-Stack
Q-Queue
St-Storlet

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

Scope
P-Performance
D-Data
R-Routing

Interfaces
↑ Northbound
↔ West/Eastbound
↓ Southbound

SDS design principles and storage functionalities described in Section 2, targeting at least one of the planes of functionality, are contemplated in this classification.[2]

3.1.1 Cloud Infrastructures. Cloud computing infrastructures 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. Cloud premises consist of hundreds to thousands

[2]Systems not originally defined as SDS but that follow the same design principles are marked with ⋆ in the classification table.

of compute and storage servers. Compute servers are virtualized and abstract multiple physical servers into an infinite pool of resources exposed through virtual machines (VMs) or containers. Resources are shared between tenants and mediated by hypervisors. Storage servers accommodate heterogeneous storage systems and devices with distinct levels of granularity and performance. These servers persist all data and are exposed to VMs as virtual devices. Compute and storage servers are connected through high-speed network links that carry all infrastructure traffic. However, behind this virtualized environment lie complex and predefined I/O stacks, which result in increased end-to-end complexity and difficulty in enforcing storage requirements and end-to-end control over storage resources [28, 99]. While several systems have been proposed to partially address this problem (e.g., QoS provisioning and scheduling [13, 27–29, 112]), none have considered end-to-end enforcement nor holistic orchestration of infrastructure resources. To address such shortcomings, SDS-enabled systems have moved toward cloud infrastructures.

The term Software-Defined Storage was first introduced by IOFlow [99]. Specifically, IOFlow enables end-to-end policy enforcement under multi-tenant architectures. Queue-based stages employ performance and routing primitives from VMs to shared storage, allowing efficient performance isolation, differentiated I/O treatment, and end-to-end performance control. A reactive flat-based control plane discovers stages and dynamically configures their storage services to attend the manifold objectives of applications built on top (e.g., bandwidth aggregation, prioritization, throughput, and latency objectives). While originally designed to enforce policies over storage-related layers, it was later extended to support caching and networking [96, 97]. Moirai [97] extends IOFlow's design to exercise direct and coordinated control over the distributed caching infrastructure to improve resource utilization and achieve performance isolation and QoS guarantees. Stages are deployed as stackable and programmable caching instances to employ performance services over incoming I/O requests, such as workload aggregation and maximization of cache hit ratios. At the control plane, a logically centralized controller built on top of IOFlow's traffic classification mechanism orchestrates stages in holistic fashion and maintains (cache) consistency and coherence across the I/O stack. Further, it continuously monitors the infrastructure and maintains key performance metrics of each workload running on the system (e.g., throughput, read-write proportion, *hit ratio* curves).

To override the rigid and predefined I/O path of cloud infrastructures, sRoute [96] goes toward combining storage and networking primitives. It extends IOFlow's design to employ routing services throughout the I/O path, turning the storage stack more programmable and dynamic. The data plane consists of programmable switches (*sSwitch* stages), which provide flow regulation and customization, and queue-based stages, which implement performance management and I/O differentiation at hypervisors. Such a design allows I/O flows to be redirected to any point of the I/O stack (e.g., controller, *sSwitch-enabled* stages). The control plane holds a hierarchical distribution made of a centralized controller and several *control delegates*, which are restricted control daemons installed at *sSwitches* for control plane efficiency. Each of these *delegates* performs decisions locally, alleviating the load of the centralized component, and thus providing a more scalable environment. Similarly, JoiNS [108] orchestrates storage and network primitives over networked storage premises. While sharing similar control principles, JoiNS leverages from existing SDN data planes [63] and programmable switches to enforce routing primitives over the storage infrastructure, while storage stages implement predetermined performance features over block device drivers.

While IOFlow-enabled systems are designed to achieve end-to-end optimization in cloud storage, several works address specific problems and layers of the I/O stack in an SDS fashion. Façade [59] and AVATAR [117] propose a virtualization layer that seats between clients and the storage utility of shared storage systems and enforces throughput and latency objectives in the

presence of bursty and volatile workloads. Even though not physically decoupled, Façade provides a centralized controller that employs a non-linear feedback loop to allocate storage shares for each workload and adjust data stages according to targeted workload latencies, and a queue-based data plane that governs the queue depth of a storage utility. On the other hand, AVATAR proposes a two-level scheduling framework that enforces 95th-percentile latency objectives. At a high level, a centralized controller orchestrates per-workload FIFO queues and regulates workflows to achieve isolation, while at a lower level, a queue-based data plane rate limits requests before dispatching them to the storage utility. At multi-tenant cloud environments, Pisces [92] and Libra [91] provide system-wide performance isolation and fair resource allocation. In both systems, control and data are not physically decoupled. Specifically, in Pisces, a centralized controller provides per-tenant weighted fair shares to enforce throughput objectives, while queue-based stages schedule per-tenant rules over network resources of storage servers. On the other hand, while sharing the same design primitives as Pisces, Libra's stages enforce per-tenant app-request reservations over low-latency SSD drives.

Despite providing isolation and fairness over network and storage resources, previous systems focus on sharing storage bandwidth, which is simpler to control than tail latency, as bandwidth is an average over time not affected by the I/O path's cumulative interactions. Moreover, single resource enforcement either over storage [59, 91, 117] or network [92] limits the ability to enforce end-to-end storage policies. This led to the design of multi-resource SDS systems [119, 120]. PriorityMeister (PM) [120] combines prioritization and rate limiting over network and storage resources to meet tail latency at the 99.9th and 99.99th percentiles. A proactive flat-based controller automatically orchestrates the data plane under varying degrees of workload burstiness, while queue-based stages deployed over storage and network devices provide per-workload latency differentiation. Each stage consists of multiple rate limiting queues that, while improving burstiness, introduce increased computation time and number of required computing servers. To address this, WorkloadCompactor (WC) [119] extends the design of PM to consolidate multiple workloads onto a storage server. WC's controller automatically selects rate limiting and priority profiles, enforcing them over storage and the network to minimize the number of instances that cloud providers use to serve all workloads. While enforcing high-latency percentile objectives, PM and WC cannot simultaneously serve throughput-based services. As such, PSLO [54] provides an efficient storage environment that simultaneously enforces tail latency and throughput objectives over consolidated VMs under shared storage infrastructures. Deployed at the hypervisor level, PSLO holds a centralized controller and queue-based stages that employ an integral feedback control loop combined with linear programming models to govern the arrival rate of I/O requests in a per-VM basis.

Multi-tenant systems composed by hundreds of small partitioned services (e.g., Service-Oriented Architectures (SOAs)) are often used on cloud premises to build large-scale web applications [98]. These systems consist of fine-grained and loosely coupled services, each running on a physical or virtual machine. However, their limited visibility hinders the ability to provide efficient storage enforcement. Wisp [98] proposes a distributed framework for building efficient and programmable SOAs that adapt storage resources under multi-tenancy. It provides a fully decentralized design, where each SOA accommodates an SDS tuple made of a controller and data plane stages. Each controller gathers local information and propagates it to other peers to execute distributed control algorithms, while queue stages enforce local performance services over SOA resources.

Cloud providers offer a wide array of storage services with tradeoffs in performance, cost, and durability, leading applications to opt for simplicity instead of resorting to different services with conflicting properties. The Wiera-Tiera SDS system provides a geo-distributed cloud environment that facilitates the use and specification of multi-tiered storage across data centers. At the data

plane, Tiera [77] provides a programmable storlet-based middleware that encapsulates and repurposes existing services into an optimized interface that can be glued to comply with data management and routing policies (e.g., encryption, compression, data placement). At the control plane, Wiera [69] provides a centralized controller that enforces global storage policies across multitiered data centers. It allows the combination of storage features available at different tiers of the cloud storage hierarchy, enabling the creation of new services via composition.

Storage features besides QoS provisioning and performance isolation can be also explored in an SDS fashion. For instance, flexStore provides a framework for dynamically adapting a data center to cope with QoS and energy consumption objectives. At the data plane, stack-based stages are employed over storage systems (e.g., object stores) and hypervisors. System-level stages adjust the data layout of storage devices and collect performance metrics to enforce energy-related policies (e.g., reduce number of storage devices), while hypervisor-level stages employ performance and data management services, such as deduplication and caching management. On the control side, a flat-based controller that leverages from linear programming models enforces QoS and energy-related policies under multi-tenancy by managing the life cycle of dedicated storage volumes and allocating them to VMs.

Cloud-based SDS has become an active research topic for improving overall performance and resource efficiency of cloud storage infrastructures. Enterprise-grade systems such as VMware Cloud [103] and Microsoft Windows Server [65] have been paving the way of the paradigm in industry, fostering its adoption at a global scale. Moreover, while several storage subsystems have been proposed to address storage-related resources of cloud premises, such as proportional sharing [27, 29, 44] and I/O scheduling [13, 28, 57, 106, 112], these can now be repurposed as control algorithms or applications to be used in existing SDS systems.

3.1.2 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 [100]. Indeed, supercomputers are the cornerstone of scientific computing and the *de facto* premises for running compute-intensive applications. Modern infrastructures are composed by compute and storage nodes. Compute nodes perform computational-related tasks through manycore processors that deliver massive parallelism and vectorization. Storage nodes persist applications' data in a shared Petabyte-scale parallel file system (e.g., Lustre [84], GPFS [82]) that offers high-performance storage and archival access on top of hundreds of storage drives. Communication across nodes is made through specialized high-performance interconnects. Furthermore, many of the current *Top500* supercomputers comprehend a third group of nodes, namely I/O forwarding nodes (or I/O nodes), that act as a middleware between compute and storage instances and are responsible for receiving compute nodes' requests and forwarding them to storage ones [14, 100]. 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).

The long and complex I/O path of HPC infrastructures makes performance isolation, end-to-end control of I/O flows, and I/O optimizations increasingly challenging [111]. The variety of access patterns exhibited by applications has led HPC clusters to observe high levels of I/O interference and performance degradation, inhibiting their ability to achieve predictable and controlled I/O performance [58, 115]. While several efforts were made to prevent I/O contention and performance degradation (e.g., QoS provisioning [110, 118], job scheduling optimization [39, 95]), none 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 [35] provides the building blocks for designing coordinated system-wide cross-layer mechanisms, such as parallel I/O scheduling, load balancing, and elastic collective I/O. A queue-based data plane stages data between applications and storage nodes and implements performance and routing mechanisms for transferring data between compute and storage nodes at the middleware layer (e.g., MPI-IO). A hierarchically distributed control plane offers the mechanisms for coordinating and controlling routing-related activities. Controllers are hierarchically deployed over compute premises and perform different levels of control and enforcement. Similarly, SIREN [42] enforces end-to-end performance objectives by dynamically allocating resources according to applications' demands. 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. A hierarchy-based controller efficiently enforces performance objectives over managed resources, while data plane stages deployed throughout the I/O stack (e.g., request schedulers, parallel file systems) dispatch I/O requests through a queue-based structure that enforces the reservations and shares specified by control instances.

The recent efforts on designing and implementing SDS-enabled HPC infrastructures have proven its utility and feasibility on high-performance technologies. As we move closer to the Exascale era [22], the adoption of the SDS paradigm 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 other infrastructures, HPC premises contain different requirements in terms of architecture and hardware, turning unfeasible the applicability of non-HPC-based SDS systems over such environments. First, HPC storage backends are generally composed by a shared file system [84], which becomes a major performance bottleneck when concurrently used by hundreds to thousands of applications competing for shared resources, leading to high levels of I/O interference and performance degradation [76, 111]. Second, HPC applications generate complex workflows (e.g., scientific simulations, real-time visualizations), which translate into different storage objectives and services to be employed over I/O flows [35].

3.1.3 Application-specific Infrastructures. Application-specific infrastructures are storage stacks built from the ground up, designed for specialized storage and processing purposes to achieve application-specific I/O optimizations [87]. Production-grade clusters include multi-tenant distributed storage systems such as Hadoop [93], Ceph [107], and OpenStack Swift, being mainly composed by proxy and storage servers. Proxy servers map application requests to the respective data location and provide global infrastructure visibility, system-wide management activities (e.g., load balancing, lease management), and high availability. Storage servers are user-space daemons that persist applications' data. While these systems are built to run on commodity hardware, enterprise-grade infrastructures may hold hundreds to thousands of storage servers interconnected with dedicated network links. Each server accommodates several multicore processors and storage drives hierarchically organized. Such a specialized environment leads to hard-coded designs and predefined I/O stacks, making the programmability of such systems challenging [87]. Further, the absence of performance guarantees and isolation leads to greedy tenants and background tasks (e.g., garbage collection, replication) consuming a large quota of resources, impacting the overall system performance [60, 62]. While several mechanisms have been proposed to address different system intricacies (e.g., workload awareness, availability), none have considered end-to-end enforcement of storage policies or improved programmability of specialized stacks. As such, several SDS-enabled systems have been proposed to address such challenges. Even though these 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, we classify these in a separate category.

The requirements of isolation and fairness in distributed storage systems have led researchers to shift from hard-coded single-purpose implementations to software-defined approaches. Retro [60] enforces performance guarantees and fairness over multi-tenant Hadoop stacks by identifying and rate limiting workflows bottlenecking shared resources. A reactive flat-based controller enforces fine-tuned policies in the presence of bursty and volatile workloads, while queue-based stages abstract arbitrary system resources (e.g., storage devices, CPU, thread pools) and employ performance management features over priority queues, fair schedulers, and token-buckets implemented along the I/O path. Further, Cake [105] introduces end-to-end enforcement of 99th-percentile latency objectives over Hadoop storage stacks. Similarly to AVATAR [117], Cake proposes a two-level scheduling framework, where a centralized controller continuously monitors latency performance and orchestrates queue-based stages to perform per-tenant prioritization through proportional sharing and reservations. Stages are deployed at RPC layers, providing differentiated scheduling of I/O requests and enabling multi-resource control throughout the storage stack.

While these systems focus on the performance and resource management of Hadoop stacks, others focus on multi-tenant object stores. Crystal [26] provides an SDS-enabled object store that supports resource sharing and isolation in the presence of heterogeneous workloads. Implemented over the *OpenStack Swift* [71], a storlet-based data plane injects user-defined services over I/O flows, such as compression, caching, encryption, and bandwidth control. At the control plane, flat-based controllers dynamically adapt stages according to tenants' requirements. Controllers are twofold, divided into *global controllers* with system-wide visibility that continuously control, monitor, and disseminate storage policies to data stages and other controllers, and *automation controllers* with limited visibility that enforce dedicated control actions over selected points of the I/O path.

Commercial storage systems have also experienced a thrust toward the software-defined domain. For instance, Coho Data [18, 109] proposes an SDS 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 [109] provides a flat-based dynamic storage placement service that orchestrates heterogeneous scale-out storage systems. To enforce routing and data management activities, the control plane continuously collects resource metrics and workload profiles of the cluster and uses solvers to calculate enforcement plans. At the data plane, Strata [18] implements a stack-based network-attached object store that manages high-performance storage devices under multi-tenancy. Stages are deployed over both SDN-enabled switches, for flow customization and data placement, and PCIe flash devices, to employ striping, replication, and deduplication over I/O requests.

As several systems 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 [87, 88, 101]. For instance, Mantle [86, 88] decouples management-based policies from the storage implementation, allowing users to fine-tune and adapt the storage environment under volatile requirements. At the control plane, a heuristic-based policy engine injects management policies into distributed storage systems, such as Ceph [107], while a storlet-based data plane abstracts underlying storage artifacts through a data management language, allowing users to build flexible and fine-grained policies, such as programmable caching and metadata management. On the other hand, SuperCell [101] relies on the flexibility and availability of Ceph and proposes an SDS-based recommendation engine that measures and provides cluster configurations under varied workload settings. A centralized controller measures workload characteristics (e.g., I/O size, read/write proportion) and generates enforcement strategies tailored to meet users' requirements in a cost-effective manner. At the data plane, SuperCell fine-tunes storage settings and configurations of Ceph deployments at runtime to cope with different performance and data objectives.

Differently, other systems propose novel abstractions and storage features over application-specific stacks [75, 87]. Malacology [87] is a controllerless SDS system that provides novel storage abstractions by exposing and repurposing code-hardened storage artifacts (e.g., resources, services, abstractions) into a more programmable environment. Rather than creating storage systems from the ground up, Malacology encapsulates existing system functionalities into reusable building blocks that enable general-purpose systems to be programmed and adapted into tailored storage applications via composition. Implemented over Ceph, Malacology decouples policies from storage 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. Following these same principles, SafeFS [75] aims at repurposing existing FUSE-based file system implementations into stackable storage services to employ over I/O requests. Specifically, SafeFS provides a flexible and extensible stack-based data plane that abstracts the file system layer to enable the development of POSIX-compliant file systems atop FUSE. Its stackable organization enables layer interoperability and allows system operators to simply stack independent layers to enforce different storage objectives, such as encryption, replication, erasure coding, and caching.

Previous works on application-specific storage have already crossed the path of software-defined principles [8, 43, 64]. Specifically, as a first attempt toward SDS, Triage [43] introduced an adaptive control architecture to enforce throughput and latency objectives over the Lustre parallel file system [84], in the presence of bursty and volatile workloads. Adaptive flat-based controllers orchestrate per-client I/O flows and regulate request queues according to user-defined performance objectives. At the data plane, queue-based stages rate limit requests before dispatching them to Lustre storage servers. Differently, PADS [8] provides a policy-based architecture to ease the development of custom distributed storage systems. Control and data planes are not physically decoupled, and part of the control logic is shared with a storlet-based data plane. Control applications hold routing and blocking policies to define the logic of the correspondent storage system. Routing policies define data flows, while blocking policies specify consistency and durability objectives. 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 systems via composition, by simply defining a set of policies rather than implementing them from the ground up. Further, Mesnier et al. [64] propose a classification architecture to achieve I/O differentiation at the kernel level. As 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 differentiate them at storage servers according to user-defined policies, thus ensuring performance isolation and resource fairness.

The introduction of software-defined principles into application-specific storage infrastructures has led to significant improvements in terms of programmability and resource efficiency. Its design allows users to experience sustained QoS provisioning and performance isolation in multi-tenant settings, instead of the formerly predefined and single-purposed approaches. However, as these infrastructures typically provide a homogeneous I/O stack, employing application-specific SDS systems over the cloud or HPC can be a challenging endeavor, due to their wide array of storage subsystems (that do not operate holistically [35]) and heterogeneous workloads [42, 92].

3.2 Survey by Control Strategy

As SDS systems are employed over different storage contexts, controllers may assume different control strategies to adapt existing services to the specified objectives. We now survey SDS systems regarding control strategy employed at the control plane, namely *feedback control* and *performance*

Table 3. Classification of SDS Systems Regarding Control and Enforcement Strategies

	Control Strategy			Enforcement Strategy			
	Feedback	*Modeling*	*Algorithms*	*T. Bucket*	*Scheduling*	*P. Queues*	*Injection*
IOFlow [99]	*Reactive*		*PS*	●		●	
Moirai [97]	*Reactive*		*S*				
sRoute [96]	*Reactive*		*PS*	●			
JoiNS [108]		*H*				●	
Façade [59], *AVATAR* [117]	*Non-linear*		*PI*		*EDF*		
Pisces [92]			*PS*		*DRF, DRR*		
Libra [91]			*S*		*DDRR*		
PM [120], *WC* [119]	*Proactive*	*LP*	*I*	●		●	
PSLO [54]	*Integral*	*LP*			●		
Tiera [77], *Malacology* [87]							●
Wisp [98]	*Reactive*				*DRF, BRF, EDF, LSTF*		
flexStore [66], *Mirador* [109]		*LP*	*S*				
SIREN [42]		*ML*	*S*		●		
Retro [60]	*Reactive*		*P*	●	*DRF, BRF*	●	
Cake [105]	*Reactive*		*PS*		●		
Crystal [26]	*Reactive*		*P*				●
Mantle [86], *SuperCell* [101]		*H*					●
Triage [43]	*Adaptive*	*LP*	*I*		●		

Modeling. *(H)euristic, (L)inear (P)rogramming, (M)achine (L)earning.* **Algorithms.** *(P)roportional sharing, (I)solation and priority, (S)hares and reservations.*

modeling. Table 3 highlights the control strategies and algorithms used by SDS controllers and depicts the current trends and unexplored aspects of the paradigm.

3.2.1 Feedback Control. Control-theoretic approaches have been widely used to provide sustained storage performance [31]. A feedback-based controller avoids the need for accurate performance modeling by dynamically adjusting I/O workflows to meet different storage objectives. It does so through a control loop, which depends on *input metrics*, *control actions*, and *control intervals*. The controller continuously monitors system metrics (e.g., throughput, latency) and validates them with installed storage policies. In case of policy violation, the controller adjusts data plane stages through control actions, which rely on the enforcement strategy employed at the stage (e.g., adjust arrival rate of I/O, increase queue depth). Monitoring is made periodically in a predefined control interval. Large intervals result in longer unsupervised control periods, leading to policy violations and performance degradation in case of burstiness or volatile workloads. Small intervals lead the controller to react to performance outliers, resulting in fine-grained adjustments that inhibit sustained storage efficiency.

Reactive SDS controllers employed by IOFlow and sRoute continuously collect throughput and latency metrics of different points of the I/O path. For each stage, the controller enforces control actions over a token-bucket that rate limits queues according to max-min fairness algorithms [12], efficiently providing distributed and dynamic enforcement and differentiated I/O treatment. Differently, controllers of Wisp and Crystal rely on throughput-only observations. Wisp rate limits request queues of micro-services according to different scheduling policies, while Crystal observes per-tenant throughput at *OpenStack Swift* nodes and allocates proportional bandwidth shares to ensure performance guarantees. In proportional sharing, processes are assigned with a notion of

weight, and resources are proportionally allocated based on it [104]. Further, Moirai uses average latency and hit ratio curves to adjust the configurations of stack-based caching stages.

Reactive approaches are also used to enforce tail latency objectives over complex storage settings. For instance, Cake provides a two-level scheduling framework that continuously collects latency and resource utilization metrics over distributed storage stacks, and dynamically adjusts per-client queues according to proportional shares and reservations. In these algorithms, shares specify the resource allocation that a certain process receives, while reservations express the lower bound of I/O performance reserved to a process [42]. Similarly, Retro observes per-workflow latency and resource usage and employs control actions over token-buckets, schedulers, and priority queues, according to max-min fair shares. The collection of heterogeneous metrics, along the multiple enforcement points deployed along the I/O path, allows Cake and Retro to differentiate I/O flows and enforce 99th-percentile latency objectives over Hadoop storage stacks.

Nonetheless, while able to enforce different performance objectives over varied storage stacks, reactive controllers cannot sustain efficient performance at high-latency percentiles under bursty workloads, as they experience several policy violations before beginning a new control loop and adjusting stages accordingly [120]. As such, several systems follow a proactive control strategy to enforce 99th-, 99.9th-, and 99.99th-percentile latencies. For example, PM and WC provide a proactive feedback controller that models per-workload worst-case latency and enforces different control actions over multiple rate limiters and priority queues, according to isolation and priority rules. Each stage consists of per-workload token-buckets and priority queues and efficiently enforces services over network and storage resources.

While these systems are designed to either provide throughput or tail latency objectives, PSLO achieves both by providing an integral feedback controller backed by a forecast model. It continuously monitors per-VM X^{th} percentile latency and throughput and adaptively configures the level of I/O concurrency and arrival rates, providing isolated and differentiated service levels.

Other approaches follow a non-linear feedback control to enforce proportional sharing and isolation in the presence of bursty and volatile workloads [59, 117]. For instance, Façade collects the average latency of requests accessing the storage utility of a shared storage system and dynamically adjusts the depth of the device queue. AVATAR follows a similar approach but enforces 95th-percentile latency. Differently, adaptive feedback controllers backed by self-tuning estimators, such as the one proposed by Triage, provide predictive and differentiated storage performance under varying workloads. Triage periodically observes latency perceived by Lustre-deployed applications and throttles per-client request queues to provide sustained throughput and latency.

3.2.2 Performance Modeling. Other strategies often used by SDS systems to efficiently control the storage environment are *heuristics*, which control and adjust selected enforcement points to meet a specific storage objective [88, 101, 108], and *performance models*, which characterize the behavior of the system and its workloads [42, 43, 54, 66, 109, 119, 120].

Heuristics. SDS controllers resort to heuristic-based mechanisms to estimate throughput or latency performance of selected points of the I/O stack. For instance, JoiNS continuously monitors latency and bandwidth utilization at network and storage stages and provides a simple heuristic that estimates network latency of networked storage systems. From this estimation, the controller adjusts priority queues installed at programmable switches to meet average and tail latency requirements. Similarly, SuperCell observes read and write latencies of Ceph storage nodes and implements a bandwidth-centered heuristic that calculates per-workload maximum bandwidth to provide adaptive configuration under read- and write-intensive workloads. Mantle, on the other hand, supports user-defined heuristics to provide programmable metadata management and load balancing over Ceph deployments.

Linear Programming. Linear programming (LP) mechanisms are also frequently used to support SDS control actions. For instance, flexStore resorts to an integer linear program to enforce adaptive replica consistency under varied energy constraints, and network and disk bandwidth, while Triage continuously collects latency measurements to serve a Recursive Least-Squares estimator that supports the feedback controller actions. Differently, general-purpose solvers used by Mirador estimate the performance of network-attached storage systems according to user-defined objectives. These solvers leverage from the continuous observations of network and storage resources, as well as periodic workload profiles, to optimize network traffic and data placement.

Latency analysis models are also used to enforce tail latency objectives under bursty scenarios. Leveraging from network calculus principles, PM and WC propose a model that estimates per-workload worst-case latencies. It models multiple system endpoints and induces time, flow, rate limit, and work conservation constraints to maximize the available time to serve a workload. Similarly, PSLO provides a forecast model that predicts per-VM high-percentile latency violations to simultaneously enforce X^{th} percentile latencies and throughput objectives.

Machine Learning. The use of maching learning (ML) to implement control strategies has just been recently adopted by SDS controllers. Specifically, SIREN uses an ML-based algorithm (i.e., classification and regression trees [17]) to assign proportional shares and reservations of compute and storage resources to HPC applications. While SIREN proposes resource enclaves for the efficient management of HPC infrastructures, the algorithm identifies opportunities for enclave migrations, due to workload and I/O demand variance. The introduction of such storage automation mechanisms allows more accurate enforcement strategies and fine-grained control over storage infrastructures.

3.3 Survey by Enforcement Strategy

The need to enforce varied storage policies throughout the I/O path leads SDS systems to assume different enforcement strategies. We now survey SDS systems regarding enforcement strategy employed at data plane stages, namely *token-buckets*, *scheduling*, *priority queues*, and *logic injection*. Table 3 highlights the enforcement strategies used by SDS data planes and depicts the current trends of the paradigm.[3]

3.3.1 Token-Bucket. A token-bucket is an abstract structure used by queues to control the rate and burstiness of I/O flows. A bucket is configured with a *bucket size*, which delimits the maximum token capacity, and a *bucket rate*, which defines the rate at which new tokens are added. When an I/O request arrives at the queue, it consumes tokens to proceed. If the bucket is empty, the request waits until sufficient tokens are in the bucket. Each bucket executes locally but is configured by SDS controllers according to existing storage policies and the current system state. Several SDS systems resort to token-bucket mechanisms for enforcing performance-oriented policies [60, 96, 99, 119, 120].

Per-queue token-buckets, such as the ones of IOFlow and sRoute, enforce max-min fair shares over I/O flows. As stages are deployed throughout the I/O path, queues are adjusted with different rates and sizes, providing differentiated I/O treatment and dynamic end-to-end control. Similarly, Retro proposes multipoint per-workflow token-buckets, employed over thread pools and RPC queues to achieve performance guarantees and resource fairness objectives in Hadoop stacks.

Per-workload token-buckets enforce tail latency objectives under bursty environments [119, 120]. To better bound the workload burstiness, PM implements multiple token-buckets per workload at each data plane stage, which in turn are continuously controlled and modeled by a proactive

[3]Systems that enforce scheduling mechanisms but do not detail employed policies are marked with ●.

feedback controller. On the other hand, WC optimizes the choice of bucket parameters through a *rate-bucket size curve* that characterizes workload burstiness while consolidating workloads into a storage server, in order to both meet tail latency objectives and minimize overall resource usage.

3.3.2 Scheduling. Scheduling has been a long-term strategy of storage systems to govern how I/O requests are served. In SDS-enabled architectures, scheduling is generally made over data plane queues to employ proportional sharing algorithms, prioritize and isolate requests, and enforce performance objectives over storage and network resources. For instance, single queue scheduling systems, such as Façade and AVATAR, manage per-workload requests to meet average and tail latency objectives. Requests are dispatched to a queue and served to a storage utility following an *Earliest Deadline First (EDF)* policy. As latency objectives are enforced at per-workload granularity, the deadline of a workload is the deadline of its older pending request [59].

Other solutions implement multi-point resource scheduling mechanisms to achieve fairness and sustained latency performance. Retro, for example, orchestrates per-workflow requests, employing a *Dominant Resource Fairness (DRF)* policy [25] to ensure resource fairness, and a *Bottleneck Resource Fairness (BRF)* policy [60] to throttle aggressive workflows and ensure proportional use of resources. Similarly, Pisces employs a per-node scheduler that implements *DRF* and *Deficit (Weighted) Round Robin (DRR)* policies [90, 92] to achieve system-wide fairness in multi-tenant cloud environments. Under a *DRF* policy, per-node schedulers track the resource usage of each tenant and recompute its resource allocation to continuously ensure max-min fair shares, while in *DWRR*, Pisces ensures per-tenant weighted fair shares of throughput. Libra, on the other hand, follows a similar approach but provides throughput reservations over disk resources through a *Distributed Deficit Round Robin (DDRR)* scheduling policy [55]. Further, Wisp rate limits microservice workflows through *BRF* and *DRF* policies to achieve throughput objectives and simultaneously prioritizes individual requests through *EDF* and *Least Slack Time First (LSTF)* [98] to enforce latency-related objectives.

3.3.3 Priority Queues. A number of SDS systems ensure prioritization and performance control through priority queues [60, 99, 108, 119, 120]. Controllers define and adjust the priority of queue-based data planes to provide different levels of latency among workloads according to installed storage policies. For instance, IOFlow, PM, and WC define the priority of token-bucket-enabled queues. Specifically, token-buckets serve first the highest-priority queues until no token is left, serve next lower-priority queues upon the replenishment of the bucket. Moreover, PM and WC specify per-workload priority queues over both storage and network resources. Retro, on its turn, enforces per-workflow priority queues over multi-point data plane stages, while JoiNS provides per-workload priority queues over programmable network switches.

3.3.4 Logic Injection. Storlet-based stages implement programmable enforcement structures to allow system designers to inject custom control logic over I/O flows [26, 87, 88, 101]. For instance, Mantle and Malacology leverage from existing storage subsystems of Ceph, such as durability, load balancing, and resource sharing, and inject control logic to enforce performance and data management storage policies. Mantle decouples policies from storage services by letting administrators inject metadata migration code to dynamically adjust metadata distribution of Ceph deployments. On the other hand, Malacology encapsulates existing system functionalities into reusable building blocks and injects Lua scripts to enable general-purpose systems to be programmed and adapted into tailored storage applications via composition. Further, SuperCell continuously monitors read and write requests of Ceph storage nodes and provides different storage reconfigurations to adapt storage settings under volatile workloads.

Differently, Crystal provides a programmable framework that allows the injection of programming logic to perform custom computations over object requests. This design allows administrators to implement a wide array of storage services to cope with different performance and data management policies, such as compression, cache optimization, and bandwidth differentiation.

3.4 Discussion

The SDS paradigm has drawn major focus on providing controlled I/O performance and fairness over cloud and application-specific infrastructures. While generally providing centralized and flat distributions, the next step toward scalable environments is to foster the development of hierarchical and decentralized control planes in such infrastructures. SDS-enabled HPC infrastructures, which are still at an early research stage, are composed of hierarchical control designs due to the increasing scale and performance requirements of supercomputers.

The centralized control distribution assumed by several SDS systems presents obvious limitations in scale and resilience. However, systems experience such limitations at different magnitudes, as they may provide a different number of stages and enforce storage policies with different degrees of complexity. Specifically, Façade and AVATAR enforce performance-oriented objectives over a single enforcement point, while Pisces, Libra, PSLO, and Cake need to orchestrate multiple points of control. Further, Wiera, SuperCell, and Mantle enforce data-oriented policies by injecting control logic at stages, which is less demanding than continuously adjusting stages for performance policies.

Differently, other systems follow a flat control distribution, with a prevalence on performance enforcement. While providing a more dependable design, the control centralization leads to clear scalability limitations. Interestingly, as systems enforce different storage policies over infrastructures, they implement different control strategies to adapt data stages to specified objectives. For instance, some systems like IOFlow resort to feedback control to continuously adjust the storage environment, while others, such as flexStore and Mirador, employ performance modeling strategies. As both folds provide a single control strategy, they are unable to provide a fully adaptable SDS environment. On the other hand, PM, WC, Triage, and PSLO combine feedback control and performance modeling strategies, thus providing a more adaptable storage environment, capable of enforcing complex storage policies under volatile environments.

Regarding hierarchical controllers, while providing a scalable design, some solutions present dependability limitations. For instance, the failure of a controller in Clarisse and SIREN systems leads to unsupervised control points in the infrastructure. Contrarily, sRoute and JoiNS issue control delegates to enforce policies in specific points of the I/O path, which, in case of failure, can be replaced online by another delegate.

On the data plane side, stack-based approaches focus on data management services, as data flows follow a pass-through layout and do not employ enforcement strategies. Existing stack-based systems may provide dedicated stacks designed for a specific objective such as flexStore and Moirai, or multiple stacking layers as done in SafeFS and Strata to provide a variety of storage services.

Queue-based data planes, on the other hand, are mainly designed to meet performance objectives. Nonetheless, even though operating over similar storage structures, existing approaches may differ from each other in several aspects. For instance, single queue systems, such as Façade and AVATAR, enforce average performance policies and are unable to meet high latency percentiles. Differently, Pisces and Libra provide system-wide performance isolation and fairness over multi-tenant cloud environments by enforcing per-tenant max-min fair shares. Other systems, such as PM and WC, provide multi-resource scheduling to enable complex storage policies to be enforced over network and storage resources. Further, IOFlow and PSLO provide multiple

enforcement queues, each serving at different rates in a per-VM basis, providing distributed and dynamic policy enforcement.

Finally, storlet-based systems introduce novel storage abstractions and programmability to existing storage systems. For instance, Crystal, Tiera, and SuperCell repurpose existing storage subsystems and configurations to enforce data and routing activities. Malacology, Mantle, and PADS, on the other hand, abstract underlying storage systems to ease the development of custom storage systems. As opposed to stack-based designs, which are transparent as stacks seat between two I/O layers, storlet-based stages introduce increased complexity to the design of storage solutions.

4 LESSONS LEARNED AND FUTURE DIRECTIONS

We now discuss the key insights provided by this survey, grouped by storage infrastructure (***SIx***); planes of functionality, namely data (***Dx***) and control planes (***Cx***); SDS interfaces (***Ix***); and other aspects of the field (***Ox***). We focus on the design space and characteristics of current SDS systems and on possible future research directions of the paradigm.

SI1: SDS research is widely explored over cloud and application-specific infrastructures. SDS research has drawn major focus on cloud and application-specific designs. However, the former are generally composed by centralized and flat controllers and queue-based stages, while the latter focus on storlet-based designs. With the continuous increase of data centers' complexity, as well as the emergence of serverless cloud computing [40], further research on hierarchical and decentralized control plane distributions will be needed to employ over such infrastructures.

SI2: HPC-based SDS systems are at an early research stage. The increasing requirements of scale and performance of supercomputers have led to the first advances toward SDS-enabled HPC systems being composed by hierarchical control distributions backed by high-performance queues. Considering that few proposals address this challenge, novel contributions are expected to foster research in the SDS-HPC field and to attend to the requirements of incoming Exascale infrastructures [1, 22], as well as to approximate HPC and cloud ecosystems [67].

SI3: SDS systems for emerging computing paradigms are unexplored. SDS-enabled systems have been employed over modern storage infrastructures to achieve different objectives. However, with the emergence of novel computing paradigms such as serverless cloud computing [40], IoT [5], and Exascale computing [22], a number of challenges (e.g., scalability, performance, resiliency) need to be addressed to ensure sustained storage efficiency. As such, the research and development of novel decentralized SDS architectures (e.g., wide-area SDS systems, gossip-based control protocols), as well as the convergence of different software-defined technologies (e.g., storage, networking, security), will be essential to provide a fully programmable storage environment and attend to the requirements of emerging computing paradigms.

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 led 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: Performance management services dominate SDS systems. Performance-oriented services have dominated the spectrum of storage policies and services supported by SDS systems. This

design has led to a large research gap for the remaining policy scopes. Nevertheless, the advent of modern storage technologies, such as kernel-bypass [116], storage disaggregation [46, 89], and new storage hardware [41, 45], along with the emergence of novel computing paradigms, requires 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 design. 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 [40] and IoT infrastructures [1, 5], as well as on the approximation of HPC and cloud ecosystems [67].

D5: Repurposing of existing storage subsystems is overlooked. Existing 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 [75, 87], there is no SDS system that leverages from existing storage subsystems (e.g., QoS provisioning, I/O scheduling) to be repurposed as programmable storage objects and reused in different storage contexts throughout the I/O path (e.g., key-value stores, distributed file systems). Such a design would open research opportunities toward programmable storage stacks and foster reutilization of complementary works [44, 76, 112] and existing storage subsystems [23].

D6: Heterogeneous data planes are unexplored. Despite the number of possible configurations and design flavors of SDS data planes, the combination of different stage designs has not yet been explored. This turns the data plane domain mostly monolithic, tailored for specific storage objectives and suboptimal enforcement efficiency. As such, following the steps of the SDN paradigm [50], novel contributions toward heterogeneous data plane environments that explore the different tradeoffs of combining stack-, queue-, and storlet-based designs should be pursued.

C1: Current systems are unsuitable for larger environments. A large quota of SDS controllers follow a flat distribution to serve small to medium-sized storage infrastructures [99]. However, the emergence of novel computing paradigms made of complex and highly heterogeneous storage stacks (e.g., serverless computing, IoT) make current control centralization assumptions unsuitable. As such, leveraging from the initial efforts of decentralized controllers [98], it is essential to further investigate this topic and provide novel contributions toward control decentralization.

C2: Controllers lack programmability. Current hierarchical controllers resort to delegate functions or micro-services to improve control scalability, providing limited control functionality to control peers. Instead, it would be interesting to follow similar design principles as SDS data planes and make 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. SDS systems use a *"logically centralized"* controller to orchestrate the storage environment [99]. However, behind this simple but ambiguous assumption lies a great deal of practical complexity of dependability, leaving no clear definitions on its practical challenges and actual impact in performance and scalability at the overall storage infrastructure. Similarly to other software-defined approaches [52], these assumptions leave several open questions regarding controllers' dependability that require further investigation such as fault tolerance and consistency [11, 15, 16, 48], load balancing and control dissemination [21], controller synchronization [81], and concurrency [24].

C4: Controllers are self-adaptable. Several controllers resort to feedback control mechanisms to dynamically adjust SDS settings, while few proposals rely on performance modeling 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. As such, it is essential to advance the research of autonomous mechanisms for supporting control decisions of SDS controllers, by combining and providing novel control strategies. For instance, exploring distributed ML techniques [53] would be of great utility to attend to the needs of both modern and emerging infrastructures, not only for the obvious reasons of scale but also for ensuring new levels of accuracy in heterogeneous and volatile environments.

I1: Communication protocols are tightly coupled to planes of functionality. SDS interfaces are used as simple communication APIs, making communication protocols tightly coupled to either the control or data plane implementation. This design prevents the reutilization of alternative technologies and inhibits 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 research opportunities to attain the communication challenges of novel storage paradigms [5, 40] and network fabrics [1].

I2: Interfaces lack standardization and interoperability. Contrarily to SDN [50], 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 toward standard and interoperable SDS interfaces should be expected.

O1: Black-box and end-to-end monitoring are unexplored. Current monitoring mechanisms of SDS controllers are predefined and static. Integrating black-box [68] and end-to-end monitoring systems [61, 111] in these would bring novel insights to the field and would allow assisting controllers to define accurate enforcement strategies over I/O flows. Further, black-box-based approaches do not require *a priori* knowledge of the I/O stack and comprehend near-zero changes to the original codebase.

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

5 CONCLUSION

In recent years, the SDS 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, SDS has proven to be a fundamental solution to enable end-to-end policy enforcement and holistic storage control and ensure performance isolation, QoS provisioning, and resource fairness.

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, depicting the design principles, internal

organization, and storage properties. As a first contribution, we present a definition of SDS systems and outline the distinctive characteristics of an SDS-enabled infrastructure. We define SDS as a storage architecture that decouples storage mechanisms and policies into a control plane and a data plane that enforces storage policies over I/O flows.

As a second contribution, we define the SDS design principles and categorize planes of functionality regarding internal organization and distribution. We surveyed existing work and distilled a number of designs for data plane stages, being categorized as stack, queue, or storlet based. At the control plane, we categorize distributed SDS controllers as being flat or hierarchically organized.

Further, we propose a taxonomy and classification of existing SDS systems to organize the manifold approaches according to their storage infrastructure (i.e., cloud, HPC, and application specific), control strategy (i.e., feedback control and performance modeling), and enforcement strategy (i.e., token-bucket, scheduling, priority queues, and logic injection). Cloud-based solutions address the requirements of general-purpose 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 dedicated 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 research directions for the 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 and alternative data plane combinations, and novel storage policies and services of less explored scopes. Noticeably, since HPC-oriented systems are at an early research stage, novel contributions to improve performance variability and interference can be expected. Likewise, novel SDS solutions capable of meeting the requirements of scale and performance of incoming infrastructures, namely serverless, IoT, and Exascale computing, can also be expected. To conclude, due to its 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.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions that helped us improve this article.

REFERENCES

[1] George Amvrosiadis, Ali R. Butt, Vasily Tarasov, Erez Zadok, Ming Zhao, Irfan Ahmad, Remzi H. Arpaci-Dusseau, Feng Chen, Yiran Chen, Yong Chen, Yue Cheng, Vijay Chidambaram, Dilma Da Silva, Angela Demke-Brown, Peter Desnoyers, Jason Flinn, Xubin He, Song Jiang, Geoff Kuenning, Min Li, Carlos Maltzahn, Ethan L. Miller, Kathryn Mohror, Raju Rangaswami, Narasimha Reddy, David Rosenthal, Ali Saman Tosun, Nisha Talagala, Peter Varman, Sudharshan Vazhkudai, Avani Waldani, Xiaodong Zhang, Yiying Zhang, and Mai Zheng. 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*. USENIX, 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*. 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*. 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] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33.

[7] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. 2018. Distributed SDN control: Survey, taxonomy, and challenges. *IEEE Communications Surveys & Tutorials* 20, 1 (2018), 333–354.

[8] 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*. USENIX, 59–73.

[9] 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*. USENIX, 49–65.

[10] 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.

[11] 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*. ACM, 1–6.

[12] Dimitri P. Bertsekas and Robert G. Gallager. 1992. *Data Networks*. Prentice Hall.

[13] Jean-Pascal Billaud and Ajay Gulati. 2013. hClock: Hierarchical QoS for packet scheduling in a hypervisor. In *8th ACM European Conference on Computer Systems*. ACM, 309–322.

[14] Francieli Z. Boito, Eduardo C. Inacio, Jean L. 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] 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*. IEEE, 73–78.

[16] 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*. IEEE, 169–180.

[17] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. 1984. *Classification and Regression Trees*. CRC Press.

[18] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffrey 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*. USENIX, 17–31.

[19] 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*. ACM.

[20] 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.

[21] 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*. ACM, 7–12.

[22] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E. Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. 2011. The international exascale software project roadmap. *International Journal of High Performance Computing Applications* 25, 1 (2011), 3–60.

[23] 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*. IEEE, 155–164.

[24] 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*. ACM, 402–415.

[25] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.

[26] 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*. USENIX, 243–256.

[27] 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*. ACM, 13–24.

[28] 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*. USENIX, 437–450.

[29] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter Varman. 2012. Demand based hierarchical QoS using storage resource pools. In *2012 USENIX Annual Technical Conference*. USENIX, 1–13.

[30] Red Hat. 2020. What is software-defined storage? Retrieved Feb. 27, 2020, from https://www.redhat.com/en/topics/data-storage/software-defined-storage.

[31] Joseph L. Hellerstein, Yixin Diao, Sujay S. Parekh, and Dawn M. Tilbury. 2004. *Feedback Control of Computing Systems*. John Wiley & Sons.

[32] 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*. IEEE, 127–136.

[33] 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.

[34] Patrick Hunt, Mahadev Konar, Flavio Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems. In *2010 USENIX Annual Technical Conference*. USENIX.

[35] 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*. IEEE, 346–355.

[36] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Holzle, 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*. ACM, 3–14.

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

[38] 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.

[39] 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*. USENIX, 265–279.

[40] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley.

[41] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. SplitFS: A file system that minimizes software overhead in file systems for persistent memory. In *27th ACM Symposium on Operating Systems Principles*. ACM, 494–508.

[42] 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*. IEEE, 95–104.

[43] Magnus Karlsson, Christos T. Karamanolis, and Xiaoyun Zhu. 2005. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage* 1, 4 (2005), 457–480.

[44] 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*. ACM, 374–385.

[45] 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*. USENIX.

[46] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *11th European Conference on Computer Systems*. ACM, 29:1–29:15.

[47] 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*. USENIX, 427–444.

[48] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: A distributed control platform for large-scale production networks. In *9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 25:1–25:14.

[49] 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*. ACM, 25:1–25:17.

[50] 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.

[51] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[52] 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*. ACM, 1–6.

[53] Mu Li, David G. Andersen, Jun W. 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*. USENIX, 583–598.

[54] 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*. ACM, 28:1–28:14.

[55] Tong Li, Dan P. Baumberger, and Scott Hahn. 2009. Efficient and scalable multiprocessor fair scheduling using distributed weighted Round-Robin. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 65–74.

[56] libfuse. 2001. libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. Retrieved Feb. 27, 2020, from https://github.com/libfuse/libfuse.

[57] Ke Liu, Xuechen Zhang, Kei Davis, and Song Jiang. 2013. Synergistic coupling of SSD and hard disk for QoS-aware virtual memory. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 24–33.

[58] 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*. IEEE, 1–12.

[59] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. 2003. Façade: Virtual storage devices with performance guarantees. In *2nd USENIX Conference on File and Storage Technologies*. USENIX.

[60] 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*. USENIX, 589–603.

[61] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems* 35, 4 (2018), 11:1–11:28.

[62] Ricardo Macedo, Alberto Faria, João Paulo, and José Pereira. 2019. A case for dynamically programmable storage background tasks. In *2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW'19)*. IEEE.

[63] 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.

[64] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. 2011. Differentiated storage services. In *23rd ACM Symposium on Operating Systems Principles*. ACM, 57–70.

[65] Microsoft. 2020. Microsoft Windows Server. Retrieved Feb. 27, 2020, from https://docs.microsoft.com/en-us/windows-server/storage/storage.

[66] 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*. IEEE, 81–90.

[67] 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.

[68] 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*. IEEE, 534–541.

[69] 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*. ACM, 165–176.

[70] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*. USENIX, 305–319.

[71] OpenStack. 2018. OpenStack Documentation: Storlets. Retrieved Feb. 27, 2020, from https://docs.openstack.org/storlets/latest/.

[72] 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*. ACM, 471–484.

[73] João Paulo and José Pereira. 2014. A survey and classification of storage deduplication systems. *ACM Computing Surveys* 47, 1 (2014), 11:1–11:30.

[74] 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*. USENIX, 1–16.

[75] 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*. ACM, 9:1–9:12.

[76] 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*. ACM, 6:1–6:12.

[77] Ajaykrishna Raghavan, Abhishek Chandra, and Jon B. Weissman. 2014. Tiera: Towards flexible multi-tiered cloud storage instances. In *15th International Middleware Conference*. ACM, 1–12.

[78] 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).

[79] 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.

[80] 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*. ACM, 20:1–20:10.

[81] 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.

[82] 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*. 231–244.

[83] 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*. USENIX, 1–16.

[84] Philip Schwan. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, Vol. 2003. 380–386.

[85] 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*. USENIX, 67–80.

[86] 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*. IEEE/ACM, 203–212.

[87] 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*. ACM, 175–190.

[88] 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*. IEEE, 1–12.

[89] 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*. USENIX, 69–87.

[90] M. Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit Round Robin. In *ACM SIGCOMM 1995 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. ACM, 231–242.

[91] David Shue and Michael J. Freedman. 2014. From application requests to virtual IOPs: Provisioned key-value storage with Libra. In *9th European Conference on Computer Systems*. ACM, 17:1–17:14.
[92] 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*. USENIX, 349–362.
[93] 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*. IEEE, 1–10.
[94] 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.
[95] 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*. ACM, 17:1–17:11.
[96] 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*. USENIX, 197–212.
[97] 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*. ACM, 174–181.
[98] 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*. ACM, 611–623.
[99] 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*. ACM, 182–196.
[100] Top500. 2020. Top 500: Supercomputers. Retrieved Feb. 27, 2020, from https://www.top500.org/.
[101] Keitaro Uehara, Yu Xiang, Yih-Farn R. 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*. IEEE, 103–112.
[102] 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*. ACM, 13:16–13:16.
[103] VMware. 2020. VMware Cloud. Retrieved Feb. 27, 2020, from https://cloud.vmware.com/.
[104] Carl A. Waldspurger and William E. Weihl. 1994. Lottery scheduling: Flexible proportional-share resource management. In *1st USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 1–11.
[105] 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*. ACM, 14:1–14:14.
[106] Yin Wang and Arif Merchant. 2007. Proportional-share scheduling for distributed storage systems. In *5th USENIX Conference on File and Storage Technologies*. USENIX, 47–60.
[107] 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*. USENIX, 307–320.
[108] 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*. IEEE, 194–200.
[109] Jake Wires and Andrew Warfield. 2017. Mirador: An active control plane for datacenter storage. In *15th USENIX Conference on File and Storage Technologies*. USENIX, 213–228.
[110] 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*. IEEE, 1–12.
[111] 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*. USENIX, 379–394.
[112] Suli Yang, Tyler Harter, Nishant Agrawal, Salini S. 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*. ACM, 474–489.
[113] 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*. ACM, 432–445.

[114] Soheil H. 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*. ACM, 19–24.

[115] 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*. IEEE, 750–759.

[116] 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*. ACM.

[117] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. 2005. Storage performance virtualization via throughput and latency control. In *13th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 135–142.

[118] 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*. ACM, 18:1–18:12.

[119] 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*. ACM, 598–610.

[120] 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*. ACM, 29:1–29:14.

Received May 2019; revised January 2020; accepted February 2020