HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-644182)

Software-Defined Storage for Big Data on top of the OpenStack platform

D2.4 Reference implementation of architectural building blocks

Due date of deliverable: 31-10-2017
Actual submission date: 09-11-2017

Start date of project: 01-01-2015 Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.0
Number of pages	103
WP/Task related to this document	WP2 / T2.3
WP/Task responsible	URV
Leader	Raúl Gracia-Tinedo (URV)
Technical Manager	Gerard París (URV)
Quality Manager	Francesco Pace (EUR)
Author(s)	Raúl Gracia-Tinedo (URV), Yosef Moatti (IBM), Gerard París (URV), Josep Sampé (URV), Ramon Nou (BSC), Marc Siquier (BSC), Daniele Venzano (EUR), Francesco Pace (EUR), Pedro García-López (URV), Marc Sánchez-Artigas (URV), Filip Gluszak (GDP), Nejc Bat (ARC), Saxa Egea (IDI).
Partner(s) Contributing	URV, IBM, BSC, EUR, IDI, ARC, GDP
Document ID	IOStack_D2.4_Public.pdf
Abstract	This document presents the release version of the IOStack toolkit by providing a global view of its building blocks and their integration. We also present the achievements of the IOStack toolkit in the form of Key Performance Indicators (KPIs) and how these KPIs impact and are exploited by use-case companies. In addition, this deliverable provides an in-depth description and experimental evaluation of Crystal: the SDS building block for object storage. Finally, we also present experimental results on how the IOStack architecture is a substrate to develop coordination strategies between compute/storage components that optimize complex analytics workloads.
Keywords	Prototypes, specifications, evaluations.

History of changes

Version	Date	Author	Summary of changes
v0.1	01-09-2017	Raúl Gracia	First version of document
v0.2	21-09-2017	Francesco Pace	Fill Section 4.1.2 - KPI3: Analytics scheduling.
v0.3	16-09-2017	Ramon Nou	Progress on block/object storage filters.
v0.4	17-10-2017	Raúl Gracia	New version of Part III (Crystal) and added a piece of research (lambda migration) for Part IV.
v0.5	20-10-2017	Raúl Gracia	Initial version for Part IV of the document.
v0.6	25-10-2017	Gerard París	Updated Part II of the document and Appendix A.
v0.7	31-10-2017	Gerard París	Updated Appendix D.
v0.8	31-10-2017	Raúl Gracia	Improvements from Yosef Moatti review.
v1.0	09-11-2017	Gerard París	Improvements from Francesco Pace review.

Table of Contents

1 Executive summary

This document provides an in-depth overview of the main outcome of the project: the *IOStack toolkit*, a Software-Defined Storage (SDS) Stack for Big Data analytics.

In the first part of the present deliverable, we synthesize the main achievements of the project as Key Performance Indicators (KPIs) that our toolkit achieved. Furthermore, we briefly describe the technical contributions for each KPI and its relation and benefits to the project's use cases. This will provide an instructive and complete view of the goals, contributions and real benefits of this project.

In second place, we describe the final version of the IOStack toolkit (extending deliverable D2.3 in M24) and its software building blocks: *Zoe* (analytics virtualization), *Konnector* (block storage) and *Crystal* (object storage), as well as other components (dashboard, Stocator, Storlets, and FUSE client). We illustrate that the toolkit is integrated and ready-to-use, with several deployments running. We also show that most of the software already implements code quality best-practices (testing, continuous integration) and has practical documentation for users and developers to use it.

Moreover, in this deliverable[1], we focus on Crystal: The first SDS architecture for object storage (OpenStack Swift). We describe how Crystal implements the *filter* abstraction to accommodate new storage optimizations —including compression, caching or computations on data streams— that can be easily orchestrated via high-level policies. Moreover, we also demonstrate that Crystal exploits the control plane to dynamically react to changing workload conditions. We provide an extensive evaluation of Crystal in a 14-machine cluster under well-known benchmarks and trace replays and workloads of our use case companies (Idiada, Arctur, GridPocket). As a success story, we performed a pilot deployment of Crystal in Idiada in order to exploit it in production storage workflows.

Finally, we provide real experiments on the cooperation of compute and storage building blocks in IOStack to leverage cross-layer optimizations. In particular, we show how the IOStack control plane can orchestrate both the compute building block (Zoe) and the object store (Crystal) to better accommodate workloads that encompass both clusters. This leaves open an interesting branch of research and development to further continue the exploitation of the project's outcomes.

[1] The other components in IOStack will be described in their respective deliverables.

2 Introduction and Motivation

As the operation of companies and organizations increasingly involves more digital processes, their daily activity inherently generates larger amounts of *data* that should be stored along time. Such data has been recently acknowledged as *Big Data* given its volume, velocity and variety properties that, among other properties (5Vs), make their usage complex and resource consuming [1, 2]. There are a myriad of companies in diverse sectors —e.g., Internet of Things (IoT), Online Social Networks (OSNs), research institutions, automotive companies— that face the challenges of Big Data; these companies require scalable and practical means not only of storing such data, but also of extracting value from it.

To exploit these large amounts of data, Big Data *analytics* frameworks have rapidly become a key enabler technology increasingly involved within the business processes of many companies and organizations [3, 4, 5, 6, 7]. The reason for this phenomenon is simple: The parallel and scalable design of modern analytics frameworks represent a golden opportunity for diverse companies to effectively "extract value" from enormous amounts of data they produce. Nowadays, Big Data analytics frameworks provide rich suites of processing models, including MapReduce jobs, graph Data Bases (DBs) and parallel SQL engines, among others, which enable data scientists to explore and analyze large datasets [8, 9, 10].

The value extracted from large datasets may adopt different forms; for instance, in the case of analyzing logs of an e-commerce site, value may be a set of critical insights on how users behave within the site; or perhaps, if we consider a smart grid energy company like GridPocket, value may be instantiated as a deep understanding on the energy consumption of cities or even entire countries. Overall, this kind of information enables companies to take better and faster decisions, which make them more competitive than before.

2.1 Problems of Today's Analytics Platforms

Unfortunately, despite its potential, leveraging Big Data analytics at scale involves managing a complex ecosystem composed of storage systems and analytics frameworks, which has associated important administration and performance challenges, among others. In this project, we target the following ones:

Lack of advanced storage management for analytics. The first problem of Big Data analytics is at the storage layer, that is, where data lives. Most storage systems used in Big Data analytics (K/V stores, object storage, block storage) are scalable and provide high availability [11, 4, 12, 13], but lack from simplified and fine-grained management models. For instance, automation tools for storage provisioning and tiering (containers, block volumes) are still very early in many cases, which involves important efforts from an administrator viewpoint. Even worse, the lack of advanced management models prevents administrators from easily adapting the storage system to specific applications, in/out data flows, as well as providing specific Quality of Service (QoS) levels to tenants sharing a storage cluster.

Need for extensible storage optimizations: In a shared Big Data platform, the storage system may be subject to concurrent and heterogeneous workloads. For instance, we can imagine a storage cluster continuously storing data from IoT measurement devices or server logs. At the same time, one or many data analytics frameworks may be extracting data from the storage system for executing SQL queries. In this scenario, we believe that the storage system should be open to be extended with new optimization mechanisms to cope with such varied workloads. To illustrate this, if we retake the previous example and consider that IoT devices store compressible data, we could deploy a data compression or reduction technique on this particular data flow to save storage space. Similarly, in the case that an analytics application fetches a dataset to execute a SQL query, we could extend the storage system with a mechanism to discard useless data before serving it, which may significantly improve transfer performance.

Simple and policy-driven deployment of analytics is a must. At the compute level, administrators deploying analytics frameworks waste important resources and time to perform the necessary

configuration and setup tasks. If we consider a datacenter aiming at providing Big Data analytics in the cloud, it is clear that an advanced tool is needed for rapidly configuring and deploying a wide variety of analytics. Moreover, such a tool should be able of orchestrating and scheduling running analytics within a cluster in order to provide predictable completion deadlines.

In technical terms, these problems may negatively impact on the storage lifecycle of Big Data, as well as the further data processing of analytics frameworks. Clearly, this may reduce the competitiveness of European companies aiming at benefiting from Big Data analytics, as they can face significant administration and optimization obstacles.

2.2 Goals of IOStack

Solving these (and other) problems is the main objective of the IOStack project. To this end, the main outcome of this project is the *IOStack toolkit*[2], which, among other assets, achieves the following key contributions:

- The IOStack toolkit materializes Software-Defined Storage (SDS) models that can overcome the lack of advanced administration and management capabilities of storage systems for analytics. This includes *policy-based provisioning and automation* features to greatly facilitate the task of system administrators to manage storage for analytics services. Moreover, real-time and off-line *monitoring tools* are available for administrators to take decisions based on workload characteristics.

- To meet the storage needs of modern analytics, our toolkit targets to open the storage system used in analytics workflows with non-anticipated functionalities and optimizations for improving the performance and efficiency of the whole Big Data lifecycle. We built a framework for object and block storage —OpenStack Swift and Cinder, respectively— that is capable of injecting new code, in form of an abstraction named *filter*, that enriches the functionalities of the system, even on-the-fly. System developers can also equip filters with *control algorithms* that dynamically change the behavior of the storage based on real-time monitoring metrics.

- At the compute layer, the IOStack toolkit leverages *automated and simplified deployment of analytics* frameworks. Administrators write simple deployment descriptors to spawn entire clusters of analytics in seconds via lightweight container-based virtualization (Docker). Furthermore, running analytics instances can be then *intelligently scheduled* via policies that help administrators at predicting the execution of analytics in shared clusters.

As we describe in this document, the IOStack toolkit is a ready-to-use prototype, with several deployments currently running. We also show in this document (and in the other M36 deliverables) how the toolkit already solves problems in multiple aspects of the Big Data lifecycle of our use-case companies (Arctur, Idiada and GridPocket).

2.3 Work done in the last year

During the last year, significant efforts were devoted to identify and evaluate a battery of Key Performance Indicators in order to provide a quantitative measure on the project achievements. These KPIs are summarized in Sec. 3 and demonstrated in Sec. 4 where their impact on IOStack use-cases is also described.

In sections 6 to 12 inclusive we describe each IOStack building block extending the previous deliverable D2.3 “Public release of the IOStack Toolkit”. In the last year, a new component (FUSE Client) has been added to the IOStack Toolkit software stack, together with a set of filters specifically prepared to demonstrate one of the use cases. In addition, new filters have been developed for object storage, like JIT Prefetching, an storage filter that is able to preload objects just before they are used in order to reduce the amount of memory used for caching (see App. C). IOStack Dashboard has been

[2] Available at `https://github.com/iostackproject`.

also extended with new features and refactored as an OpenStack Horizon plugin in order to simplify its installation in existing OpenStack environments.

During the last year, Eurecom has published three releases of Zoe Analytics with several important changes. It was added support for Google Kubernetes [14] back-end, which is one of the major players in the container orchestration space. The web interface was also changed to improve the user experience. It now includes the ZApp shop, a web-based tool where users are free to compose and configure their own applications in a more intuitive way. Moreover, work has also been done on an integration of Zoe with Crystal that helps the software-defined-storage layer to select the appropriate policy to apply on object data streams depending on the needs of the applications.

Likewise, Crystal has been extended with hyper-controllers to orchestrate the storage service according to the needs of analytics applications (see Sec. 14), and also with the ability to apply lambda functions on object data streams (see Sec. 12.8). Based on this Crystal extension, in Sec. 15 we describe how Crystal can transparently migrate general-purpose dataflow computations within analytics application as lambda functions on object data streams.

Part I
IOStack: Achievements and Impact

3 Key Performance Indicators

To provide a quantitative measure of the achievements of IOStack, in Table 1 we summarize a battery of Key Performance Indicators (KPIs) and their relation to the project use-cases. According to the spirit of SDS systems, IOStack toolkit KPIs in Table 1 can be classified either as *management* (KPIs 1, 2, 4, 8, 9 and 12) or *performance* (KPIs 3, 5, 6, 7, 10 and 11) .

Table 1: IOStack Key Performance Indicators.

KPI	Technique	Metric	Improvement	Storage type	Use-case
1	SDS object storage	Management cost	Storage management [15, 16]	Object	Arctur
2	Simplified analytics virtualization	Management cost	Analytics deployment [17]	Block, Object	Arctur
3	Analytics scheduling	Turnaround time reduction	18x [17] (D5.3)	Block, Object	Arctur
4	Compute/storage orchestration	Management cost	Infrastructure optimization (Sec. 14)	Object	Arctur
5	File filters	Data transfer reduction	10x [18]	Object	GridPocket
6	Object connectors	Communication costs reduction	1.5x-18x [19, 20]	Object	GridPocket
7	Computation close to data	Application speedup	4x-37x [21], 1.5x-4x (Sec. 12)	Object	GridPocket
8	Automatic lambda migration	Management cost	Infrastructure optimization (Sec. 15)	Object	GridPocket
9	SDS block storage	Management cost	Storage management (D3.3)	Block	Idiada
10	Data reduction	Data compression	60% (D3.3)	Block	Idiada
11	Prefetching, caching	Data access throughput	$1-45\%$ (App. C)	Object	Idiada
12	File system control	Management cost	Storage management (Sec. 9)	Object	Idiada

On the one hand, IOStack is aimed at providing advanced automation of storage provisioning and analytics virtualization. Among other contributions, IOStack provides automated, policy-based provisioning of block and object storage resources (KPIs 9 and 1), as well as a simplified layer of analytics virtualization based on containers (KPI 2). Even more, the control plane of IOStack can perform coordinated orchestration of both storage and compute building blocks, thus expanding the optimization opportunities within the datacenter (KPIs 4 and 8). Moreover, IOStack offers centralized control of client file-systems, thus achieving end-to-end storage orchestration (KPI 12).

On the other hand, IOStack toolkit offers a platform to develop and deploy data services that optimize Big Data analytics, as well as other types of workloads. For instance, system developers may create data services that discard useless data during the ingestion phase of analytics (KPI 7), that op-

portunistically compress redundant data (KPI 10) or that exploit workload patterns to optimize data access (KPI 11). Such results demonstrate that IOStack improves the customization and flexibility of storage systems, which has a positive impact on client applications.

Also related to performance optimization, within the project we developed mechanisms that improve the compute-storage interactions. This involves the incorporation of advanced indexing techniques on unstructured data (KPI 5), the development of an optimized object storage connector for analytics (KPI 6), or the design of intelligent scheduling algorithms that make a more efficient use of compute resources (KPI 3). As a result, the IOStack toolkit reports both efficiency and performance gains for analytics.

Next, we demonstrate these KPIs and describe their impact on IOStack use-cases.

4 Impact of IOStack on Use-cases

In this section, we provide a detailed overview of each IOStack use case: GridPocket, Arctur and Idiada. We first describe the main activity of each company, putting special emphasis on the real-world needs that motivated their inclusion as IOStack use-cases. Second, from Table 1, we describe management KPIs and demonstrate performance KPIs with an experimental evaluation. This will help the to understand how the research contributions of IOStack solved particular use-case needs. Finally, we report for each use-case company the real life exploitation possibilities IOStack outcomes.

Please, note that in this section we only give a high-level description of the techniques employed and the results obtained, in order to provide a big picture of the project. For an in depth description of every technical contribution and its evaluation, we refer to the appropriate deliverables.

4.1 Arctur: Making Life Easier to Cloud Providers

4.1.1 Use-case Description

Established in 1992, Arctur has progressed to become the main Slovenian commercial supplier of HPC (High Performance Computing) services and solutions. Arctur has its own HPC infrastructure to be used as the technological foundation for advanced HPC and Cloud computing solutions and innovative web services in a distributed, high-redundancy environment. The company has extensive experience in server virtualization and deployment, integration of disparate IT-systems, IT support of project-management and server farm leverage for the deployment of Software as a Service (SaaS), specialised for small and media enterprises (SME).

Arctur is a small private Cloud Services provider. Their competitive advantage is in the fact that they are able to adapt and customize their services to individual customers. Therefore, a critical need for Arctur is to able of better adapting their services to the needs of their customers. For this reason, the company aims at having an automatic provisioning layer of services to different customers based on their needs. An example of the current problems that Arctur faces may be illustrative: A customer might request a more economic rental price of storage. Thus, data from such a customer should be automatically stored on a lower tier storage server. In this situation, instead of performing manual administration to achieve this goal, Arctur requires this change in the storage service of such client to be done automatically. That is, all the new incoming data needs to be automatically redirected to the new and more economic storage tier.

Additionally, similar automated provisioning may apply to other client requests apart from storage tiering; that is, client may request specific encryption, replication or caching services to apply to their data. Satisfying such requirements is currently very hard to achieve to Arctur, representing an important source of administration costs.

4.1.2 Demonstration of KPIs

KPI1: SDS for object storage. Datacenter providers like Arctur deliver virtually unlimited amounts of storage capacity to their clients. Specifically, object storage services, such as OpenStack Swift, are becoming increasingly popular for all kind of clients, and especially for those executing Big Data analytics in the cloud.

However, despite their growing popularity, object stores are not well prepared for the heterogeneity associated to multiple tenants. Typically, a deployment of an object store in the cloud uses a monolithic configuration setup, even when the same object store acts as a substrate for different types of applications with time-varying requirements [22, 23]. As a result, all applications experience the same service level, though the workloads from different applications can vary dramatically. For example, while a Web application would have to store mainly small-medium sized photos (KB- to MB-sized objects), a Big Data analytics engine would probably generate read and write requests for large files. Clearly, using a static configuration *inhibits optimization of the system* to such varying needs.

Beyond the particular needs of a type of workload, the requirements of applications can also vary greatly. For example, an archival application may require transparent compression, annotation, and encryption of the archived data. In contrast, a Big Data analytics application may benefit from the computational resources of the object store to eliminate data movement and enable in-place analytics capabilities [24, 21]. Supporting such a variety of requirements in an object store is challenging, because in current systems, custom functionality is hard-coded into the system implementation due to the *absence of a true programmable layer*, making it difficult to maintain as the system evolves. To Arctur, storage customization represents an attractive added-value, but involves important administration costs, especially at large scale.

In this project, we argue that Software-Defined Storage (SDS) is a compelling solution to these problems. As in SDN, the separation of the "data plane" from the "control plane" is the best-known principle in SDS [25, 26, 27, 15, 28]. Such separation of concerns is the cornerstone of supporting heterogeneous applications in data centers. However, the application of SDS fundamentals on cloud object stores is not trivial.

To overcome the rigidity of object stores we present Crystal: The first SDS architecture for object storage to efficiently support multi-tenancy and heterogeneous applications with evolving requirements. Crystal achieves this by separating policies from implementation and unifying an extensible data plane with a logically centralized controller. As a result, Crystal allows to dynamically adapt the system to the needs of specific applications, tenants, and workloads.

We highlight two aspects of Crystal, although it also has other assets. First, Crystal presents an extensible architecture that unifies individual models for each type of resource and transformation on data. For instance, global control on a resource such as IO bandwidth can be easily incorporated as a small piece of code. A dynamic management policy like this is materialized in the form of a distributed, supervised controller, which is the Crystal abstraction that enables the addition of new control algorithms. These controllers, which are deployable at runtime, can be fed with pluggable per-workflow or resource metrics. Examples of metrics are the number of IO operations per second and the bandwidth usage. An interesting property of Crystal is that it can even use object metadata to better drive the system towards the specified objectives.

Second, Crystal's data plane abstracts the complexity of individual models for resources and computations through the filter abstraction. A filter is a piece of programming logic that can be injected into the data plane to perform custom calculations on object requests. Crystal offers a filter framework that enables the deployment and execution of general computations on objects and groups of objects. For instance, it permits the pipelining of several actions on the same object(s), similar to stream processing frameworks [29]. Consequently, practitioners and systems developers only have to focus on the development of storage filters, as their deployment and execution is done transparently by the system. To our knowledge, no previous SDS system offers such a computational layer to act on resources and data.

To justify KPI 1 in Table 1, let us describe a real world example. Fig. 1 shows the execution of several storage automation policies on a workload related to containers `C1` and `C2` belonging to tenant `T1`. In OpenStack Swift, a container is a namespace for objects. Specifically, we executed a write-only synthetic workload (4PUT/second of 1MB objects) in which data objects stored at `C1` consist of random data, whereas `C2` stores highly redundant objects. We ran our experiments in a 13-machine cluster formed 4 Dell PowerEdge 430 nodes for Spark, 8 Dell PowerEdge 320 nodes for OpenStack Swift (1 proxy, 7 object servers) and 1 Dell PowerEdge 430 controller node connected via

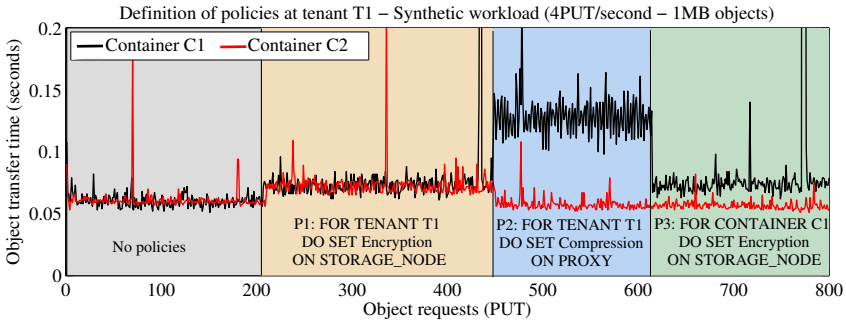


Figure 1: Enforcement of compression/encryption filters.

1 GbE switched links.

Due to the security requirements of `T1`, the first policy defined by the administrator is to encrypt T1 data objects (`P1`). Fig. 1 shows that the `PUT` operations of *both containers* exhibit a slight extra overhead due to encryption, given that the policy has been defined at the tenant scope. There are two important aspects to note from `P1`: First, the execution of encryption on `T1`'s requests is isolated from filter executions of other tenants, providing higher security guarantees [30] (Storlet filter). Second, the administrator had the ability to enforce the filter at the storage node in order to do not overload the proxy with the overhead of encrypting data objects (`ON` keyword).

After policy `P1` was enforced, the administrator decided to optimize the storage space of `T1`'s objects by enforcing compression (`P2`). `P2` also enforces compression at the proxy node to minimize communication between the proxy and storage node (`ON PROXY`). Note that the enforcement of `P1` and `P2` demonstrates the filter pipelining capabilities of our filter framework; once `P2` is defined, Crystal enforces compression at the proxy node and encryption at storage nodes for each object request. Also, as we describe later on, the filter framework tags objects with extended metadata to trigger the reverse execution of these filters on `GET` requests (i.e., decryption and decompression, in that order).

However, the administrator realized that the compression filter on `C1`'s requests exhibited higher latency and provided no storage space savings (incompressible data). Thus, the administrator defined policy `P3` that essentially enforces only encryption on `C1`'s requests. After defining `P3`, the performance of `C1`'s requests exhibits the same behavior as before the enforcement of `P2`. Thus, the administrator is able to manage storage at different granularities, such as tenant or container. Furthermore, the last policy also proves that policies can be specialized per granularity; policy `P2` at the tenant scope applies to `C1`, whereas the system only executes `P3` on `C1`'s requests, as it is the most specialized policy. In summary, this experiment shows the management capabilities of Crystal.

With IOStack, Arctur can exploit a flexible layer to customize object stores according to the necessities of end-users, while saving up administration efforts. The full technical description of this piece of research can be found in Section 12 and have been published in IEEE Internet Computing [15] and USENIX FAST'17 [16].

KPI2: Simplified analytics virtualization. For cloud providers like Arctur, Zoe Analytics solves the problem of easily, quickly and reliably deploying complex, distributed analytic applications on clusters of physical or virtual machines. It does so by implementing a thin layer on top of an existing orchestration back-end that offers easy user access to available applications and resources.

Zoe provides a simple way to provision data analytics applications. It hides the complexities of managing resources, configuring and deploying complex distributed applications on private clouds. Zoe manages applications. An example of a Zoe application is a Jupyter Notebook, with the Python kernel, the PySpark library, an Apache Spark master and several Apache Spark workers. This is a full distributed analytic application that requires extensive systems knowledge to set up correctly by hand. Zoe not only automates the deployment of this ZApp (Zoe application), but also assigns resources automatically and dynamically, based on up-to-date monitoring information. Since Spark workers are elastic (at least one is needed to make progress, if more are available progress will be

faster) Zoe can regulate the number of running workers based on overall cluster usage metrics.

ZApps are defined using a generic, very flexible application description format that lets you easily describe any kind of data analysis application; JSON document and some meta-data.

Thanks to IOStack, Arctur and other companies alike, can fully exploit the computational layer easily. By using Zoe, they are able to abstract the deeper levels of their infrastructure and spawn different types of analytics applications easily with the result of saving administration efforts. The full technical description of this piece of research can be found in Deliverable 5.3 and have been published at IEEE/ACM CCGrid'17 [17].

KPI3: Analytics scheduling: Thanks to Arctur, we know that in private data centers analytic frameworks are an important part of their daily workload. The objective of this KPI is to address the problem of scheduling user-defined analytic applications, which we define as high-level compositions of frameworks, their components, and the logic necessary for the framework to carry out useful work.

In the first part of the IOStack project,[3] we studied modern analytic applications and rigorously defined a new level of abstraction: an application is considered as a collection of frameworks, each with its heterogeneous components. The application is a single entity that needs to be scheduled taking into account the type of each component. Components that are required to produce useful work needs to be started together (core), components that are optional and provide additional performance can be started depending on resource availability and priorities (elastic components).

Given these heterogeneous, composite requests, which are neither rigid, nor malleable (but both), available scheduling heuristics in the literature fall short in addressing the sorting and allocation problems. In our work published at IEEE/ACM CCGrid'17 [17] we propose an approach that improves cluster responsiveness by halving the turnaround time, which is the average time applications spend in the system, and by increasing resource allocation.

During the second half of the project,[4] we built upon these results to further exploit the properties of analytic applications.

In most private or public cloud systems, users gain access to computing clusters by specifying the amount of resources required to run their application, in the form of a reservation request. Upon receiving a request, the cluster *scheduler* decides which application to serve based on the configured policy (e.g.; FIFO, SJF, etc.). Cluster schedulers use a resource management mechanism that is in charge of provisioning and accounting. Given a *resource request*, the resource manager determines its admission in the cluster based on its *reservation* information. Once admitted, the request triggers a *resource allocation* procedure, which eventually [31] concludes with reserved resources being exclusively allocated to the request.

In most system implementations, the concept of reservation and allocation coincide, although neither is representative of the true *resource utilization* a request might induce on the system. In fact, resource utilization is generally not constant throughout a request lifetime, and fluctuates according to application behavior [32]. The main consequence for current cloud environments is that reservation requests are *engineered to cope with peak resource demands* of an application. This is a key factor that induces poor system utilization, and ultimately, efficiency.

Thus, mechanisms to reduce the difference between resource allocation and utilization are needed, for they can prevent clusters from denying admission for new requests which would queue up, while spare capacity goes unused.

Our solution separates the concepts of reservation and allocation. The allocation of a request is dynamically adjusted by the system, thus giving the cluster scheduler more resources to allocate queued requests. Figure 2 gives an idea of the results obtained from this solution, the full description, complete with experimental results, is available in deliverable 5.3.

This work further reduces the turnaround time by 18 times compared to the solution that we presented in D5.2, which was already improving it by two times. We validated our mechanism numerically and implemented it in Zoe Analytics, a real analytic as a service system with production

[3] See D5.2 for details
[4] See D5.3 for details

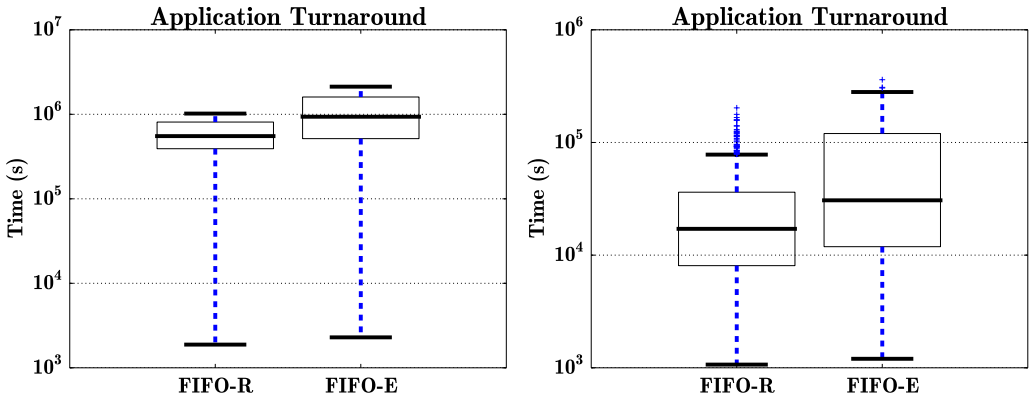


Figure 2: Comparison between an non-dynamic (left) and a dynamic (right) allocation scheduler.

deployments in Eurecom, Air France/KLM and KPMG, among others. Zoe is also being developed as part of IOStack activities.

With the scheduling and resource management mechanisms in Zoe, data centers like Arctur will be able to efficiently run analytics applications, benefiting from better resource utilization (that reduces costs and energy consumption) and lowering wait times, that improves end-user experience. The full technical description of this piece of research can be found in Deliverable 5.3 and have been published at IEEE CLOUD'16 [33] and IEEE/ACM CCGrid'17 [17].

KPI4: Compute/storage orchestration. A datacenter like Arctur consists of a disaggregated infrastructure for analytics with a compute cluster and a storage cluster. In the previous KPIs, we described that in the compute cluster there is a system that virtualizes and runs analytics applications (Zoe); on the other hand, the storage cluster is operated by a software-defined storage system (Crystal), that could enforce services on data flows via high-level policies. Therefore, both Zoe and Crystal are completely isolated components, as they do not interact with each other.

While decoupling compute and storage is a desirable property from a software design perspective, the *workloads executed on an analytics infrastructure impact on both Zoe and Crystal*. This represent a twofold problem: i) being oblivious to this fact may be significantly sub-optimal when running analytics at scale, in terms of performance and resource usage; ii) for adapting the system to the heterogeneity of multiple analytics, system administrators should bring upon their shoulders the weight of properly managing and configuring compute and storage components within a datacenter [34, 35]. This yields that, in a complex analytics stack, building a substrate for cross-layer optimizations may be even more important that optimizing compute and storage subsystems independently [36, 37].

To solve this problem, the architecture of IOStack favors the creation of cooperative strategies between analytics instances and the storage cluster to tailor and optimize specific analytics workloads, involving low administration overhead. This is possible thanks to the implementation of custom "controllers" that are capable of configuring data services in Crystal, thanks to real-time information coming from Zoe, or even user-based "hints" defined in the deployment descriptor of an application.

With coordinated compute/storage orchestration controllers, Arctur administrators will be able of optimizing both compute and storage clusters under complex workloads with low effort. The full technical description of this piece of research can be found in Section 14 and it has been submitted for publication.

4.1.3 Exploitation of Results

Arctur intends to take advantage of the various components of IOStack in different measures based on the functionalities that this component offers and on the type of service over which the component will be implemented. Arctur therefore plans to deploy Crystal to better manage their object storage system and to optimize their storage "consumption" based on the data retrieval rate.

Additionally to automating processes on the Cloud infrastructure, Arctur applied the IOStack system on their web services. Arctur produces and hosts a very large amount of web portals. The

data that is stored on this web portals is less frequently accessed with time. Using fast storage to serve data that is rarely accessed represents a big cost and inefficiency for Arctur. For this reason Arctur applied an automated filter on their web servers. Data is now tiered based on the time when data was uploaded and how frequently it is accessed. New and more accessed or popular files are stored on Tier0 storage while older and less used files are automatically migrated to slower storage after a predefined amount of time. If this data is still requested it just gets cached and is readily available for the time it is needed. By applying this rules Arctur has optimized their storage utilization.

Likewise, compute orchestration components of IOStack have some promising features for Arctur. There is a series of use cases where Arctur's end users might get major benefits by having the option to rapidly deploy a large infrastructure based on a number of small (virtual) servers, this is particularly interesting in such cases where traditional physical HPC servers are not flexible enough. Arctur will keep following the development of Zoe Analytics in order to be able to offer its advantages to their clients.

4.2 GridPocket: Towards Faster and More Efficient Data Intensive Analytics

4.2.1 Use-case Description

GridPocket is an innovative software-as-a-service solution oriented company focused on development of energy value-added services and platforms for the smart grid utilities. The solutions of GridPocket include applications for energy management, demand response control software, M2M and behavioral experts systems for electricity, water and gas utilities. GridPocket distributes its applications through partnerships with energy distributors, ESCOs (energy saving companies), equipment manufacturers and utilities worldwide. The business model of GridPocket is B2B2C, white label and Software as a Service (SaaS). GridPocket's applications enable end-users to take full control over their energy spending and reduce their CO2 emissions.

GridPocket's applications gather and store energy consumption data. Currently, GridPocket uploads consumption data every 30 minutes for thousands of users. But in the short term, it will be required to upload millions of users' consumption every 10 minutes and in the long term, to upload millions of users' consumption in "real time". GridPocket's solution allows each user to display his consumption history, real time comparison to neighbors and analytics about consumption behavior of given areas such as average, min/max consumption over a period of time (e.g., days, years).

The current solution, although fully adequate with current customer needs, has *serious scalability constraints*. That is the reason why GridPocket plans to move towards using object storage and Spark SQL tools. The important number of consumers' data that have to be uploaded and stored in GridPocket's databases needs a large bandwidth. Performing analytics requests will require high performance to process large amount of data ingestion. Furthermore, gathering relevant information from databases sometimes needs important number of requests to be processed.

4.2.2 Demonstration of KPIs

KPI5: File filters. IoT use cases like GridPocket create large collections of objects that require scalable and low-cost storage, such as object storage. In turn, such amounts of data are then ingested and computed by analytics engines (e.g., Spark SQL). In this scenario, KPIs for Spark SQL queries in terms of performance and cost are the amount of data ingested from the object store to Spark and the number of incurred REST requests ingest such data.

This motivates the research done in KPI5 in which we developed Spark File Filter: a new pluggable interface for Spark SQL, which allows applications to implement custom logic to optimize the execution of queries on file based storage systems. Spark File Filter extends the existing Spark SQL partitioning mechanism and enables to dynamically filter irrelevant objects during query execution. Our approach handles any data format supported by Spark SQL (Parquet, JSON, csv etc.), and unlike pushdown compatible formats such as Parquet which require touching each object to determine its relevance, it avoids accessing irrelevant objects altogether.

The main idea behind Spark File Filters for object storage is similar to the traditional index mechanism in relational databases. One the one hand, at the storage side our mechanism build a user-

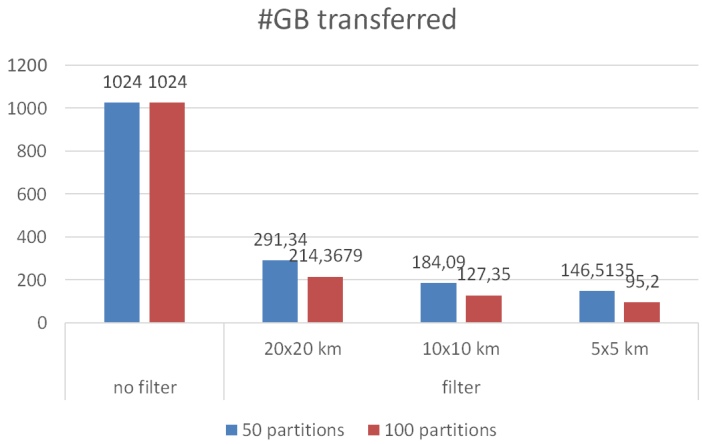


Figure 3: Performance of file filters in terms of GBs ingested.

defined index of data stored at the object store. This index is maintained in an online service that enables fast lookups to clients. On the other hand, we augmented Spark SQL with a hook whereby user defined filters can restrict the files which need to be sent from the storage service (e.g. OpenStack Swift, Amazon S3, IBM Cloud Object Storage, HDFS) to Spark. Therefore, instead of filtering data object, our mechanism allows Spark SQL to only requests the necessary objects to satisfy a certain query, thus minimizing the load against the object store.

We implemented GridPocket's filter which screens objects according to their metadata, for example geo-spatial bounding boxes which describe the area covered by an object's data points. This leads to drastically lower KPIs since there is no need to ship the entire dataset from the object store to Spark if you are only comparing yourself with your neighborhood.

As a proof of concept, Fig. 3 shows data ingestion results of 2 Spark nodes (32 cores and 128 GB memory) operation over a 1TB GridPocket data stored at IBM Cloud Object Storage (COS) [38] service. In this experiment, we executed 10 randomly located SQL queries via Notebook that exploits the underlying Spark File Filters. As can be observed, GridPocket analytics notebooks obtain data reduction values from 2.51x to 9.75x, depending on the data partitioning. We also achieved query speedups up to 9.57x.

This mechanism enables GridPocket to execute SQL queries faster and more efficiently. The full technical description of this piece of research can be found in Deliverable 4.3 and it has been presented in Spark Summit'17 [18] and submitted for publication.

KPI6: Object storage connectors. For GridPocket and many other companies, object storage is the persistence layer of choice for analytics in the cloud; however, the Spark/Hadoop ecosystem has focused on interfaces based on HDFS, which is a distributed file system. In other words, the community tends to treat object stores as a file system.

However, due to their stateless design (e.g., no metadata service), object stores have differing semantics compared to file systems: in particular, rename is not a native object store operation and is very expensive since implemented as a copy operation followed by delete operation. Write workloads spawn multiple rename operations which greatly harm their performance.

Even worse, existing object storage connectors may fail: Spark, through Hadoop, renames temporary files to achieve fault tolerance and enable speculative execution. But this does not work for eventually consistent object storage: a container listing may not yet include a recently created object. Thus, an object may not be renamed, leading to incorrect results. These problems caused inefficient and unstable upload path to Object Store and therefore were barriers for many companies to reliably use object stores.

One of the early outcome of IOStack is Stocator: an advanced object store connector for Spark which uses object store semantics to achieve high performance and fault tolerance. It eliminates the

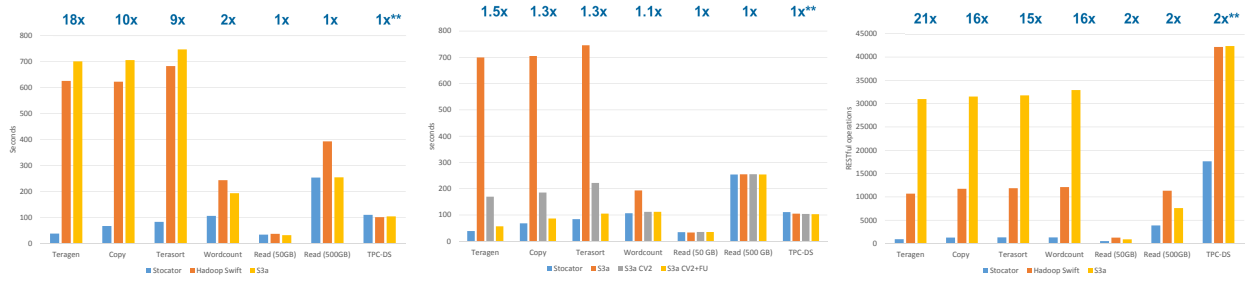


Figure 4: Standard benchmarks executed against an object store (IBM COS) that compare write speedup (left), read speedup (middle) and number of REST operations (right) of Stocator and other community object storage connectors.

rename paradigm, and writes each output object to its final name; this includes an attempt number and is unique per attempt. Stocator extends an already existing success indicator object written at the end of a Spark job, to include a manifest with the names of all the objects that compose the final output. A subsequent job will then correctly read the output, without resorting to a possibly inconsistent list operation.

To justify KPI6 in Table 1, next we provide some experimental results. The setup of our performance comparison experiments is as follows: We ran the experiments over bare metal Spark and IBM COS clusters. The Spark cluster has 3 servers with a total number of 144 cores. The IBM COS cluster has two Accessers and twelve Slicestors, where each Slicestor has 12 data disks. The erasure code is $(12, 8, 10)$, meaning that each data segment is split into 12 pieces, where 8 are needed to reconstruct the data and a write returns success after at least 10 pieces have been written. IBM COS exposes both the S3 API and the Swift API so that we could measure the performance of both in the same environment.

Fig. 4 compares the performance for common benchmarks and state-of-the art object store connectors. Fig. 4 (left, middle) shows that Stocator is much faster for write-intensive workloads (up to 18x faster) and similar or faster than other connectors for read-intensive workloads (up to 1.5x faster). Indeed, one of the main reasons for the performance improvements of Stocator lies in the optimization of communication costs with the object store. Fig. 4 (right) shows that Stocator reduces the number of REST operations of the executed workload between 2x and 21x, which yields that the object store is in turn able to serve faster more clients.

The Stocator connector is in production use at IBM from 2016 in the Bluemix service. Thanks to this connector, Gridpocket can now easily upload its IoT data to object storage thus easing the whole process of data analytics. The full technical description of this piece of research can be found in Deliverable 4.3 *and has been published in ACM SYSTOR'17 [19] (poster) and ACM SoCC'17 [20] (poster).*

KPI7: Computation close to data. One of the novel features of IOStack is to provide a platform to deploy data services (or "filters", as we describe in Section 5) at the storage side. Such filters may have different purposes depending on their implementation. However, one of the main veins of research in IOStack has been to exploit such filters to accelerate data intensive analytics by executing computations close to the data. Next, we briefly describe the techniques used to compute close to the data in IOStack and their results in the GridPocket use-case.

As a first step in this direction, we translated a concept akin to "predicate pushdown" in the traditional database literature [39, 40] into a disaggregated analytics ecosystem. Our implementation enables efficient execution of SQL queries on raw Comma-Separated Value (CSV) data stored in OpenStack Swift, to accelerate GridPocket analytics. At the analytics side, we extended the CSV data source in Apache Spark, which can now offload SQL projections and selections on parallel object requests against Swift. At the object store, we contributed to the OpenStack Storlets: a framework to intercept and execute sandboxed code on object requests in OpenStack Swift (see Section 11). Among other things, we extended Storlets with the capability of efficiently executing computations close to

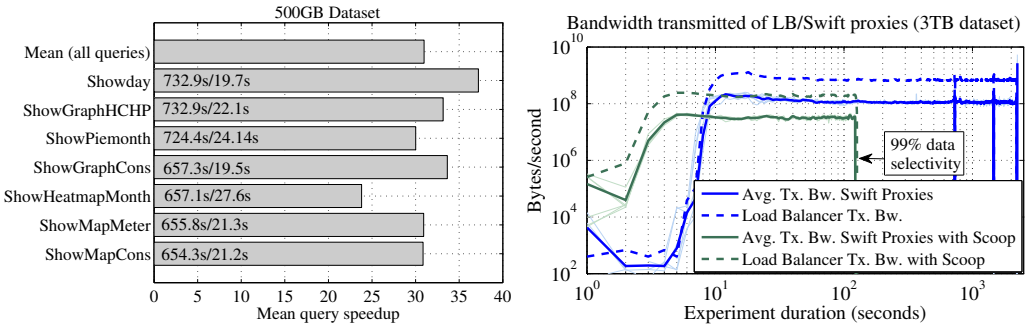


Figure 5: Speedup of typical GridPocket queries with our SQL pushdown storlet (left). Consumption of bandwidth resources with/without the execution of our SQL pushdown storlet (right).

the data. Moreover, we created a new Storlet code to perform the SQL projection and selection filtering on CSV objects. Spark SQL interfaces with data source implementors, such as Spark CSV, with an API which permits to off-load selection and projection filtering tasks. Thus, while Storlets allow to run arbitrary code on the object store, we concentrate on the filtering aspects of SQL queries.

To evaluate this technique, we executed extensive experiments on a 63-machine cluster —a collaboration with the OpenStack Innovation Center [41]— over real IoT data from GridPocket energy meters. We summarize our results in Fig. 5. First, this mechanism in IOStack can accelerate the end-to-end SQL processing time on the semi-structured data by 4x up to 37x, depending on the dataset size and amount of filtered data by the query (Fig. 5, left). Moreover, apart from achieving significant query speedups, this mechanism also can greatly reduce resource consumption in disaggregated compute/storage clusters. For instance, Fig. 5 (right) shows the effects in network of filtering 99% of data at the storage. Even more, the consumption of CPU and memory that Spark exhibits to compute queries become also reduced, in exchange of consuming some spare CPU power at the storage side.

Therefore, IOStack enables GridPocket to benefit from the scalability of object storage, while making their analytics workloads much faster and more efficient. The full technical description of this piece of research can be found in Deliverable 3.3 *and have been published in IEEE ICDE'17 [21].*

While the previous approach was good to accelerate SQL queries, we found that our filter abstraction in IOStack could provide a more dynamic, general-purpose mechanism to execute computations on data streams close to the data. In fact, we realized about the synergy between the dataflow programming model of computing engines like Spark or Flink and the traction of stateless computations (namely, *lambdas*) that some cloud services already offer (e.g., AWS Lambda, IBM OpenWhisk).

To this end, we created a Storlet filter in IOStack that enables to "pushdown" and execute lambda functions passed by parameter on object data streams. For the sake of clarity, in this context we refer as a lambda function —also known as closure in programming languages— as a self-contained piece of logic that operates synchronously on data stream records and can be chained to form a pipeline. In particular, our implementation uses the Java 8 Stream API [42] to create data streams and operate on them with a rich set of calls.

Our objective is to allow administrators to define policies with lambda functions as input for the filter. To wit, in IOStack a GridPocket data analyst could write the following policy to discard all the lines in a CSV file that do not contain "Paris", as they are not necessary for a specific analytics application:

```
FOR CONTAINER C1 DO SET LAMBDAS WITH lambda1=filter(s → !s.contains("Paris"))
```

As a result, administrators would be able to discard unnecessary data during the ingestion phase of analytics; for instance, they can define lambdas using the `map` call to discard rows of a dataset, or the `filter` call to discard lines within an object based on a predicate. But as one can infer, the available functionality on data streams goes far beyond this: and administrator can select the `max` value of a file, can execute `reduce` calls or even get the `distinct` values of data object, among other

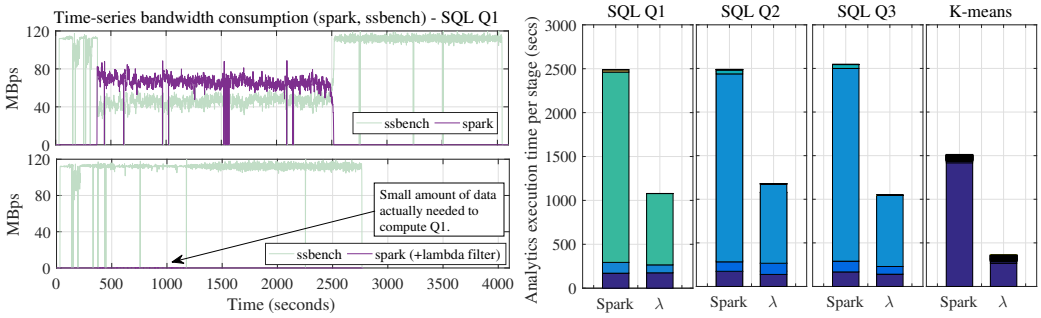


Figure 6: Consumption of bandwidth resources at the proxy (left) and analytics execution times (right) with/without the execution of our lambda pushdown filter in a multi-tenant workload.

functionality. As visible in the policy above, lambdas are defined in the form of tuples to express the order of the function will be executed and its body/type. This enables an administrator to control the pipeline of lambda functions to be executed on a data stream.

To evaluate our approach, we ran our experiments in a cluster formed by 4 Dell PowerEdge 430 nodes for Spark, 8 Dell PowerEdge 320 nodes for OpenStack Swift (1 proxy, 7 object servers) and 1 Dell PowerEdge 430 controller node. All nodes are connected via 1 GbE switched links. To emulate a multi-tenant scenario, we ran one `ssbench` workload (24 threads) executing 20K 16MB GETs in one node representing an IO intensive tenant, as well as 3 compute nodes —each one contributing with 12 CPUs and 28GB of RAM— running Spark 2.11 in cluster mode as a tenant executing analytics. In Spark, executed some queries of GridPocket on IoT dataset (140GB) as well as a k-means job on a real cloud service log traces (100GB, UbuntuOne [43]).

In first place, the code below shows how a data analyst can define lambdas in a policy to perform computations close to the data that filter data for a GridPocket query (SQL Q1). This code is received by the Storlet filter, compiled on runtime and then executed on the data stream of the object upon a GET request. As a result, we can obtain important data transfer reduction. For example, the code below reports a data transfer reduction of 99.96% to compute a query that "gets the aggregated energy consumption per energy meter of Paris meters in January 2015".

```
map(s -> { List<String> l = new ArrayList<String>(11); String[] a = s.split(",");
for (int i=0; i<11; i++) //GridPocket dataset has 11 columns separated by commas
  if ((i==0 || i==1 || i==5 || i==7) && i<a.length) l.add(a[i]); else l.add("");
return l; } //We only need 4 columns and rows of January 2015 and meters located at Paris
filter(l -> l.get(0).startsWith("2015-01") && l.get(7).equals("Paris"))
```

The upper plot in Fig. 6 (left) clearly show the problems of multi-tenancy in a shared object store. That is, Spark starts the ingestion phase with an evident consequence: `ssbench` receives less than the half of the bandwidth, given that it has lower parallelism (24 `ssbench` threads vs 3x12 Spark workers). The worst aspect to take into account is that most of the bandwidth consumed by Spark is actually useless data that is discarded by the tasks during the compute phase. Fortunately, the lower plot of Fig. 6 (left) shows the benefits of executing the lambdas at the storage side: `ssbench` can now benefit from almost all the bandwidth during the experiment, while Spark only ingests the required data.

In this sense, when bandwidth resources are scarce and shared, we can observe that executing lambda functions to discard data at the object store provides important benefits in terms of application completion times. To illustrate this, in Fig. 6 (right) GridPocket queries exhibit a speed-up ranging from 2.12x (Q2) up to 2.43x (Q3), depending on the query at hand. Even more, the k-means job experienced a speed-up of 3.95x thanks to the lambda pushdown.

Now, GridPocket can exploit a more flexible approach to computation close to the data, thus being able to accelerate other types of data intensive analytics. The full technical description of this piece of research can be found in Section 12 and it has been submitted to a journal (extending the USENIX FAST'17 paper).

KPI8: Automatic lambda migration. Until now, the IOStack computation close to the data mechanisms are explicitly used by GridPocket data scientists. That is, GridPocket data scientists should

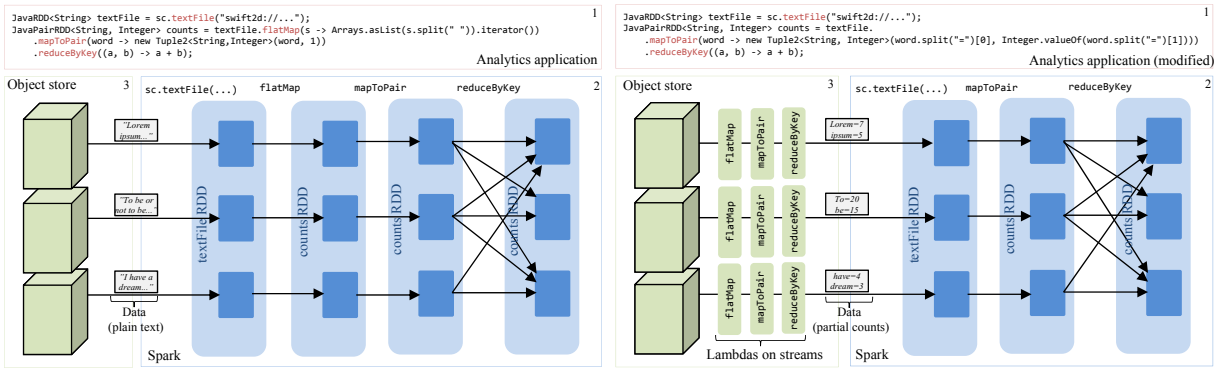


Figure 7: Example of (1) an input wordcount application, (2) the DAG execution of Spark and (3) the data ingestion process from an object store (left). Rationale behind IOStack: (1) To modify the input application, and (2) migrate some dataflow computations as lambdas in object streams (right).

take care of configuring the computations at the storage side that could benefit the performance of analytics at the compute side. We want to extend the automation capabilities of IOStack so it can choose which computations from an input application can be migrated to the storage side.

To this end, we extended IOStack with a new abstraction, namely *job analyzers*. Basically, a job analyzer receives as input an analytics application and has two main tasks: i) to identify operations within the code that are suitable for execution at the storage side in a stream fashion, and ii) modify the input application accordingly (if necessary). Our motivation is to blend both dataflow computing (Spark, Flink, etc.) and lambda processing models as a unified processing engine.

To better understand this, let us show an example. Fig.7 (left) depicts the code and execution of a regular `wordcount` Spark application (Java). The program first declares a RDD that contains the lines of a set of text data objects stored remotely (`textFile`). Then, a second RDD (`counts`) is aimed to actually contain the counts of each word in the dataset by applying a set of transformations on textFile. That is, we first split each line into words (flatMap), and then we map each word into a (word,1) pairs that are further aggregated via a reduceByKey transformation. As visible in the application DAG of Fig. 2, all these operations are executed in parallel on data partitions.

The objective of job analyzer is to transform from situation Fig.7 (left) to Fig.7 (right), which yields to move some of the transformations on Spark RDDs at the storage side and adapt the input application accordingly. That is, in Fig. 7, we observe that the input application is different from the original one. Now, Spark assumes that the data is a set of `(word,count)` pairs instead of plain text, which are aggregated later on (`reduceByKey`). To this end, storage nodes execute a pipeline of lambda functions per data object according to a set of dataflow operations defined in the input application (`flatMap`, `mapToPair` and `reduceByKey`). In this example, storage nodes are essentially retrieving word counts of each data object that are then summed up by Spark. Such a mixture of stream processing and dataflow computing has two main benefits: i) the application DAG in Spark is simplified and consumes less resources by amortizing spare computing power at the storage side; ii) and even more importantly, data ingestion is heavily reduced (with this approach, a wordcount on Wikipedia dumps reduces data ingestion in $\approx$ 95%).

With this mechanism, GridPocket analysts can transparently exploit the compute close to the data capabilities for different types of data-intensive applications. Moreover, this contribution enacts IOStack as a platform to develop new application analyzers that accelerate applications for other compute engines. The full technical description of this piece of research can be found in Section IV and have been submitted to a high-quality conference.

4.2.3 Exploitation of Results

GridPocket is collaborating with IBM for possible exploitation of the Spark File Filters, that could be used to reduce data ingestion computation by analytics engines such as Spark SQL. The migration

of technologies in Gridpocket towards object storage and Spark will also involve the use of Stocator for optimizing object storage interactions in the daily activity of data scientists. In this setting, Gridpocket would be interested in using Storlets for exploiting computation close to the data techniques, but at the current time, most important needs are Spark File Filters and Stocator.

All these solutions help GridPocket to be able to update database consumption with the files received all the days in very short time. They also enable it to reduce latency of analytics requests. For a commercial use perspective, these solutions to GridPocket's given problems will be integrated to GridPocket architecture to enable fast display of information to either end-users and data scientists.

4.3 Idiada: Automating Workflows in a Automotive Computations

4.3.1 Use-case Description

Applus+ Idiada is a multinational company providing design, engineering, testing and homologation services to the automotive industry. It has over 1800 employees in 23 countries. The company was first established in 1971 as Idiada or Institut d'Investigació Aplicada de l'Automòbil (Institute for Applied Automotive Research) at the Polytechnic University of Catalonia. In 1990, Idiada was separated from the university and established as an independent company owned by the Government of Catalonia. It was privatized in 1999 as a company owned 80% by Spanish company Applus+ and 20% by the Government of Catalonia.

Among its activities, Idiada design engineering simulations involve the storage of pre-processed physical vehicle models to run a set of simulations over them. These models storage implies data management costs which have direct impact on the company budget and the simulation process workflow performance. These models are stored over virtual volumes offered by a storage cabin through NFS. These NFS units are mounted on the Sun Grid Engine execution hosts and engineers' virtual machines in order to made simulation models data available to the solvers and run the simulations. Due to the increase of Idiada design engineering projects and its related simulation data growth, the improvement of Idiada's system scalability is becoming a critical task, involving storage space reduction and data management costs savings.

Idiada has two main problems in the analytics and storage workflow of car simulations: i) *Storage costs*, as the output files of car simulations produce huge amounts of data that are currently stored in expensive storage arrays. ii) *Storage management flexibility*, given that the work-space model for the organization of Idiada's projects is currently unable to customize functionality on files and folders; for instance, performing access control or compressing specific files within a project are currently costly operations from an administration viewpoint.

4.3.2 Demonstration of KPIs

KPI9: SDS for block storage. As an engineering company in the automotive sector, Idiada manages large amounts of data generated by computer simulations via a file system interface. Such file systems are mounted on top of virtual block storage volumes that physically reside in storage arrays.

One of the main problems in activity of Idiada is to manage such block volumes and apply new functionality on them. That is, if an administrator in Idiada aims at creating a new storage volume with a specific functionality, such as encryption or compression, due to a client's requirements, it involves important storage management activity. This implies that administrators should be dealing with such low level storage management, leading to additional costs for the company. The root for this problem is that most storage arrays are not "programmable", so an administrator or a system developer can extend their functionality to satisfy new requirements.

To solve this problem, IOStack provides a SDS component for block storage: Konnector. Konnector is a framework for connecting storage array volumes to server compute nodes. First, Konnector's API attaches SAN Storage Array Volumes (SAV) to a compute server node over FC or iSCSI SANs in a policy-based manner. This means that Idiada administrators can manage storage volumes easily, group them, and define static configurations to accommodate their requirements.

But even more importantly, Konnector's API allows a user to insert data processing functions between the SAV and the compute node application. These in-line functions that operate on the data

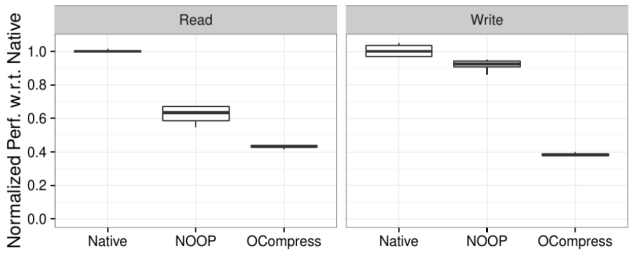


Figure 8: Performance on Idiada dataset using OCompress.

flow are called "storage filters", and represent an important advance: administrators can now add new functionality to storage volumes without changing the closed controllers of storage arrays. To this end, Konnector provides an Software Development Kid (SDK) to develop storage filters which are deployed as ".so" DLL functions. Filters are dynamically inserted by the Konnector run time manager into the data flow between the compute server node and the storage array. Filters can be stacked so that the data flows across a number of filters in a filter stack. As we show later on, a filter stack example could be IOPS/BW filter, encryption filter and multi-copy.

As an example, let us imagine that a user wishes to connect over iSCSI a storage volume and insert in the data flow an in-line processing function such as encryption or compression. The user would write his inline data processing function using the Konnector SDK and deploy to the Konnector filter directory. Using the REST API Konnector can now discover and attach a storage array volume to the server node, instantiate the filter function in the data flow and present a virtualised device to the application. This example demonstrates how storage functionality can be extended on the compute server node without any change on the storage array.

Konnector allows Idiada administrators to manage block volumes easily, as well as customize functionality per block volume. The full technical description of this piece of research can be found in Deliverable 3.3 *and it has been submitted for publication.*

KPI10: Data reduction. For Idiada analytics activity, saving storage space is a critical aspect to consider given the amount of output generated by automotive compute models. Intuitively, satisfying such necessity can be done via storage filters that implement either deduplication or compression, or a combination of both. In this project, we implemented data reduction filters for both object and block storage; next, we describe our experience with block filters for data reduction with Konnector.

In this case, the space can be saved directly (in the device) or indirect (in the memory). If the space savings are direct, there is a direct reduction in cost ($), if it is indirect it may increase performance as we are using less memory, thus increasing the memory that can be used to cache other files.

Idiada typically does not reuse datasets in a short period, so our memory compression and memory deduplication filters have no benefits on their workloads. The deduplicable filter does not show benefits on the used dataset as it is not deduplicable. However, if we switch to a compression schema, the benefits are translated directly to a 60% of more cache memory (shared resource improvement) thanks to the compression of the data set.

As the reutilization of the dataset is low, we created an output compression filter. The output compression filter generates a compressed filesystem inside the virtual volume (transparently). The compression level is the same (as the compressor technology is the same), a 60%. However, as the storage is fast, there is a penalty for including compression on the data flow of a 50% (Figure 8). This penalty, in our tests with slower devices or faster compressors, can be eliminated and on some scenarios we obtain more throughput. We can see that using a NOOP filters has important penalties, which calls to further develop optimizations for that filter (threading, buffering or prefetching).

Block storage filters may help Idiada to survey different ways to reduce storage space consumption, either permanently or temporary using memory usage reduction techniques. The results for block storage filters, can be found on D3.3 and have been submitted for publication.

Table 2: Access latency improvement with JIT on Arctur Web workload.

Download Time improvement	100Mbit/s	200Mbit/s	300Mbit/s	1Gbit/s
small	22.2%	23.5%	0%	5.5%
medium	44.8%	32%	15%	15%
large	69.5%	59.8%	31.7%	10.2%
Total	37.5%	23.07%	4.76%	4.76%

KPI11: Prefetching, caching. Partners like Idiada or Arctur may also benefit from storage filters that improve performance in object storage, as they make intensive use of it in various scenarios (e.g., cold data storage, Web servers). In this case, we demonstrate a prefetching filter, as we want to reduce the latency to get data from the data sources to the clients.

For our use-cases, prefetching is an important feature when getting the data has a high penalty. In local storage, the kernel itself tries to prefetch the next blocks of sequential accesses so we can reduce several orders of magnitude (ms to ns) the time to access the data. Prefetching in object storage environments, is similar, we may want to start moving data from the object servers to the proxy or even the clients if possible. Object storage environments offer a good place to use prefetching, moreover, if we combine it with SDS the user or the system can enable or disable the prefetching depending on the workload, or the obtained metrics.

This is important as the benefits of prefetching are dependent on the speed of getting an object from storage, if it is low prefetching will accelerate some metrics. For instance, if getting an object is fast, prefetching may introduce overheads and it is better to disable it. On the other side, big objects and low speed networks will also introduce overheads if we do not have enough time to do the prefetch. Therefore, we developed a storage filter for object storage as a native filter for Crystal and Openstack Swift called JIT prefetching. The prefetch system is able to preload objects just before they are used in order to reduce the quantity of memory used for caching.

We justify this KPI via experiments based on a real trace provided by Arctur that can be found in IOStack website. The Arctur trace captures 2.97TB of a read-dominated (99.97% read bytes) Web workload consisting of requests related to 228K small data objects (mean object size is 0.28MB) from several Web pages hosted at Arctur datacenter for 1 month. In this particular experiment, we augmented the size of files (10x) to better illustrate the impact of pre-fetching. All the tests presented in this work are executed in the IOStack testbed (`http://testbed.iostack.eu`) provided by Arctur. Swift (Kilo version) installation consists of three single HDD storage nodes connected through 3 GbE plus one proxy node connected with 1 Gigabit Ethernet to the clients. We executed a workload replay process (SwiftWorkloadExecutor) a client node requesting to the proxy node, thus outside the 4 Swift machines cluster.

As a preparatory phase for the experiment, we need to execute the trace but pushing the objects to Swift in order to be able to perform `GET`s afterwards. Using this SwiftWorkloadExecutor we are also able to modify the trace in order to increase the file size, sort the trace, introduce some errors (changing requests times, order, etc.).

In Table 2, we obtain a mean benefit of a 5% in terms of download times when the cluster is totally available for the execution of our experiment. However, in the real world, an object storage cluster is periodically executing background maintenance tasks that slow down the throughput of storage nodes. To emulate this situation, we executed the Arctur workload under different degrees of network congestion between storage nodes and proxies. In this case, we note that under high utilization of the internal storage network (e.g., replication processes, data durability checks, etc.) the improvements from JIT prefetching may reach up to 69.5%, depending on the size of downloaded files. This speed decrease also implies that the prefetched data is obtained slower from the proxy and

the prefetch system needs to adapt (automatically) to the new speeds.

Idiada and Arctur can reduce the latency and increase the throughput of their repetitive workloads thanks to a dynamic (via Crystal) filter than learns and adapts to the medium. The JIT prefetcher will be presented on a future paper, including other workloads. More information about JIT prefetcher can be found in Appendix C.

KPI12: Client file system control. In Idiada design engineering's department, a set of teams along with various business units must share projects and data for work on them using a file system interface. However, with increasing urgency, Idiada needs a way to enforce specific functionality on folders and files within projects, depending on the requirements of their clients. The requirements may be disparate: for instance, Idiada needs some kind of access control due to our client's confidentiality agreements, so engineers can only access to a certain subset of files. Other projects may need data at rest encryption due to security concerns. As another example, some files of a project may require to be processed by specific or even proprietary compression engines in order to save up storage, due to their proprietary binary format.

To give a solution to these requirements, in IOStack we created a framework to intercept file system operations on clients based on FUSE file systems. In particular, we extended the S3QL file system to be able of i) intercepting user operations, and ii) execute specific functionality, namely storage filters, on user operations. Technically, S3QL is a FUSE file system for object storage: this means that with Crystal and our client filters we have end-to-end control of object storage requests. Moreover, S3QL filters can be orchestrated from the IOStack object storage dashboard, thus having an integrated SDS framework for object storage.

To demonstrate the end-to-end orchestration of object storage requests, we equipped the FUSE client with the ability to add custom tags to object storage requests. It allows administrators or users to put tags on a file so they can process it in a special way in the server side. For example, a `SECRET` tag in a file may prevent access or encrypt the contents of the file at the object storage side by triggering a Crystal filter. The key point here is that the FUSE client has the complete metadata view of the file-system (which is not available at the object store), so it can provide fine grained tagging on specific files and folders, thus providing the required management flexibility to Idiada projects.

But not only this, the FUSE client can also plug and execute custom logic at client machines, or even redirect requests to a specific server if necessary. To demonstrate the practicality of these features, we make use of the FUSE client with different compression engines. First, we implemented a gzip compression filter at the client side for Idiada files that results in a compression of 4.1 G of a dataset of 5.7GB (28.07% data reduction) with a state of the art compressor (7z) we have a 31.6% of data reduction. It is expected that with the special compressor from Idiada we could get higher compression levels.

On the other hand, the FUSE client may also redirect or treat requests differently according to certain rules. For instance, Idiada owns a proprietary compression engine for special Idiada simulation results files. Thus, in order to keep only one node running such proprietary compressor (which has a per-node license), we created a special filter (filterqsub) upon the upload of files of such specific format. In that case, the FUSE clients forwards the upload of a file to an external job queue where the proprietary compressor resides. Thus, the compression engine dequeues the file request, compresses it and stores it to the object store. As can be inferred, the data compression performance of executing the proprietary compressor at either one node or at each FUSE client is the same, but the running costs for Idiada's infrastructure are lower. These are clear examples of the management flexibility that the FUSE client framework for object storage provides to Idiada.

With the filtering and control capabilities of our FUSE client, Idiada is able of satisfying varying requirements within an object storage environment while keeping its file system, workspace-based working methodology. The results for S3QL framework and filters can be found in Section 9.

4.3.3 Exploitation of Results

Idiada's use case is mainly focused on providing a solution for the CAE/CAD department as well as other departments like ADAS (Advanced driver-assistance systems) or Electronics. All of them work in conjunction with several business units worldwide, thus they need to share project's information.

Currently they work accessing the main storage and creating copies of the files in their respective local storage. This working methodology may cause some versioning problems and someone can use an incorrect file's revision. IOStack provides the solution to access a single environment which, based on policies defined by an administrator, will grant or deny the access to the information with a high level of granularity and flexibility.

Some Dummy models are licensed by a period of time, and there is only a contractual agreement which compromises Idiada to remove all the Dummy files after the end date. With IOStack Idiada is able to define a policy for that kind of special files that will enforce the compliance with the agreement. Also Idiada is able to forbid the access to those Dummies based on restrictions of the agreement. For instance, some dummies are available only to some countries.

Idiada will use the features provided by IOStack allowing them to define:

- *Encryption*: Some files may be encrypted based on the kind or content.
- *Compression*: Idiada uses some special compression utilities based on file type. It will execute a process to compress the file based on the policies defined by the administrator.
- *Placement policies*: Based on the container's rules, Idiada will place each project at the business unit which needs to access.
- *Access to files depending on the source network*: Some files can only be seen on well-known networks. Idiada will define some policies based on Security Policy Levels.
- *Access to files based on the kind*: Idiada may define access rules based on file type. Some users may not access file results or models in a project, but may access reports.
- *Removal of files based on rules*: For instance, period of time, file type.

IOStack makes all end users access the main information repository, allowing them to concentrate on their main task: Engineering. It also allows Idiada to fulfil the security policies required by an industry with high security requirements, an industry where secrecy is key.

The same departments simulate the engineering models in an HPC cluster. Each node in the cluster has a local storage which accommodates the temporary files whilst the simulation is running. That cluster is currently deployed as static computers with a Grid Scheduler to fit the simulations in a set of nodes. This causes Idiada to be slow with the changes requested by the engineers, and thus all reconfigurations need to be done statically.

With block storage and the Konnector manager Idiada can reconfigure the system dynamically and deploy file systems to the nodes with an automatic configuration. With the filters Idiada may define some policies depending on the kind of the simulation or the project. They will be able to define as much file systems as they need on each simulation, isolating the scratch per each simulation. This will increase the confidentiality in the scratch system. It will split the job of the Grid Scheduler and make each process simpler. Idiada will be able to define filters that apply to the content of the scratch folder and that will allow them to speed up the preparation phase.

Part II
The IOStack Toolkit

5 IOStack Toolkit: A Software-Defined Storage Stack for Big Data Analytics

In the following, we revisit the main design concepts already presented in deliverable D2.3 to understand how they have been instantiated at each building block prototype. We also describe the progress of the building blocks of the IOStack toolkit during the last year of the project.

5.1 Revisiting Design Concepts in IOStack

Following the principles of SDS, the IOStack toolkit is designed to decouple *control and data planes*. The control plane is intended to expose simple means for enabling administrators to orchestrate the underlying system, even resorting to intelligent, real-time algorithms. On the other hand, the data plane executes the actual logic on live workflows to enforce the services defined by the administrator at the control plane.

Concretely, to instantiate such a general design model, in IOStack we propose several abstractions. At the data plane we find the *metric and filter* abstractions; at the control plane, IOStack provides a *controller and policy* abstractions.

Filter[5]: In IOStack, a storage filter can be defined as a performance control or general-purpose data transformation that applies to specific data flows. This abstraction is quite general, as it can range from data compression or caching filters to IO bandwidth differentiation. Overall, the idea is that filters extend the functionalities of a storage system to meet requirements non-anticipated in their design.

Metric: This concept represents information of a particular aspect of the system operation at runtime. One can think in workload metrics that describe in real time some characteristics of the workload at hand, such as the in/out bandwidth of a tenant, the number of IOPS of a volume, and so on. But metrics can also refer to the usage of the underlying resources, such as the CPU consumption or disk usage of storage servers.

Controller: A controller is an algorithm that receives as input workload metrics to manage the behavior of the system at runtime.

Policy: Contract with the SDS system to provision a service/resource to a tenant. This is the main mechanism for datacenter administrators to interact with the IOStack toolkit.

A key strength of IOStack is to simplify the management of data storage and analytics application deployments via *policies*. To enforce complex policies, the IOStack control plane builds a distributed layer to deploy arbitrary controllers; to wit, a developer can design a controller to manage the execution of storage filters, or controllers for dictating under which conditions an analytics deployment should scale. In this sense, IOStack controllers use runtime workload or resource metrics to take dynamic decisions; for instance, the IO bandwidth of containers/volumes or the CPU usage of compute instances. Moreover, in terms of storage, IOStack leverages the *storage filter* abstraction as a mean of executing computation of storage flows to optimize or provide added value services on specific workloads, such as data compression, caching or bandwidth differentiation.

In the following, we will overview the building blocks that constitute the IOStack toolkit, and we will describe how these building blocks mapped the previous design concepts in their respective domains.

5.2 Overview of the Toolkit

This section provides a high-level overview of the building blocks of the IOStack toolkit, as well as the relationships among them. As visible in Fig. 9, the architecture of IOStack toolkit can be divided into four main building blocks: *Administration*, *Analytics-as-a-Service*, *Block Storage* and *Object Storage*.

[5]This abstraction only applies to the storage building blocks, not for the compute building block of IOStack.

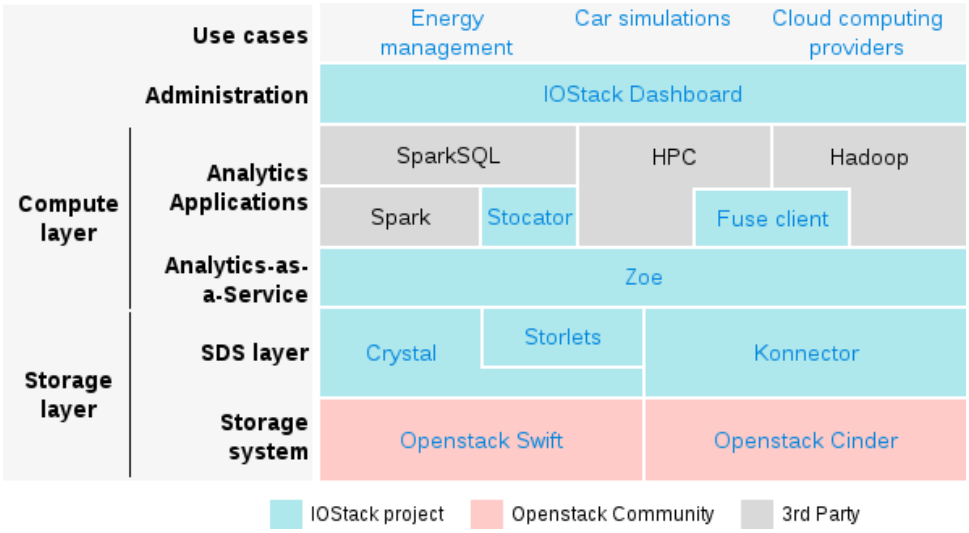


Figure 9: IOStack software stack.

Administration. The IOStack web dashboard is the single point of access to all IOStack services. It integrates the other building blocks in a simple and functional web interface, allowing an administrator to manage the underlying services more easily via policies. The dashboard also integrates the monitoring services that present useful performance and activity information that can be used to take administrative decisions based on workload characteristics.

Analytics-as-a-Service (Zoe). Zoe is the compute component of IOStack: It provides a simple way to provision data analytics clusters and workflows using Docker containers. With Zoe, administrators can use the IOStack dashboard for launching complex data analytics frameworks in a few clicks, or use the APIs to call Zoe from your own scripts. Zoe is independent from applications. A generic application description language is used to build compositions of analytics services, define resource constraints and configuration options. For example, a user can run Spark or MPI jobs on Zoe, by providing appropriate descriptions and Docker images. Moreover, Zoe exhibits high performance: it can create a fully configured Spark cluster, with 20 compute nodes and an iPython notebook in a few seconds. Zoe is built from the start to make full use of the available capacity in your Docker Swarm cluster. Not only Zoe is smart in placing containers, but when resources are exhausted, Zoe will queue new requests using state of the art scheduling algorithms.

Block Storage (Konnector). In IOStack, Konnector is the SDS framework for block storage (OpenStack Cinder). Konnector is instantiated on the "client side" and it provides to the consumer node applications a virtual storage device (VSD). The key feature of Konnector is to implement the filter abstraction at the block-level: A Konnector instance intercepts IOs from the client VM to the storage array and it can add arbitrary computations on the read/write path of these IOs, such as compression and encryption filters. From a design perspective, multiple Konnector VSD devices are managed by the SDS controller. The SDS controller itself is managed through a higher level programming interface and policies. The management of the virtual block storage controller is through the SDS controller, the role of which is to instantiate and control the virtual devices and their filters. The SDS controller keeps all the metadata of the virtual device, allowing the virtual device to be instantiated anywhere in the datacenter without regard to physical storage devices.

Object Storage (Crystal). Crystal is the first SDS architecture for object storage (OpenStack Swift) to efficiently support multi-tenancy and heterogeneous applications with evolving requirements. Crystal adds a filtering abstraction at the data plane (e.g., Storlet filters) and exposes it to the control plane to enable high-level, yet powerful, policies at the tenant, container and object granularities. Crystal translates these policies into a set of distributed controllers. As a result, Crystal offers a great

deal of flexibility to dynamically adapt the system to the needs of specific applications, tenants and workloads.

Apart from the flagship building blocks, IOStack also contributed other components that are very important for the operation of the toolkit. On the one hand, in the context of IOStack we developed **Stocator**, a fast and efficient Spark driver to connect Spark to Swift. As we will see later on, Stocator enables us also to trigger custom computations on Swift from Spark, which can be exploited to accelerate analytics. On the other hand, in the IOStack project we open-sourced **Storlets** as an official Openstack project and we also contributed to it; OpenStack Storlets allows to execute sandboxed computations on Swift storage requests, opening the door to the implementation of a wide variety of filters in object storage. Besides, we contributed a **FUSE Client** as an extension of the S3QL project, in order to control and orchestrate file-system clients accessing to both block volumes and object containers.

It is worth mentioning that despite the fact that the IOStack building blocks are completely integrated and accessible within the administration dashboard, this does not prevent them to be exploited separately based on the necessities of a company. This model is flexible and maximizes the exploitation opportunities of the project's outcomes.

IOStack toolkit in action: Let us illustrate how these pieces work together in relation with the use cases of IOStack. From the administrator perspective, Arctur operators can rapidly deploy analytics applications in containers via the IOStack dashboard with few clicks, as well as monitor their execution, thanks to Zoe.

For example, Arctur administrators can deploy a Spark instance for GridPocket in order to execute SQL queries on IoT data stored in OpenStack Swift, which is being generated by their smart energy meters. With IOStack, connecting Spark to an object store like Swift is much more efficient than before thanks to the Stocator driver. Moreover, Crystal provides a rich layer of SDS services on top of the object store. That is, Arctur administrators can easily add a data compression filter to reduce the storage space consumed by smart meters storing redundant data. Even more, Arctur administrators can deploy active storage filters —implemented as Storlets for security and isolation— that optimize GridPocket SQL queries making the object store able to perform calculations close to the data.

As another usage example, IOStack enables Arctur administrators to deploy other types of applications, such as MPI parallel programs used by Idiada car crash simulations that manage data from block volumes. That is, apart from connecting to object stores, analytics applications virtualized with Zoe can also use block storage as a storage layer, either as physical volumes or by exploiting nested virtualization. In this scenario, the block storage part of IOStack helps Arctur administrators to deploy block volumes with different characteristics, according to a set of policies (e.g., network, storage tier, etc.). Moreover, IOStack also instantiates the concept of filter in the block storage world (OpenStack Cinder) thanks to Konnector: An administrator can mount a volume for Idiada analytics with a pipeline of storage filters, such as caching or compression. This innovation allows cost reduction and performance improvements for analytics running in block volumes.

5.3 IOStack Release Comments

In the last year of the project, most development efforts have been devoted to publish a final release of each building block paying attention to software development best practices: documentation, test coverage, continuous integration, etc., as having a complete final release for each component is essential for their subsequent exploitation.

The source code of all building blocks is available on GitHub. Table 3 shows the Source Lines of Code (SLOC) of each component.

We have used web tools like Travis CI[6], Coveralls[7] or Landscape[8] (see Fig. 10) to help in the continuous integration of IOStack components. These tools run tests and analyze source code quality

[6] `https://travis-ci.org/`
[7] `https://coveralls.io/`
[8] `https://landscape.io/`

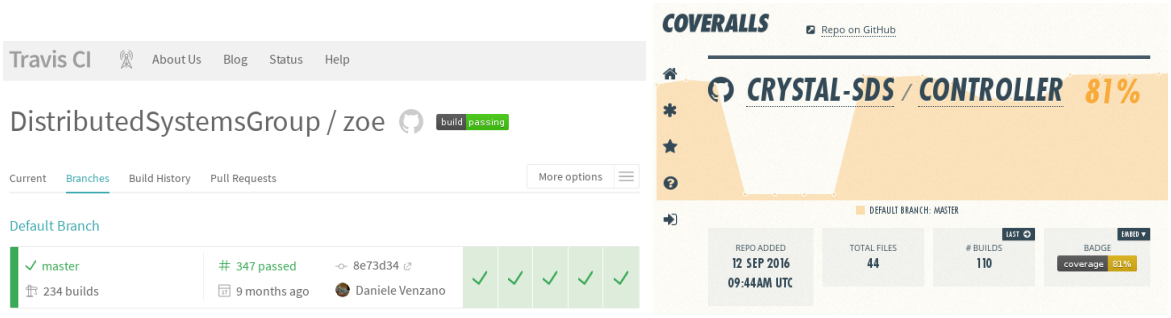


Figure 10: Continuous integration and test coverage tools.

Table 3: Source Lines of Code of each IOStack components

Building Block	**SLOC**
Zoe	14,129
Stocator	7,821
Fuse client	960
Konnector, SDS block APIs, filters	3,718
Block filters designed with Konnector	6,540
Storlets	1,820
Crystal	18,170
TOTAL	**34,988**

and coverage whenever new commits are pushed to IOStack GitHub repositories. Moreover, for internal builds and deployments we used tools like Jenkins[9] and SonarQube[10].

In the next sections we describe each IOStack building block in detail. In the beginning of each section, a table indicates where we can find the source code of each component, as well as its documentation, continuous integration platform, web page, or mailing list.

In addition to software releases, we have also made publicly available some data sets contributed by IOStack partners. These data sets have been widely used in the experiments to evaluate and validate the project contributions, and are available for downloading at IOStack web page:

- *Arctur trace*: 2.97TB of a read-dominated (99.97% read bytes) Web workload consisting of requests related to 228,000 small data objects (mean object size is 0.28MB) from several Web pages hosted at Arctur datacenter for 1 month.
- *Idiada trace*: 1.28TB of a write-dominated (79.99% write bytes) document database workload storing 817,000 car testing/standardization files (mean object size is 0.91MB) for 2.6 years at Idiada.

The consortium has also published as open source two data generation tools that have been used to provide synthetic data for experiments:

- *SDGen* [44]: A synthetic data generator for storage benchmarks. SDGen goal is to enable users creating methods to generate realistic data to feed storage benchmarking tools. It is designed to capture characteristics of data that can affect the outcome of applying data reduction techniques on it.

[9] `https://jenkins.io/`
[10] `https://www.sonarqube.org/`

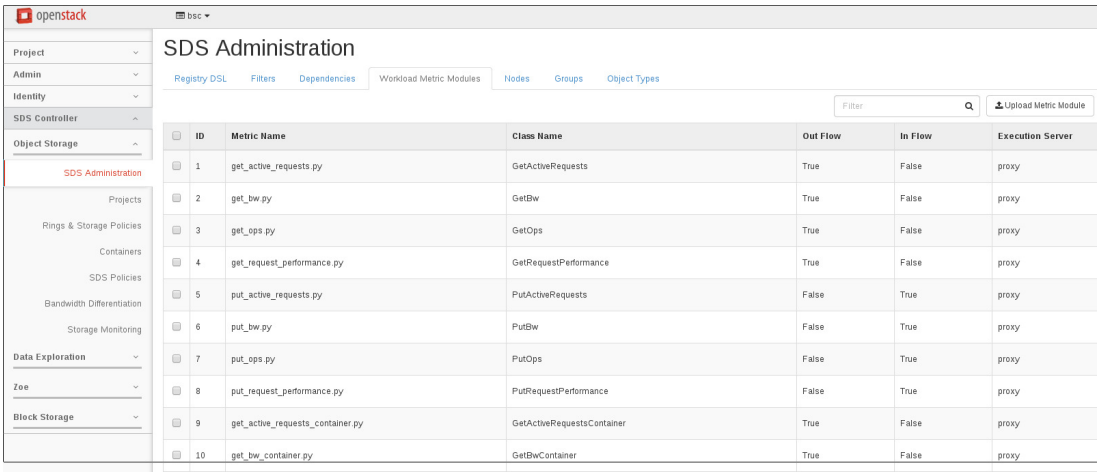


Figure 11: SDS Administration in the IOStack Dashboard.

- *Meter_gen* [45]: An application developed by Gridpocket that generates synthetic datasets similar to real smart meters logs.

6 Integrated Administration Dashboard and Monitoring

6.1 Administration Dashboard

The various building blocks that make up the IOStack toolkit are accessible through their respective APIs and can be independently integrated in third-party projects. In fact, this is a fundamental requirement in order to encourage the use of project results.

However, presenting all the building blocks together in an integrated toolkit offers the opportunity to visualize the relationships and interactions of the different components more clearly. To do so, we have extended OpenStack Horizon, the canonical implementation of OpenStack's Dashboard. Horizon provides a web based user interface to various OpenStack services including Nova, Swift, etc. Our dashboard aims at providing a simple and functional interface to the toolkit, in order to ease its adoption by the enterprise community (see Fig. 11).

During the first part of the project, preliminary versions of all building blocks were integrated in the web dashboard and all of them were deployed and working in the Arctur testbed. During the last months, these building blocks have been improved and updated with their final versions. Moreover, the Dashboard itself has been refactored as an Horizon plugin, following OpenStack recommendations. In this way, IOStack dashboard is guaranteed to work in future releases of Horizon, and is easier to integrate in existing OpenStack environments of potential users of our toolkit. The Dashboard has also been improved with new features like Swift cluster management that allows the user to configure storage policies from the user interface. The information displayed about the cluster has also been improved with devices' status, and node region and zone.

IOStack Dashboard adds a new *SDS Controller* menu to Horizon with four sections: Object Storage, Block Storage, Zoe and Data Exploration.

Object Storage section allows to invoke Crystal API actions from a web interface. It greatly simplifies the management of filters, metrics and policies. The administrator can upload new filters and metrics and define policies (using the Crystal DSL or a web form). The administration panel also offers more advanced features like creating groups of tenants or object types that maximize the specificity of the defined policies.

One of the main benefits of the object storage administration panel is that it helps figure at a glance which metrics are enabled or which policies are currently active. Information about the current status of the object storage cluster nodes is also provided at runtime, offering an option to restart them if they stop working.

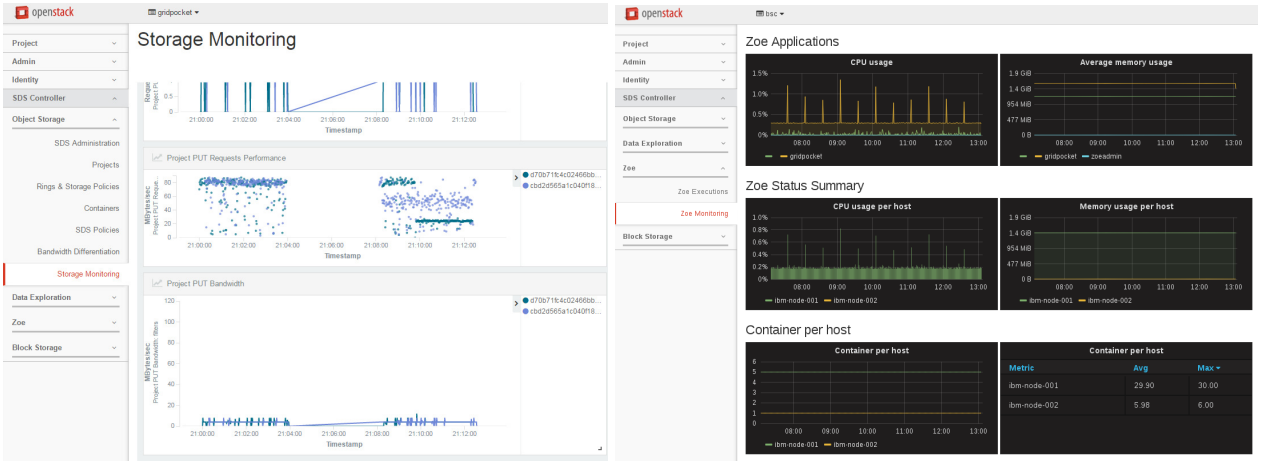


Figure 12: Object storage Monitoring in the IOStack dashboard (left). Zoe Monitoring in the IOStack Dashboard (right).

This section also offers a pre-defined set of graphics that show real-time monitoring information (Fig. 12, left). This *Storage Monitoring* panel shows two different kinds of monitoring data: system resources usage and object storage activity. System resources data is obtained with collectd[11], a small daemon that runs on each host to be monitored, collects statistics about the system and provide mechanisms to forward the samples via the network to be centrally aggregated. Object storage monitoring information is obtained with our own metrics middleware that intercepts Swift requests and performs real-time measurements like the number of `GET` operations per second of a tenant (see Section 12.5. As we will see later on, all these monitoring information is sent to Logstash, stored in Elasticsearch and visualized with Kibana.

Block Storage section allows an administrator to define storage policies and storage groups, and to create SDS volumes based on these storage groups. Storage policies support filter pipelining, i.e. define a set of filters that will be executed one after the other for the same data blocks. Once a particular policy is defined, the administrator can create a storage group selecting the policy and the storage nodes that will form the group. Then, in the SDS volumes tab, the administrator can create an SDS Volume for this group. Once this volume is attached to an instance, read/writes to this volume will have the previously defined policy applied to them.

The block storage section also offers a monitoring panel, that with the help of additional code based on Elasticsearch and Kibana, helps to monitor performance of individual volumes and of underlying RAIDs and disks in the storage nodes.

Zoe integration allows an administrator to create application executions by configuring the number of workers and memory limits. The Zoe executions panel offers details of each execution like the scheduled time, the status, and information about each service endpoint URL.

Monitoring is also integrated in this section (Fig. 12, right), showing graphical information of CPU and memory usage at the application and host level, as well as the number of containers per host. This monitoring solution was built combining 3 open source projects and custom tools developed to cater to IOStack specific needs; as it was explained in Deliverable D5.1, collectd was chosen for metrics gathering, Carbon/Graphite for storage, and Grafana for metrics visualization.

Finally, in **Data exploration** section, we use Kibana as an off-line monitoring tool (*data exploration*) that allows an administrator to explore data and create new interactive plots and dashboards that may be useful to take decisions based on workload characteristics. Kibana is an open source data visualization plugin for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Users can create bar, line and scatter plots, or pie charts on

[11] `https://collectd.org`

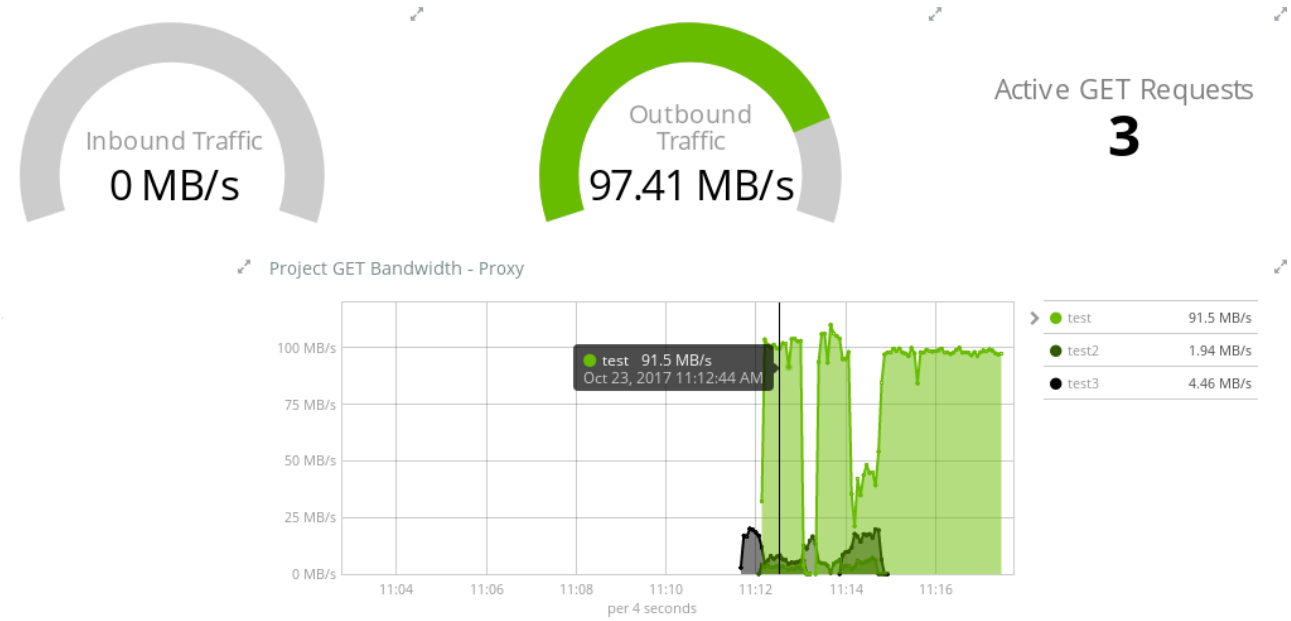


Figure 13: In data exploration section, an administrator can create interactive plots and data dashboards.

top of large volumes of data. These plots can be combined to create custom dashboards that help administrators get an overview of the system (Fig. 13).

6.2 Monitoring

Summarizing what we have outlined in the previous section, we see that monitoring is a key part of IOStack toolkit. Monitoring data is used to:

- Define dynamic policies that react on workload changes and to have a control feedback to implement these policies.
- Have real-time monitoring in a web dashboard.
- Have off-line monitoring tool that allows to explore the data and helps administrators decide which policies can be applied to improve system performance.

In Fig. 14 we can see the monitoring data flow from the different components that collect or generate data, to the components that store and consume them. CollectD is used by the different building blocks to collect monitoring information from storage or compute systems. IOStack control plane captures monitoring information with RabbitMQ, a Message Oriented Middleware (MOM) broker that provides high-performance event processing service.

To provide a clear organization of monitoring events, we instantiate a queue per input metric type. That is, all the events related to the IO transfers of storage nodes will be inserted into one queue, whereas events related to the storage capacity of storage nodes will belong to another queue. By doing this, we ease the consumption of monitoring events from the viewpoint of workload metric processes.

By default, CollectD provides interesting information about the physical usage of the storage system, including the IO capacity or the current CPU state of a storage node, to name a few. Apart from information about the physical resource utilization, we generate monitoring information concerning the OpenStack service at hand (e.g., Swift). That is, we are capable of extracting workload metrics related to the logical viewpoint of the service, such as the read/write throughput (i.e., MBps) of a

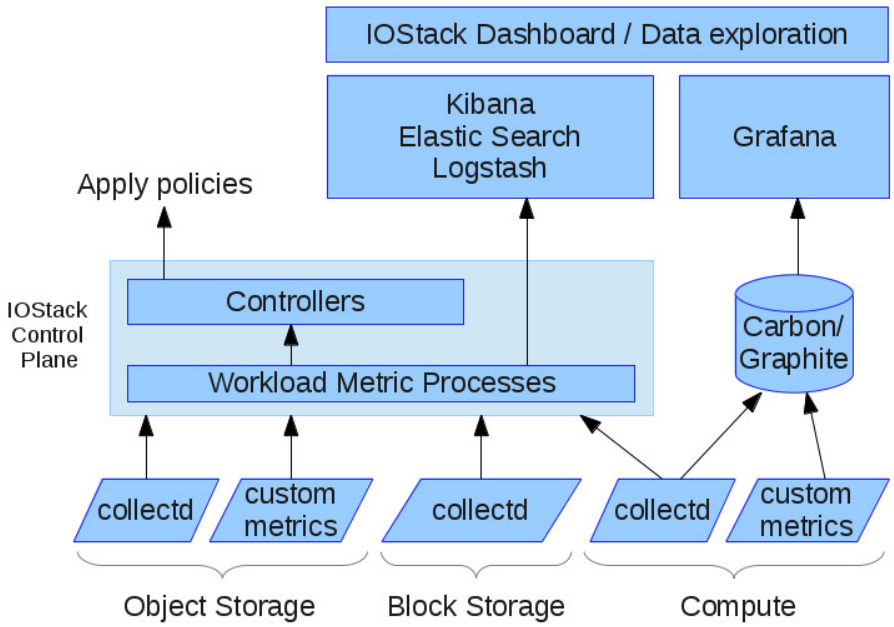


Figure 14: High-level overview of IOStack monitoring layer. As can be observed, the IOStack control plane serves as a centralized recipient for most monitoring information sources. Such monitoring information is then consumed by control algorithms that may dynamically change the behavior of building blocks.

tenant within a time interval, or the number of GET/PUT requests performed by a tenant. This approach represents a more accurate notion of the workload supported by the service and enables us to track and analyze the activity of tenants and enforce the appropriate filters on them.

Moreover, we can generate and inject arbitrary types of monitoring events to the system, for their further exploitation by different types of dynamic policies. For instance, we could monitor the *compressibility* of data objects that are transferred to/from the system. Such kind of policies can be achieved in IOStack by introducing processes that generate the desired metrics and send the values to RabbitMQ. We already demonstrated this feature by providing one of such custom metrics regarding the IO bandwidth exhibited by tenants and containers.

IOStack monitoring provides a high degree of precision related to the storage monitoring, defining monitoring events at the tenant and container/volume granularity. This high degree of precision allows workload metric processes to enable triggering policies also at the tenant and container/volume granularity, because there is a degree of dependency between the resolution of monitoring information and the definition of dynamic storage policies.

As depicted in Fig. 14, workload metric processes also send monitoring information to an Elastic stack (Logstash, Elasticsearch and Kibana). Logstash is a dynamic data collection pipeline that is able to ingest data from a multitude of sources simultaneously, transform it, and then route it to a variety of outputs (in Elastic stack, data is sent to Elasticsearch). Elasticsearch stores all the monitoring data and acts as the search and analytics engine. On top of the content indexed by Elasticsearch, Kibana provides dynamic visualization capabilities. As said in the previous section, Kibana is used both for real-time and off-line monitoring.

Finally, it is worth mentioning that control plane gets data from all building blocks. This make it possible to consider the application of "cross-layer" optimizations, for example between compute and object storage building blocks (see Section 14).

7 Zoe Analytics

Web page	http://zoe-analytics.eu/
Source Code	https://gitlab.eurecom.fr/zoe/main
Documentation	http://docs.zoe-analytics.eu/
Continuous Integration	https://gitlab.eurecom.fr/zoe/main/
Issue tracker	https://github.com/DistributedSystemsGroup/zoe/issues
Mailing List	http://www.freelists.org/list/zoe

Zoe was created in August 2015 as an open source project [46] to satisfy the need of having an easy and integrated way to deploy distributed analytic applications on a cluster of physical or virtual machines. Users can define analytic applications starting from a number of ready-made building blocks, Zoe will schedule and deploy them matching resources requests and availability.

Zoe is developed in Python and is conceived as a thin layer that builds on top of an existing low-level cluster management system, which is used as a back-end to provision resources to applications. Raising the level of abstraction to manipulate analytic applications is beneficial for users and ultimately to the system design itself: application scheduling decisions can be taken with a small amount of state information, and do not happen at the same (extremely fast) pace at which low-level task scheduling does.

Next we overview Zoe's software design and implementation. In the final part of this section we will describe the role of Zoe in IOStack. Deliverables 5.2 and 5.3 contains more in-depth information about Zoe, including the road-map and community engagement information.

7.1 Zoe applications

Zoe schedules applications. Each application is made of one or more components, that run each in its own Linux container. For example, the Spark Notebook application that a user submits to Zoe is made of one Jupyter Notebook[47] component, one Apache Spark[48] master and one or more Apache Spark workers.

In order to produce useful work, in this case for the application to be useful to the user, there is a core set of components that can be identified: the notebook, the master and just one worker. The application can mark additional workers as "optional" (elastic in Zoe's terminology). Zoe will start them only if there are free and unused resources.

To simplify application descriptions and build a library of building blocks, an intermediate concept of frameworks (groups of components that work together) has been introduced.

Zoe applications (ZApps) are described by users via a simple JSON description that follows a high-level configuration language (CL) to specify applications, frameworks and components with their classes (core or elastic), resource reservations and constraints. The CL is simple and extensible: it aims at conciseness and, with framework templates, can be also used by "casual", in addition to "power" users [49]. An example of the simplicity and effectiveness of the Zoe CL, building a batch application for the distributed version of Tensorflow[50], only required less than 25 lines of CL.

A typical Zoe application description contains a number of application-global metadata items, such as size information for the size-based scheduler. Then it contains a list of the components, each with its own metadata, that includes:

- image: the Docker image location for the component
- environment: the environment variables that are used to configure the component
- volumes: external storage to mount inside the container
- resources: the resource reservations required to run one instance of this component
- total count: the total number of instances of this component that can be started

- essential count: the minimum number of instances (out of total count) required for the application to produce useful work

The last Zoe release (2017.09) includes a ZApp shop, a web-based tool where users are free to compose and configure their own applications in a more intuitive way. A number of ZApp packages were made available via GIT and in the source code Zoe distribution, so that Zoe deployers can start with a rich selection of pre-built ZApps.

7.2 Internal architecture

Zoe is divided into two multi-threaded processes. The Zoe Master and the Zoe API. Both store state information into an external Postgres database, that has well-known reliability and fault tolerance characteristics.

Users interact with the Zoe API process. It offers the web interface and the REST API and does an initial validation of user input and application descriptions.

The Master process contains the scheduler and other threads to process events generated by the back-end, manage asynchronous application termination and respond to requests coming from Zoe API. The two processes use a Zero-MQ based protocol, that has been developed taking in all the recommendations to build a robust protocol in the face of crashes or disconnections.

The Master and the API processes do not store any state internally and can be restarted at will in case of upgrades or crashes without any consequence. The API process can be scaled horizontally behind an http-based load balancer.

7.2.1 State

Three main tables are maintained in the state store, the platform table, the executions table and the service table. In this context, executions are instances of Zoe applications, while services are instances of components.

The platform table stores information about the number of containers and the amount of free resources for each node available in the cluster. This information is taken as-is from the back-end API and stored into the database for caching reasons. Each entry describes the resources (memory, cores and containers) free and total for one single node. Data is updated whenever the scheduler is triggered.

Each entry in the execution table refers to a single application execution submitted by the user. It records information such as:

- identifier of the user who submitted the application
- timestamps of submission, start and termination events
- current status (submitted, running, error, etc.)
- error message in case the execution failed due to an error

Each entry in the services table refers to a single instance of a component. As was explained in the previous sections, application descriptions contain a list of component with total and essential counts for each of them. These descriptions are exploded into the state table in as many services as is the total number of instances requested. The information recorded for each instance is:

- status: the service status from Zoe point of view
- backend status: the container status from the back-end point of view
- backend identifier: the unique ID the back-end uses to identify this container
- service group: the component name as given into the application description
- error message: filled in when there is an error during the container lifetime (for example image not found or out-of-memory termination)

7.2.2 Scheduling and placement

The scheduler thread is triggered by several events:

- an execution being added to the scheduling queue
- an execution that terminates, either by itself (batch) or by the user
- a timer, to account for resources that are used outside of Zoe's control.

When the scheduler is triggered and selects an application execution to start, according to the configured policy, it performs a placement simulation, trying to find the best fit of core and elastic services in order to maximize the number of running executions. Executions for which all core services cannot be started at the same time are left in the queue, in order to start only executions that can produce useful work.

The placement simulation uses a simple filter-and-sort algorithm. For each container to be placed it filters out nodes that do not match the required resource constraints (amount of memory, but also special hardware needs can be taken into account). The container will be placed in the node with the least amount of running containers and the biggest amount of free memory.

This procedure is repeated for each service until either:

- all core services are placed: then the execution is started, following the simulated plan
- a core service cannot be placed: the execution is re-added to the queue and the simulation results are thrown away

The same filter-and-sort algorithm is used to take placement decisions in most modern cluster manager systems, like OpenStack Nova and Docker Swarm.

7.3 Back-ends

The main design idea of Zoe is to hide the complexities of low-level resource provisioning from application scheduling and use an existing cluster management system, for which many alternatives exists, instead. Currently, Zoe supports three back-ends: Docker Engine, legacy Docker Swarm and Kubernetes. Zoe builds on top of these back-ends and uses them for orchestration, dependency management, resource isolation, naming and networking.

7.4 User interaction

Users can interact with Zoe through a web or a command-line tool. The web interface provides an high level overview of the application executions for each user and their status. The command-line tool is more advanced and, for example, can be used to script the execution of multiple batch applications, creating simple automated workflows. Zoe has also a REST client API that can be used to develop more tools and services.

When an application is submitted, via REST, command-line or web interface, Zoe creates an entry in the application state store, and adds it to a pending queue. Our system allows plugging several scheduling policies to manage the pending queue, ranging from simple to sophisticated size-based strategies. The scheduler strives at making sure the application selected for execution can make progress as soon as resources are allocated to it: to this end, it relies on the back-end to place all core components according to the simulated plan. Elastic components are scheduled when possible and contribute to decreased application run-time. As a consequence, batch applications (either rigid like Tensorflow or MPI, or elastic such as Spark) can make progress as soon as core components start; similarly, interactive applications – which can be given precedence to reduce queuing times and improve user experience – can also start being used even if not all elastic components (if any) are scheduled.

In the last Zoe release (2017.09), the web interface was rebuilt to have a better usability and provide useful information at a glance.

7.5 Zoe and IOStack

IOStack promotes the separation of compute and storage layers allowing increased flexibility in meeting the variable demand of computation resources, while keeping the data storage system in a stable state. Zoe targets the compute layer of IOStack, by providing a simple way to define and deploy arbitrary analytic applications.

Zoe has been integrated into the IOStack dashboard to give users a single point of access to all IOStack services. Zoe applications can use the storage services offered by the other IOStack components directly (Swift) or through the back-end (volumes).

During the last year of the project, the ZApp description of Zoe has been augmented with new fields that help the Software-Defined-Storage layer (Crystal) in identifying the correct policy to apply at the storage that will be involved.

8 Stocator: A Fast Spark Connector for Object Stores

Web page	`https://spark-packages.org/package/SparkTC/stocator`
Source Code	`https://github.com/SparkTC/stocator`
Documentation	`https://github.com/SparkTC/stocator/blob/master/README.md`
Continuous Integration	`https://travis-ci.org/SparkTC/stocator/`

Apache Spark can access multiple data sources that include object stores like Amazon S3, OpenStack Swift, IBM SoftLayer, and more. To access an object store, Apache Spark uses Hadoop modules that contain drivers to the various object stores.

Apache Spark needs only a small set of the object store functionalities. Specifically, Apache Spark requires the following operations: listing the containers, listing the objects of a given container creation, object read, and getting data partitions. Hadoop drivers, however, must be compliant with the Hadoop eco system. This means they support many more operations, such as shell operations on directories, including move, copy, rename, etc. which are not native object store operations. Moreover, Hadoop Map Reduce Client is designed to work with file systems and not with object stores. The temporary files and folders it uses for every write operation are renamed, copied, and deleted. All this leads to dozens of useless requests targeted at the object store. It's clear that Hadoop is designed to work with file systems and not object stores.

Stocator, although implementing the Hadoop operations, is implicitly designed for the object stores, and it has a very different architecture from the existing Hadoop driver. It does not depend on the Hadoop modules and interacts directly with object stores.

Stocator is a generic connector, that may contain various implementations for object stores. It was initially provided with complete Swift driver, based on the JOSS package[12], however it can be very easily extended to more object store implementations.

The Stocator connector has been in production use in IBM Cloud [51] from the start of 2017 for connecting the Apache Spark service to the IBM Cloud Object Storage [38]. A full description and evaluation of Stocator can be found in deliverable D4.3.

9 FUSE Client: Taking Control of File-system Object Storage Clients

Source Code	`https://github.com/iostackproject/s3ql`

Idiada workflow uses block storage instead of object storage. Changing the workflow to use object store includes too many changes and an adaptation period. For this reason we searched for a solution to be able to use Swift as the backend for Idiada (and be able to use all the Crystal filters including storlets) but maintaining the POSIX block storage interface. One of the solutions was to build a FUSE layer intercepting and translating calls. Although building the FUSE layer may introduce a better interface for filters (as we can control directly all the layers), building a FUSE filesystem with enough maturity to become usable by Idiada in their day workflow is a large task.

[12] `http://joss.javaswift.org`

For this reason, we looked for similar and mature projects that could be integrated into the workflow so we will be able to center our efforts on the filtering part. We found S3QL project [13] that includes support for different object storage backends and the capabilities of compression, deduplication and encryption in the client side.

Those capabilities were the ones that Idiada need and the ones we wanted to develop. However, in order to be able to integrate the solution inside the Crystal framework we decided to modify S3QL with the next extra capabilities:

1. Ability to add filters on the I/O flow.
2. Ability to control the configuration using REDIS.
3. Ability to upload the filter code using REDIS.
4. Ability to enable or disable the native compression using REDIS.

The usage of Redis as the control point enables the utilization of the management interface of Crystal to manage the client filters in S3QL. This produces a end-to-end control so we can dynamically enable, for example, the compression on the client (reducing CPU resources on the storage node) or move the compression to a storlet (reducing CPU resources on the client node, but increasing the network traffic).

9.1 Implementation

On this section we will explain the different implementations tried, and how S3QL is configured to run the created filters. We will explain also the API to create new filters by the administrator or the user. The code can be found in `https://github.com/iostackproject/s3ql`.

Filters on the I/O flow. We tried two implementations to give the user the chance to use filters on the client-side. Given that S3QL is a very mature project, our first approach was to modify only a small part of the software. We implemented the filter with a similar approach than the BlockStorage filters of Konnector (Section 10). The insert point was just before the data gets and leaves the buffering system of S3QL. This implementation had several problems.

The main problem was that the size of the object cannot be modified, so compression filters can not be used. This is the same problem that we had in block storage filters. Secondary issues that appeared were the inability of identify the object or the file. On the other hand, cache filters can be applied but as they are already implemented in S3QL did not show any benefit.

The final implementation mimics the compression filter implementation inside S3QL. We implemented a dynamic filter stack divided into *In->Out* and *In<-Out* python filters that can be inserted into the normal *filter* stack of S3QL. As it is a filter stack we can define a chain of filters that will be executed following a defined order in both directions.

Configuration control. To control the configuration of the filter stack we need a initial filters.ini file, containing ip, port and REDIS key to the configuration contents. i.e.

```
[General]
Param : 127.0.0.1 6379 client_filter_configuration
```

The configuration is read once each time we open a file in the S3QL side, this allows dynamic changes to the filter stack. client_filter_configuration key in REDIS should contain something like:

```
"Filterqsub:127.0.0.1 6379 client_filter_qsub_extension client_filter_qsub_process;
Filtertag:127.0.0.1 6379 client_filter_tag_list;
Filter1:127.0.0.1 6379 client_filter_compression;
Filter2:22;
ORDER:Filtertag,Filter2 /tmp/"
```

[13] `https://bitbucket.org/nikratio/s3ql/`

The format is `NameOfTheFilter:parameters`. Each Filter is separated by a semi-colon. There is a special key `ORDER` that configures the stack and the path where the filters will be downloaded: `ORDER: Filter1,Filter2,Filter3 <path>`. The path is a temporal path, because we need to load the filters as python modules dynamically.

Finally we have for each filter the code in Redis. You can find sample filters in the `src/s3ql/` path. In order to upload to Redis do this:

```
redis-cli -x SET client_filter_Filtertag_code < src/s3ql/Filtertag.py
```

The filter code is loaded dynamically from Redis.

Controlling native compression filters. Using the Redis key client_native_compression with values : zlib, lzma, bzip2 or none. The compression can be switched on-off by a file basis. If the key is not set, the compression uses the default parameters (lzma)

As we explained before, the configuration is checked every time a file is opened.

9.2 Included filters

We have developed a filter as example, like the gzipFilter, and another set prepared for Idiada use case.

gzipFilter is a complex filter example, as it makes use of the capabilities to read or write more data through the filter stack. As for example, if the compression filter needs 1 more byte to finish the compression, it should be asked to lower layers. However the size of the recovered data may have more than 1 byte so the surplus should be stored for later use.

The compression uses gzip command line to compress each of the blocks. This filter includes the capability to be activated or deactivated in each block. As this is a configuration of the filter, the check of the content of the key is decided by the filter. `client_filter_compression` may be `on` or `off` to enable or disable compression.

filterqsub This is a filter for IDIADA use case, with this filter the client can define a set of extensions that will be monitored when a file is closed. If there is a hit, the process specified in the configuration will be run using as first parameter the name of the file.

In this filter, we include the capabilities to know the name of the file, this is something that breaks the separation of the s3ql code as we shouldn't be able of knowing the name of the file from the object. One problem is that as we do not know when the file is closed, the process is called every time an object is closed (a file contains several objects that are uploaded to swift). Although this is not a big issue, as the file will be finished in a few seconds, is something that we need to take into consideration.

As an example, the configuration of the file is the next:

```
client_filter_qsub_extension "txt c3d" client_filter_qsub_process "/tmp/process.sh"
```

filtertag The last filter is also an Idiada filter. The filter looks for the file name and tries to get a submatch. If we have a hit, we add the specified tag to the object. The tag can be checked from the server side, using a Storlet for example. It can also be checked on the client side, we have an example implemented that changes the content based on the tag. For example, returning a 0's file if the tag is SECRET.

As an example, the configuration of the file is the next:

```
client_filter_tag_list "file1.txt#SECRET file2#MORESECRET"
```

Tags will be stored as Meta-Tag<index> in the metadata of each object (we look for partial matches, so a file can have multiple tags).

9.3 Implementing filters

Filters only need four functions that need to be implemented.

Init will be called at the start of each file, **flush** will be called on every close. However, the **close** will be called on each object close of the file (a file is divided in 1 MByte objects). **readxform_c** and **writexform_c** will be called for a part of the data.

```
def init (param):
return None

def readxform_c (self,buf,param):
arr = bytearray(buf)
buf = bytes(arr)
return buf

def writexform_c (self,buf,param):
arr = bytearray(buf)
buf = bytes(arr)
return buf

def flush(self,param):
buf = None
return buf
```

We have also some self variables that can be used to store data from different calls (each file creates a new filter object)

For example, self.db provides access to the database to gather the filename. Many examples can be found on the provided filters.

If more variables need to be used, the developer can modify `s3ql/comprenc.py` **InStdOutFilter** or **InFilterOutStd** and include it.

9.4 Loading filters

The filters are stored inside REDIS, S3QL gets the code and writes it to a tmp directory. Then invalidates the code cache from python and finally imports the module.

```
code = r.get("client_filter_"+name+"_code")
open(path+name+".py", 'w').write(code.decode("UTF-8"))
importlib.invalidate_caches()
filterlist[name] = importlib.import_module(name)
```

10 Konnector: SDS for Block Storage

Source Code	`https://github.com/MPSTOR/Konnector`
API specifications	`https://github.com/MPSTOR/Konnector/blob/master/Konnector-API.adoc`

In many scenarios, the execution of data analytics jobs involve alternative storage substrates, such as *block storage*. In general, compute nodes executing analytics and *storage arrays* are physically disaggregated, which simplifies the management of a datacenter infrastructure. Thus, similarly to the usage of a physical disk or SSD, compute nodes interact with a storage array via standard block-level IO protocols (e.g., iSCSI) that operate through a high speed network link (e.g., 10GBps, fiber channel).

Once provided a block storage communication protocol, compute nodes can mount *logical volumes* that map physical storage space of the storage array for managing data. For instance, in the Idiada use-case, data scientists execute car crash simulations on VMs making use of file systems that

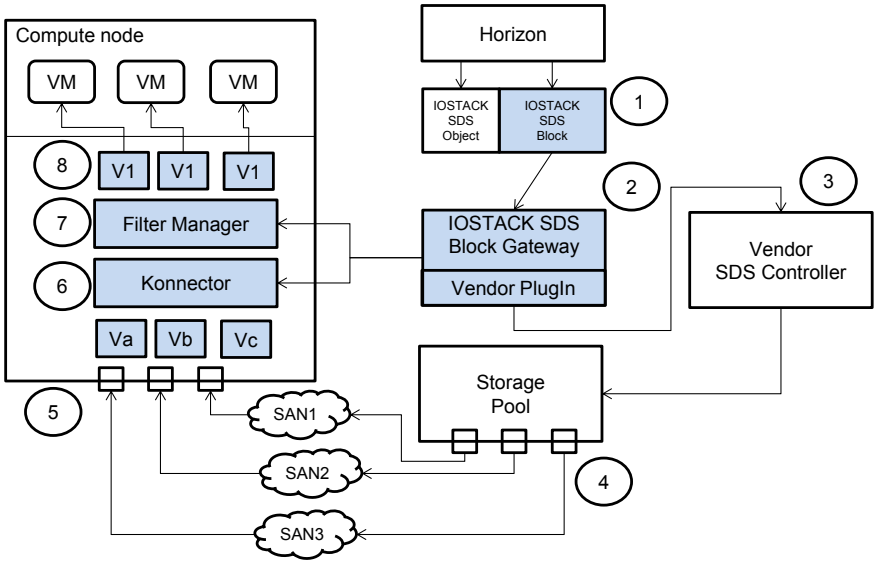


Figure 15: Overview of the SDS Gateway and Konnector block storage components.

mount volumes on large storage arrays. This makes life easier for Idiada's data scientists: they write analytics code that manages data from a large storage cluster, just as they would do in a file system of a single machine on top of a traditional HDD.

Although being a commercial standard, block storage arrays are normally proprietary closed systems, often complex to manage, and they are perceived as expensive. But more importantly, storage arrays being mission critical systems are slow to evolve, which results in an *inflexible innovation platform*. In contrast to block storage arrays, compute nodes have an open software architecture and they can execute an increasingly varied types of analytics. This mismatch is true even in the case that some storage arrays can be delivered with a fixed set of built-in storage features —e.g., de-duplication or replication—, given that implementing a new feature or adapting the existing ones to the particular necessities of Big Data analytics is not practical.

In IOStack, the SDS toolkit provides a framework to *manage block storage arrays* (SDS Gateway) and *extend their functionality* by providing a filter stack on the open compute platform (Konnector). In the following, we briefly overview these components; a full description of the SDS toolkit for block storage can be found in deliverable D3.3.

10.1 SDS Gateway: Advanced Storage Provisioning and Automation

The IOStack SDS Gateway is the tool that enables administrators to manage storage volumes in an automated way. In particular, in an OpenStack environment, it allows an administrator to provision block storage according to a defined policy. Following the principles of SDS, the SDS Gateway allows the user to choose a volume type, this volume type is tagged to the compute group within which is configured with a set of storage arrays, storage tiers, fabrics and consumer node filter functions. When a volume is created within this storage group the SDS Gateway virtualizes for the user all the operations of choosing which storage array, fabric and media tier to use. This greatly simplifies the process of provisioning storage. The second step of attaching a storage array volume and creating on the consumer node a filter stack is also fully automated process. These complex provisioning tasks reduce the cost of provisioning managed storage by removing the need of the user to know anything about the datacenter internals and simply use the services created by the the datacenter administrator. This is specially true considering that all these options are exposed within the IOStack administration dashboard.

Technically, the SDS Gateway provides a REST API for a client to manage storage. In Fig. 15 the client is the OpenStack Nova Cinder controller. The Cinder API commands are translated and

forwarded to the SDS Gateway. The SDS Gateway maintains an object model which is configured by the Horizon Dashboard SDS plugin.

The SDS plugin for the Horizon dashboard creates storage groups, a set of storage nodes are added to each storage group. Associated with each storage group is a policy. The policy configures configures default storage options for all volumes provisioned within that storage group. Some of the options are storage node specific, such as the fabric to export the volume on, other options are specific to the compute node —namely, consumer node— where the storage volume is attached to.

The consumer node specific component is a description of the filter that that is dynamically built once the storage volume has been attached to the consumer node. Once a storage volume has been attached to a consumer node, the SDS Gateway uses the Konnector API to dynamically create a filter stack between the attached volume and the volume presented to the final consumer, for example a VM. This operation is shown in Fig. 15 call-outs 5,6,7,8. This schema allows storage volumes to be created on storage nodes and a stack of dynamic storage functions to be applied to the storage volume independent of the storage node.

As we show next, the Konnector component leverage compute node flexibility by moving storage functions (filters) into the compute node.

10.2 Konnector: Extending the Functionalities of Block Storage

The goal of the filter stack is to provide a flexible platform for innovation and value added functions on top of the storage array volumes. That is, by moving the data flow from the kernel space into the "user space" filters can be dynamically created at run time. Run time creation of the filter stack in mandatory because the volume and its stack are only instantiated when the storage array volume is attached to the consumer node or VM.

As visible in Fig. 15, the filter stack is a layer between the terminated storage volume on the compute node and the consumer of the storage volume, such as a virtual machine. A filter stack can provide functionality such as encryption, compression de-duplication on the data flow between the consumer and the storage array.

In our implementation, the filter stack is dynamically created as each filter is implemented as a `.so` (a dynamic linked library). The Filter manager when requested through the Konnector API will dynamically build the filter stack using a set of `.so` dll library functions. This approach allows the filter stack to be built on demand, it also allows any third party to create filter functions and add an IO processing function into the data flow between the consumer (e.g., Virtual Machine) and the storage array volume. The storage array volume is agnostic to the filter function since the storage array just sees data in/out of the attached storage array volume on the consumer node, it does not see nor is aware of any of the filter transformations.

From a developer perspective, filter objects are built from C source files. The SDS toolkit provides a number of skeleton filter samples which serves to act as a template for filter development. Developers have several API functions to transform IO operations intercepted by connector:

- `write_xform`: This function exists to perform a transformation at its defined filter level on payload write data bound for the device.
- `read_xform`: It is called at the same predefined filter execution level and is designed to act on the read payload data from the device en-route to the initiator.
- `pre_read`: We can redirect reads to another zone of the disk.
- `pre_write`: We can redirect writes to another zone of the disk.
- `get_name`: This function can be called from the controller. This just returns the name of the filter which is defined as a static string in the filter object. This is just added for debug purposes and is called when the filter daemon is executed in debug mode.

- `pass_args`: This function allows the user to pass arguments to the filter function when the filter is instantiated. The arguments can be specified using the syntax of the filter specification in the Horizon dashboard.

Next, we describe some of the advanced filters that we developed making use of these API calls to improve the Big Data processing of our use cases.

10.3 Block Filters Designed with Konnector

Source Code	`https://github.com/bsc-ssrg/BlockStorageFilters-IOSTACK`

In the basic package of block filters, we include simple proof-of-concept ones such as a `xor` filter that transforms each byte of a stream with an "exclusive OR" transformation, or a `noop` filter that simply intercepts the IO flow without actually manipulating it. While useful for basic testing purposes, we still need to exploit the filter framework to develop real-world filter for our use cases.

For this reason, we developed filters that go a step further of the original behavior of the filter framework. The original behavior was a 1:1 block transformation (reading or writing), however, in our scenarios we need to be able also to perform *n*:*n* transformations. For example, data prefetching or caching filters need to check before the actual read occurs whether the content is available or not, then the framework knows if the read should be issued to the storage array. On the other hand, in the case filters that require to modify the IO flow, such as in the case of data compression and deduplication, require to manage their own metadata space transparently to the real block device; this is needed it writes the real data it should know if the data should be written or not. As one can infer, the design of these filters may provide performance and cost reduction benefits to real use cases, but are challenging to materialize efficiently.

In particular, we developed the following filter for our use cases and for advanced evaluation purposes:

1. Prefetch filters (prefetch)
2. Cache filters (dedupcache and compress)
3. Output modification filters (OCompress)
4. Evaluation filters (mockup)

Prefetch filter. This filter preloads data in advance to reduce network latency. `prefetch` filter is divided into two filters, the first one logs the offset and sizes of the data that is being used in the VM. The second filter preloads the data in advance. We are also going to develop a new filter on the next period that will enable just-in-time prefetching so the blocks are only preloaded just when they are going to be used.

Cache filter. This filter stores any block read into its memory space. Its objective is to increase the performance on workloads that are being read twice or more times. The cache is reduced using deduplication (`dedup`) and compression (`compress`), so we can store more data with less memory usage and surpass the buffer cache space.

Output modification filters. These filters generate a different output from the original one, such as compression or encryption. The main difference is that the filter is persistent, so the output can only be read if the same filter is used. `ocompress` generates a compressed file system using two compressors (for Idiada use case). The filter is transparent for the user. However, further modifications in the filter framework are needed to allow to export a, i.e., 6GB real volume, and present it inside the VM as a 10 GB volume. The main issue is that the increment of space should be static and in advance (using some % gathered or introduced by the user) and can not be changed as it will confuse the VM operating system.

Evaluation filter. This filter tries to expose several "turning knobs" or parameters to evaluate the framework; for instance, we can emulate latency delays or CPU usage delays in the filter workflow. `mockup` filter has those parameters, so delays are introduced on each read or write request.

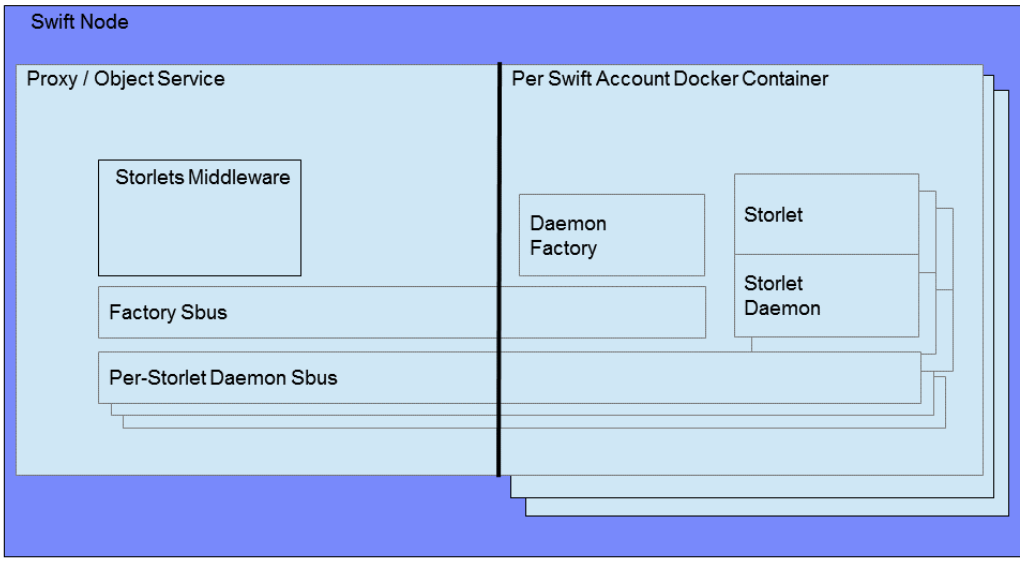


Figure 16: Storlets architecture

11 The Storlets Framework

Web page	`https://wiki.openstack.org/wiki/Storlets`
Source Code	`https://github.com/openstack/storlets`
Documentation	`https://docs.openstack.org/storlets/latest/`
Tests	Unit tests and functional tests
Continuous Integration	Gerrit Code Review + Jenkins

In the IOStack project we open-sourced and contributed to the **OpenStack Storlets** project, that was used to implement a wide variety of filters for object storage.

Storlets is a framework to intercept and execute sandboxed code on object requests in OpenStack Swift. Storlets provide a powerful extension mechanism to OpenStack Swift —without changing its code— to run computations close to the data in a secure and isolated manner making use of Docker. With Storlets a developer can write code, package and deploy it as a regular object, and then explicitly invoke it on data objects as if the code was part of the Swift's WSGI pipeline. Request interception can occur not only at the proxy but also at the object servers thanks to the Storlet's WSGI middleware integrated in Swift, which "wraps" storage requests and responses.

At the high level the storlet engine is made of the components described below:

11.1 The storlet middleware

The storlet middleware is a Swift WSGI middleware that intercepts any given storlet invocation requests and reroutes the stream of data through the Docker container in which the specified storlet is executed. The storlet middleware is deployed both in the proxy-server and the object-server pipelines.

The storlet middleware is written in a way that allows to extend the engine to support sandboxing technologies other than Docker. All what is required from given sandbox is to implement the "storlet gateway" API which defines the functionality to run storlets.

11.2 Swift accounts

The storlet engine is tightly coupled with accounts in Swift in the following manners:

1. In order to invoke a storlet on a data object residing in some Swift account, that account must be enabled for storlets. That is, the designated, user defined, metadata flag on the account must be set to true.

2. Each Swift account must have certain containers required by the engine. One of these containers is the "storlet" container, in which storlets are being uploaded. After a given storlet has been uploaded from a given account to to the "storlet" container, it can be invoked on any data object pertaining to that account, given that the invoking user has read permissions to the "storlet" container.

3. Each account has a separate Docker image (and container) where storlets are being executed. All the storlets that are executed on data objects belonging to some account, will be executed in the same Docker container. This permits differentiating images as function of the Swift accounts. The Docker image name must be the account id to which it belongs.

11.3 The Docker image

As mentioned above there is a Docker image per account that is enabled for storlets. At a high level this image contains:

1. A Java run time environment. This is needed when you run storlets written in Java.

2. A daemon factory. A Python process that starts as part of the Docker container bring up. This process spawns the "per storlet daemons" upon a request from the "storlet docker gateway" that runs in the context of the storlet_middleware.

3. A storlet daemon. The storlet daemon is a generic daemon that once spawned, loads a certain storlet code and awaits invocations. Different storlets, e.g. a filtering storlet and a compression storlet are loaded into different daemons. A daemon is invoked the first time a certain storlet needs to be executed.

4. The storlet common jar. This is the jar used for developing storlets in Java. Amongst other things it contains the code of the interface which must be implemented by java storlets.

11.4 The storlet bus

The storlet bus is a communication channel between the storlet middleware at the Swift side and the factory daemon and the storlet daemon in the Docker container. For each Docker container (or Swift account) there is a communication channel with the storlet factory of that container. For each storlet daemon in the container there is a communication channel on which it listens for invocations. These channels are based on unix domain sockets.

11.5 IOStack integration

As we will see later in the section 12.5.2, the filter framework for object storage integrates Storlets as one of the filter execution environments of IOStack. Storlets act as an isolated filter execution environment to run computations on object requests with higher security guarantees. Combined with Crystal, Storlets are enhanced with pipelining and stage execution control (i.e., proxy/storage node) functionalities.

Furthermore, IOStack web dashboard integrates the deployment of Storlets as well as native filters, allowing an administrator to transparently define a filter pipeline that combines both kinds of filters.

In the context of the Gridpocket use case (SQL pushdown mechanism), the Storlet WSGI middleware in Swift was extended to support running Storlets at storage nodes for byte ranges. Also, we

contributed a new storlet that can perform projection and selection filters over CSV data. This work was explained in depth in Deliverable D4.2.

Part III
IOStack for Object Storage

12 Crystal: SDS for Multi-tenant Object Stores

The objective of Crystal is to constitute the first SDS platform for object storage that efficiently handles workload heterogeneity and applications with evolving requirements. To achieve this, Crystal separates high-level policies from the mechanisms that implement them at the data plane, to avoid hard-coding the policies in the system itself. As mentioned in Section 5.1, it uses three main abstractions: *filter*, *metric (or trigger)*, and *controller*, in addition to *policies*.

12.1 Abstractions in Crystal

Filter. A filter is a piece of code that a system administrator can inject into the data plane to perform custom computations on incoming object requests[14]. In Crystal, this concept is broad enough to include *computations on object contents* (e.g., compression, lambda functions), *data management* like caching or pre-fetching, and even *resource management* such as bandwidth differentiation (Fig. 17). A key feature of filters is that the instrumented system is oblivious to their execution and needs no modification to its implementation code to support them.

Inspection trigger. This abstraction represents information accrued from the system to automate the execution of filters. There are two types of information sources. The first corresponds to the *real-time metrics* from the running workloads, such as the number of `GET` operations per second of a data container or the IO bandwidth allocated to a tenant. As with filters, a fundamental feature of workload metrics is that they can be deployed at runtime. A second type of source is the *metadata from the objects* themselves. Such metadata is typically associated with read and write requests and includes properties such as the size or type of objects.

Controller. In Crystal, a controller represents an algorithm that manages the behavior of the data plane based on monitoring metrics. A controller may contain a *simple rule to automate* the execution of a filter, or a complex algorithm requiring *global visibility* of the cluster to control a filter's execution under multi-tenancy. Crystal builds a logically centralized control plane formed by supervised and distributed controllers. This allows an administrator to easily deploy new controllers on-the-fly that cope with the requirements of new applications.

Policy. Our policies should be extensible to allow the system to satisfy evolving requirements. This means that the structure of policies must facilitate the incorporation of new filters, triggers, and controllers.

To succinctly express policies, Crystal abides by a structure similar to that of the popular IFTTT (If-This-Then-That) service [52]. This service allows users to express small rule-based programs, called "recipes", using *triggers* and *actions*. For example:

`TRIGGER`: compressibility of an object is $> 50\%$

`ACTION:` compress

`RECIPE: IF` compressibility is $> 50\%$ `THEN` compress

An IFTTT-like language can reflect the extensibility capabilities of the SDS system; at the data plane, we can infer that triggers and actions are translated into our inspection triggers and filters, respectively. At the control plane, a policy is a "recipe" that guides the behavior of control algorithms. Such an apparently simple policy structure can express different policy types. On the one hand, Fig. 17 shows *storage automation policies* that enforce a filter either statically or dynamically based on simple rules. For instance, `P1` enforces compression and encryption on document objects of tenant `T1`, whereas `P2` applies data caching on small objects of container `C1` when the number of `GETs`/second is > 5. Such policies also allow us to write custom functions to be executed on data streams upon object requests (`P3`). On the other hand, such policies can also express objectives to be achieved by

[14] Filters work in an *online* fashion. To explore the feasibility of batch filters on already stored objects is matter of future work.

	FOR	*[TARGET]*	WHEN	*[TRIGGER CLAUSE]*	DO	*[ACTION CLAUSE]*
P1		**TENANT** T1		OBJECT_TYPE=DOCS		**SET** COMPRESSION **WITH** TYPE=LZ4, **SET** ENCRYPTION
P2		**CONTAINER** C1		GETS_SEC > 5 **AND** OBJECT_SIZE<10M		**SET** CACHING **ON PROXY TRANSIENT**
P3		**CONTAINER** C2				**SET** LAMBDAS **WITH** lambda1=filter(s → s.startsWith("storage_done"))
P4		**TENANT** T2				**SET** BANDWIDTH **WITH** GET_BW=30MBps

Content management policy Data management policy Resource management policy
Storage automation policy Globally coordinated policy

Figure 17: Structure of the Crystal DSL.

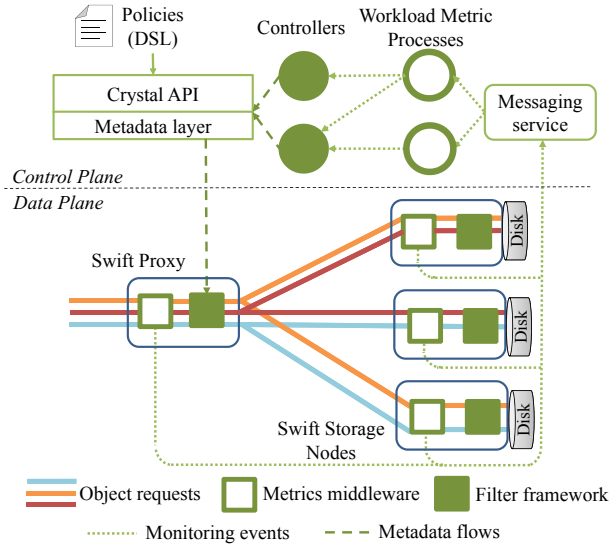


Figure 18: High-level overview of Crystal's architecture materialized on top of OpenStack Swift.

controllers requiring *global visibility and coordination* capabilities of the data plane. That is, `P4` tells a controller to provide at least 30MBps of aggregated `GET` bandwidth to tenant `T2` under a multi-tenant workload.

12.2 System Architecture

Fig. 18 presents Crystal's architecture, which consists of:

Control Plane. Crystal provides administrators with a system-agnostic DSL (Domain-Specific Language) to define SDS services via high-level policies. The DSL "vocabulary" can be extended at runtime with new filters and inspection triggers. The control plane includes an API to compile policies and to manage the life-cycle and metadata of controllers, filters and metrics (see Table 4).

Moreover, the control plane is built upon a distributed model. Although logically centralized, the controller is, in practice, split into a set of autonomous micro-services, each running a separate control algorithm. Other micro-services, called workload metric processes, close the control loop by exposing monitoring information from the data plane to controllers. The control loop is also extensible, given that both controllers and workload metric processes can be deployed at runtime.

Data Plane. Crystal's data plane has two core extension points: Inspection triggers and filters. First, a developer can deploy new workload metrics at the data plane to feed distributed controllers with new runtime information on the system. The metrics framework runs the code of metrics and publishes monitoring events to the messaging service. Second, data plane programmability and extensibility is delivered through the filter framework, which intercepts object flows in a transparent manner and runs computations on them. A developer integrating a new filter only has to contribute the logic; Crystal manages the deployment and execution of the filter.

12.3 Control Plane

The control plane allows writing policies that adapt the data plane to manage multi-tenant workloads. It is formed by the DSL, the API and distributed controllers.

Crystal Controller Calls	*Description*
`add_policy`, `delete_policy`, `list_policies`	Policy management API calls. For storage automation policies, the `add_policy` call can either directly enforce the filter or deploy a controller to do so. For globally coordinated policies, the call sets an objective at the metadata layer.
`register_keyword`, `delete_keyword`	Calls that interact with Crystal registry to associate DSL keywords with filters, inspection triggers, or coin new terms to be used as trigger conditions (e.g., `DOCS`).
`deploy_controller`, `kill_controller`	These calls are used to manage the life-cycle of distributed controllers and workload metric processes in the system.
Filter Framework Calls	*Description*
`deploy_filter`, `undeploy_filter`, `list_filters`	Calls for deploying, undeploying, and listing filters associated to a target. `deploy/undeploy_filter` calls interact with the filter framework at the data plane for enabling/disabling filter binaries to be executed on a specific target.
`update_slo`, `list_slo`, `delete_slo`	Calls to manage "tenant objectives" for coordinated resource management filters. For instance, bandwidth differentiation controllers take as input this information in order to provide an aggregated IO bandwidth share at the data plane.
Workload Metric Calls	*Description*
`deploy_metric`, `delete_metric`	Calls for managing workload metrics at the data plane. These calls also manage workload metric processes to expose data plane metrics to the control plane.

*For the sake of simplicity, we do not include call parameters in this table.

Table 4: Main calls of Crystal controller, filter framework, and workload metrics management APIs.

12.4 Crystal DSL

Crystal's DSL hides the complexity of low-level policy enforcement, thus achieving simplified storage administration (Fig. 17). The structure of our DSL is as follows:

Target: The target of a policy represents the recipient of a policy's action (e.g., filter enforcement) and it is mandatory to specify it on every policy definition. To meet the specific needs of object storage, targets can be *tenants*, *containers*, or even individual *data objects*. This enables high management and administration flexibility.

Trigger clause (optional): Dynamic storage automation policies are characterized by the trigger clause. A policy may have one or more trigger clauses —separated by `AND/OR` operands— that specify the workload-based situation that will trigger the enforcement of a filter on the target. Trigger clauses consist of inspection triggers, operands (e.g., $>$, $<$, $=$) and values. The DSL exposes both types of inspection triggers: workload metrics (e.g., `GETS_SEC`), and request metadata (e.g., `OBJECT_SIZE<512`).

Action clause: The action clause of a policy defines how a filter should be executed on an object request once the policy takes place. The action clause may accept parameters after the `WITH` keyword in form of key/value pairs that will be passed as input to customize the filter execution. Retaking the example of a compression filter, we may decide to enforce compression using a `gzip` or an `lz4` engine, and even their compression level.

To cope with object stores formed by proxies/storage nodes (e.g., Swift), our DSL enables to explicitly control the execution stage of a filter with the `ON` keyword. Also, dynamic storage automation policies can be *persistent or transient*; a persistent action means that once the policy is triggered the filter enforcement remains indefinitely (by default), whereas actions to be executed only during the period where the condition is satisfied are transient (keyword `TRANSIENT`, `P2` in Fig. 17).

The vocabulary of our DSL can be extended on-the-fly to accommodate new filters and inspection triggers. In Fig. 17 we can use keywords `COMPRESSION` and `DOCS` in `P1` once we associate "`COMPRESSION`" with a given filter implementation and "`DOCS`" with some file extensions, respectively (see Table 4).

The Crystal DSL has other features: i) *specialization* of policies based on the target scope, so that if several policies apply to the same request, only the most specific one is executed (e.g., container-level policy is more specific than a tenant-level one), ii) *pipelining* several filters on a single request (e.g., compression + encryption) ordered as they are defined in the policy, similar to stream processing frameworks [29], and iii) *grouping*, which enables to enforce a single policy to a group of targets; that is, we can create a group like `WEB_CONTAINERS` to represent all the containers that serve Web pages.

As visible in Table 4, Crystal offers a DSL compilation service via API calls. Crystal compiles static automation policies as *target→filter* relationships at the metadata layer. Next, we show how dynamic policies are materialized as controllers that extend the control plane.

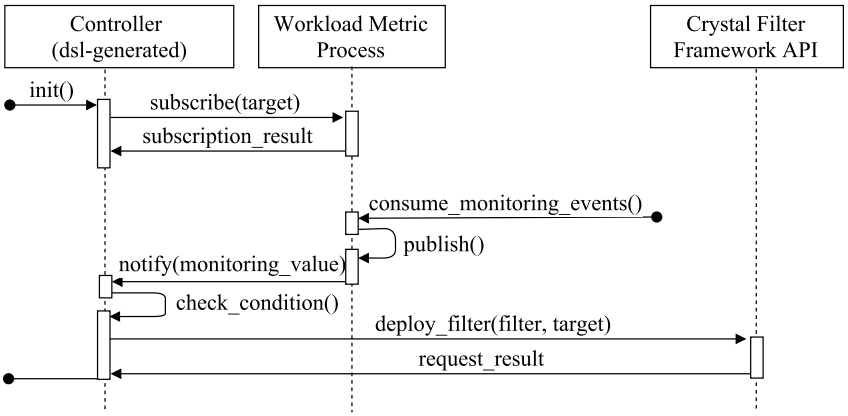


Figure 19: Interactions among dsl-generated controllers, workload metric processes and the filter framework.

12.4.1 Distributed Controllers

Crystal enables the creation of distributed controllers, in form of supervised processes, which can be deployed in the system at runtime to add new behaviors to the control plane [53, 54, 55][15].

We offer two types of controllers: *dsl-generated* and *custom* controllers. On the one hand, the Crystal DSL transparently compiles dynamic automation storage policies into dsl-generated controllers (e.g., `P2` in Fig. 17) that interact with our filter framework. On the other hand, custom controllers are not generated by the DSL; instead, by following a small set of conventions, Crystal enables developers to deploy controllers that contain complex algorithms rather than simple activation rules (e.g., `P3` in Fig. 17). For instance, this allowed us to deploy distributed IO bandwidth control algorithms (Section 12.6).

Dynamic control loop: Distributed controllers can be programmed to react and adapt to changes in the underlying storage system. Therefore, controllers must receive up-to-date inspection information from the data plane. As we explain in the next section, Crystal exposes monitoring metrics of the current state of workloads.

Technically, distributed controllers —dsl-generated and custom— and workload metrics interact in a publish/subscribe fashion. As visible in Fig. 19, once initialized, a distributed controller subscribes to the appropriate workload metric, taking into account the target granularity. The subscription request of a controller specifies the target to which it is interested in, such as tenant `T1` or container `C1`. Once the workload metric receives the subscription request, it adds the controller to its observer list. Periodically, the workload metric notifies the activity of the different targets to the interested controllers that may trigger the execution of filters.

12.5 Data Plane

To deal with heterogeneity, we offer two main extension hooks: Inspection triggers and a filter framework.

12.5.1 Inspection Triggers

Inspection triggers enable controllers to dynamically respond to workload changes in real time. Specifically, we consider two types of introspective information sources: *object metadata* and *monitoring metrics*.

First, some object requests embed semantic information related to the object at hand in form of metadata. Crystal enables administrators to enforce storage filters based on such metadata. Concretely, our filter framework middleware (see Section 12.5.2) is capable of analyzing at runtime HTTP metadata of object requests to execute filters based on the object size or file type, among others.

[15]Note that these controllers are also applicable to control dynamic functionalities in Zoe and Konnector.

Second, Crystal builds a metrics middleware to add new workload metrics on-the-fly. In our design, a new workload metric can inject events to the monitoring service without interfering with existing ones (Table 4). A salient feature of our metrics framework is that it enables developers to plug-in metrics to inspect both the type of requests and their contents (e.g., compressibility).

At the top level of our monitoring system we find *workload metric processes*. These processes are devised to consume and aggregate monitoring information to be published to controllers (see Fig. 19). Each workload metric process consumes from a different monitoring metric at the data plane. For the sake of simplicity and isolation [55], we advocate to separate workload metrics not only per metric type, but also by target granularity.

12.5.2 Filter Framework

The Crystal filter framework enables developers to deploy and run general-purpose code on object requests. Crystal borrows ideas from active storage literature [56, 57] as a mean of building filters to enforce policies.

Our framework achieves flexible execution of filters. First, it enables to easily *pipeline several filters* on a single storage request. Currently, the execution order of filters is set explicitly by the administrator, although filter metadata can be exploited to avoid conflicting filter ordering errors [58]. Second, to deal with object stores composed by proxy/storage nodes, Crystal allows administrators to define the *execution point of a filter*.

To this end, the Crystal filter framework consists of i) filter middleware, and ii) filter execution environments.

Filter middleware: Our filter middleware is a hook to intercept data streams and classify incoming requests. Upon a new object request, the Crystal middleware may contact the metadata layer to infer the filters to be executed on that request depending on the target. If the target has associated filters to be enforced, the Crystal middleware sets the appropriate HTTP headers in the request (e.g., `GET`, `PUT`) for triggering the filter execution.

Filters that change the content of data objects may receive a special treatment, such as in case of compression or encryption filters. To wit, if we create a filter with the *reverse* flag enabled, it means that the execution of the filter when the object was stored should be always undone upon a `GET` request. For instance, this yields that we may activate data compression on certain periods, but tenants will always download decompressed data objects. To this end, we store data objects with an *extended metadata* to keep track of the enforced reverse filters. Upon a `GET` request, such metadata is fetched by the Crystal middleware to trigger reverse transformations on the data object prior to the execution of regular filters.

Filter execution environments: Thanks to the interception capabilities of our middleware, it can support multiple execution platforms. Crystal features:

Isolated filter execution: Crystal provides an isolated filter execution environment to compute on object requests with higher security guarantees. To this end, we extended the Storlets framework [30] with pipelining and stage execution control functionalities. Storlets provide Swift with the capability to run computations near the data in a secure and isolated manner making use of `Docker` containers [59]. Invoking a Storlet on a data object is done in an isolated manner so that the data accessible by the computation is only the object's data and its user metadata. Moreover, a `Docker` container only executes filters of a single tenant.

Native filter execution: The isolated filter execution environment trades-off higher security for lower communication capabilities and interception flexibility. For this reason, we also contribute an alternative way to intercept and execute code natively. As with Storlets, a developer can install on runtime in Crystal code modules as filters following simple design guidelines. However, native filters can i) execute code at all the possible points of a request's life-cycle, and ii) communicate with external components (e.g, metadata layer), as well as to access storage devices (e.g., SSD). As Crystal is devised to execute trusted code from administrators, this environment represents a more flexible alternative.

12.6 Hands On: Extending Crystal

Next, we show the benefits of Crystal's design by extending the system with data management filters and distributed control of IO bandwidth for OpenStack Swift.

12.7 New Storage Automation Policies

Goal: To define policies that enforce filters, such as *compression*, *encryption*, or *caching*, even dynamically:

```
P1:FOR TENANT T1 WHEN OBJECT_TYPE=DOCS DO SET COMPRESSION ON PROXY, SET ENCRYPTION ON STORAGE_NODE
P2:FOR CONTAINER C1 WHEN GETS_SEC > 5 DO SET CACHING
```

Data plane (Filters): To enable such storage automation policies, we first need to *develop the filters* at the data plane. In Crystal this can be done using either native or isolated execution environments.

The next code snippet shows how to develop a filter for our isolated execution environment. A system developer only has to create a class that implements an interface (`IStorlet`), providing the actual data transformations on the object request streams (`iStream`, `oStream`) inside the `invoke` method. We implemented the compression (`gzip` engine) and encryption (AES-256) filters using storlets, whereas the caching filter exploits SSD drives at proxies via our native execution environment. Then, once these filters were developed, we installed them via the Crystal filter framework API.

```
public class StorletName implements IStorlet {

  @Override
  public void invoke(ArrayList<StorletInputStream> iStream,
      ArrayList<StorletOutputStream> oStream,
      Map<String, String> parameters, StorletLogger logger)
      throws StorletException {
          //Develop filter logic here
  }
}
```

Data plane (Monitoring): Via the Crystal API (see Table 4), we deployed metrics that capture various workload aspects (e.g., `PUTs/GETs` per second of a tenant) to satisfy policies like `P2`. Similarly, we deployed the corresponding workload metrics processes (one per metric and target granularity) that aggregate such monitoring information to be published to controllers. Also, our filter framework middleware is already capable of enforcing filters based on object metadata, such as object size (`OBJECT_SIZE`) and type (`OBJECT_TYPE`).

Control Plane: Finally, we registered intuitive keywords for both filters and workload metrics at the metadata layer (e.g., `CACHING`, `GET_SEC_TENANT`) using the Crystal registry API. To achieve `P1`, we also registered the keyword `DOCS`, which contains the file extensions of common documents (e.g., `.pdf`, `.doc`). At this point, we can use such keywords in our DSL to design new storage policies.

12.8 Programming Lambdas for Reducing Analytics Data Ingestion

Goal: Enabling Crystal to enforce policies that reduce data ingestion of analytics applications:

```
P3:FOR CONTAINER C1 DO SET LAMBDAS WITH lambda1=filter(s → s.startsWith("storage_done"))
```

Data plane (Filter). One of the distinctive features of Crystal is that filters can intercept data flows and compute on data. This allows system developers to create filters that discard useless data during the ingestion phase of a data analytics application running in a separate compute cluster. This may help to save bandwidth in over-subscribed inter-cluster networks, and even to improve job execution times [21].

To materialize such policies, we created an isolated (Storlet) filter that enables to "pushdown" and execute lambda functions passed by parameter on object data streams. For the sake of clarity, in this context we refer to a lambda function —also known as *closure* in programming languages— as a self-contained piece of logic that operates synchronously on data stream records and can be chained to form a pipeline. Specifically, our implementation uses the Java 8 Stream API [42] to create data streams and operate on them with a rich set of calls.

Our objective is to allow administrators to define policies with lambda functions as input for the filter. As a result, administrators would be able to discard unnecessary data during the ingestion phase of analytics. For instance, they can define lambdas using the `map` call to discard rows of a

dataset, or the `filter` call to discard lines within an object based on a predicate[16]. As visible in `P3`, lambdas are defined in the form of tuples to express the order of the function that will be executed and its body/type. This enables an administrator to control the pipeline of lambda functions to be executed on a data stream. Upon a storage request, the filter sorts the lambda tuples and compiles them on runtime as defined in the policy (e.g., `filter(s → s.startsWith("storage_done"))` in `P3`). In the case of a successful compilation, the filter encodes the byte-level stream into a Java 8 Stream and applies the lambda(s) on each record. Naturally, in the case the lambda compilation raises an error, the data object is retrieved to the user as usual. Moreover, the filter contains two optimizations in order to minimize the compilation overhead. First, the encoding overhead related to transform byte-level data streams into text is only performed when there are lambda functions to execute. Second, we implemented a cache of lambda functions, so a lambda requires compilation only the first time it reaches the filter; subsequent executions of a given lambda function reuse the compiled function object stored at the cache. Overall, we confirmed that the compilation overhead of simple lambdas ranges between 3ms-9ms in most cases. We consider this overhead to be affordable, specially taking into account the data transfer and job execution time gains reported in Section 15.5.

Control plane. We registered the filter with the `LAMBDAS` keyword via the Crystal registry API.

12.9 Global Management of IO Bandwidth

Goal: To provide Crystal with a means of defining policies that enforce a global IO bandwidth SLO on `GETs/PUTs`:

```
P4:FOR TENANT T1 DO SET BANDWIDTH WITH GET_BW=30MBps
```

Data plane (Filter). To achieve global bandwidth SLOs on targets, we first need to locally control the bandwidth of object requests. Intuitively, bandwidth control in Swift may be performed at the proxy or storage node stages. At the proxy level this task may be simpler, as fewer nodes should be coordinated. However, this approach is agnostic to the background tasks (e.g., replication) executed by storage nodes, which impact on performance [60]. We implemented a native bandwidth control filter that enables the enforcement at both stages.

Our filter dynamically creates threads that serve and control the bandwidth allocation for individual tenants, either at proxies or storage nodes. Our filter garbage-collects control threads that are inactive for a certain timeout. Moreover, it has a consumer process that receives bandwidth assignments from a controller to be enforced on a tenant's object streams. Once the consumer receives a new event, it propagates the assignments to the filter that immediately take effect on current transfers.

Data plane (Monitoring): To build the control loop, our bandwidth control service integrates individual monitoring metrics per type of traffic (i.e., `GET`, `PUT`, `REPLICATION`); this makes it possible to define policies for each type of traffic if needed. In essence, monitoring events contain a data structure that represents the bandwidth share that tenants exhibit at proxies or per storage node disk. We also deployed workload metric processes to expose these events to controllers.

Control plane. We deployed Algorithm 1 as a global controller to orchestrate our bandwidth differentiation filter. We aim at satisfying three main requirements: i) A *minimum bandwidth share per tenant*, ii) *Work-conservation* (do not leave idle resources), and iii) *Equal shares of spare bandwidth* across tenants. The challenge is to meet these requirements considering that we do not control neither the data access of tenants nor the data layout of Swift [61, 62].

To this end, Algorithm 1 works in three stages. First, the algorithm tries to ensure the SLO for tenants specified in the metadata layer by resorting to function `minSLO` (requirement 1, line 6). Essentially, `minSLO` first assigns a proportional bandwidth share to tenants with guaranteed bandwidth. Note that such assignment is done in descending order based on the number of parallel transfers per tenant, provided that tenants with fewer transfers have fewer opportunities of meeting their SLOs. Moreover, `minSLO` checks whether there exist overloaded nodes in the system. In the affirmative case, the algorithm tries to reallocate bandwidth of tenants with multiple transfers from overloaded nodes to idle ones. If no reallocation is possible, the algorithm reduces the bandwidth share of tenants with

[16] Although we focus on pushing down lambdas that discard data, the Storlet can also handle other lambdas from the Java 8 Stream API [42].

Algorithm 1 `computeAssignments` pseudo-code embedded into a bandwidth differentiation controller

```
1:  function COMPUTEASSIGNMENTS(info):
2:                                          ▷ Retrieve the defined tenant SLOs from the metadata layer
3:      SLOs ← getMetadataStoreSLOs();
4:                                          ▷ Compute assignments on current tenant transfers to meet SLOs
5:      SLOAssignments ← minSLO(info, SLOs);
6:                                          ▷ Estimate spare bw at proxies/storage nodes based on current usage
7:      spareBw ← min(spareBwProxies(SLOAssignments), spareBwStorageNodes(SLOAssignments));
8:      spareBwSLOs ← {};
9:                                          ▷ Distribute spare bandwidth equally across all tenants
10:
11:     for tenant in info do
12:         spareBwSLOs[tenant] ← spareBW / numTenants(info);
13:     end for
14:                                         ▷ Calculate assignments to achieve spare bw shares for tenants
15:     spareAssignments ← spareSLO(SLOAssignments, spareBwSLOs);
16:                                         ▷ Combine SLO and spare bw assignments on tenants
17:     return SLOAssignments ∪ spareAssignments;
18: end function
```

SLOs on overloaded nodes.

Once Algorithm 1 has calculated the assignments for tenants with SLOs, it estimates the spare bandwidth available to achieve full utilization of the cluster (requirement 2, line 8). Note that the notion of spare bandwidth depends on the cluster at hand, as the bottleneck may be either at the proxies or storage nodes.

Algorithm 1 builds a new assignment data structure in which the spare bandwidth is equally assigned to all tenants. The algorithm proceeds by calling function `spareSLO` to calculate the spare bandwidth assignments (requirement 3, line 15). Note that `spareSLO` receives the `SLOAssignments` data structure that keeps the already reserved node bandwidth according to the SLO tenant assignments. The algorithm outputs the combination of SLO and spare bandwidth assignments per tenant. While more complex algorithms can be deployed in Crystal [63], our goal in Algorithm 1 is to offer an attractive simplicity/effectiveness trade-off, validating our bandwidth differentiation framework.

12.10 Crystal Prototype

Web page	`http://crystal-sds.org/`
Source Code	`https://github.com/Crystal-SDS`
Documentation	`https://github.com/Crystal-SDS/controller`
Continuous Integration	`https://travis-ci.org/Crystal-SDS/controller`
Test Coverage	`https://coveralls.io/github/Crystal-SDS/controller`
Code Quality	`https://landscape.io/github/Crystal-SDS/controller`

We tested the Crystal prototype in OpenStack Ocata version. The APIs at the control plane are implemented with Django framework to ease the integration with other architecture components. As metadata layer, in this work Crystal uses Redis 3.0. We resort to PyActive [64] for building supervised controllers and workload metric processes that can communicate either via TCP or a Message Oriented Middleware (MOM). We use RabbitMQ 3.6 as a communication service for monitoring information. Crystal also provides a dashboard that extends the OpenStack Horizon to ease the management of the SDS framework by administrators. The code of Crystal for object storage is publicly available[17] and our contributions to the Storlets framework were accepted by the official OpenStack repository.

12.11 Related Systems

SDS Systems. IOFlow [25], now extended as sRoute [28], was the first complete SDS architecture. IOFlow enables end-to-end (e2e) policies to specify the treatment of IO flows from VMs to shared storage. This was achieved by introducing a queuing abstraction at the data plane and translating

[17] `https://github.com/Crystal-SDS`

high-level policies into queuing rules. The original focus of IOFlow was to enforce e2e bandwidth targets, which was later augmented with caching and tail latency control in [28, 27].

Despite Crystal shares with IOFlow design concepts (e.g., policies, control/data planes), our target is different; Crystal pursues to configure and optimize object stores to the evolving needs of applications, for it needs a richer data plane and a different suite of management abstractions and enforcement mechanisms. For instance, tenants require mechanisms to inject custom logic and abstractions to specify not only system activities but also application-specific transformations on objects.

Retro [60] is a framework for implementing resource management policies in multi-tenant distributed systems. It can be viewed as an incarnation of SDS, because as IOFlow and Crystal, it separates the controller from the mechanisms that implement it. A major contribution of Retro is the development of abstractions to enable policies that are system- and resource-agnostic. Crystal shares the same spirit of requiring low develop effort. However, its abstractions are different. Crystal must abstract not only resource management; it must enable the concise definition of policies that enable high levels of programmability to suit application needs. Retro is only extensible to handle custom resources.

Vertigo [65] is a framework where the control logic is directly embedded into data objects in the form of *micro-controllers*. This enables object-level policies avoiding a centralized control point. We believe that Crystal may also support policies that orchestrate micro-controllers.

IO bandwidth differentiation. Enforcing bandwidth SLOs in shared storage has been a subject of intensive research over the past 10 years, specially in block storage [66, 63, 67, 68, 69, 25, 60]. However, object stores have received much less attention in this regard; vanilla Swift only provides a non-automated mechanisms for limiting the "number of requests" [70] per tenant, instead of IO bandwidth. In fact, this problem resembles the one stated by Wang et al. [62] where multiple clients access a distributed storage system with different data layout and access patterns, yet the performance guarantees required are global. To our knowledge, Wu et al. [68] is the only work addressing this issue in object storage. It provides SLOs in Ceph by orchestrating local rate limiters offered by a modified version of the underlying file system (EBOFS). However, this approach is intrusive and restricted to work with EBOFS. In contrast, Crystal transparently intercepts and limits requests streams, enabling developers to design new algorithms that provide distributed bandwidth enforcement [71, 61].

Active storage. The early concept of *active disk* [56], i.e., a HDD with computational capacity, was borrowed by distributed file system designers in HPC environments in the last decade to give birth to active storage. The goal was to diminish the amount of data movement between storage and compute nodes [72, 73]. Piernas et al. [57] presented an active storage implementation integrated in the Lustre file system that provides flexible execution of code near to data in the user space. Crystal goes beyond active storage. It exposes through the filter abstraction a mechanism to inject custom logic into the data plane and expose it to management policies. This requires filters to be deployable at runtime, support sandbox execution [30], and be part of complex workflows.

13 Evaluation of Crystal

Next, we evaluate a prototype of Crystal for OpenStack Swift in terms of flexibility, performance and overhead.

Objectives: Our evaluation addresses aims at demonstrating that: i) Crystal can define policies at multiple granularities, achieving administration flexibility; ii) The enforcement of storage automation filters can be dynamically triggered based on workload conditions; iii) Crystal enables administrators to program lambdas that discard useless data to be ingested by analytics frameworks; iv) Crystal achieves accurate distributed enforcement of IO bandwidth SLOs on different tenants; v) Finally, Crystal has low execution/monitoring overhead.

Workloads: We resort to well-known benchmarks and replays of real workload traces. First, we use `ssbench` [74] to execute stress-like workloads on Swift. `ssbench` provides flexibility regarding the type (CRUD) and number of operations to be executed, as well as the size of files generated. All these parameters can be specified in the form of configuration "scenarios".

Type of analytics	*Data*	*Description*
SQL Query (Q1)	GridPocket (140GB)	Get the aggregated energy consumption per energy meter of Paris meters in January 2015.
SQL Query (Q2)	GridPocket (140GB)	Get the geographic coordinates of the electricity meter that reported highest consumption in 2012.
SQL Query (Q3)	GridPocket (140GB)	Get an ordered list of cities by total energy consumption in the last 3 years.
Batch job (K-means)	UbuntuOne (100GB)	Classify UbuntuOne users according to their types of storage operations (k=5, iterations=20)

Table 5: Analytics jobs executed to demonstrate the data discard capabilities of Crystal filters.

To evaluate Crystal under real-world object storage workloads, we collected the following traces[18]: ii) The first trace captures 1.28TB of a write-dominated (79.99% write bytes) document database workload storing 817K car testing/standardization files (mean object size is 0.91MB) for 2.6 years at Idiada; an automotive company. i) The second trace captures 2.97TB of a read-dominated (99.97% read bytes) Web workload consisting of requests related to 228K small data objects (mean object size is 0.28MB) from several Web pages hosted at Arctur datacenter for 1 month. We developed our own workload generator to replay a part of these traces (12 hours), as well as to perform experiments with controllable rates of requests. Our workload generator resorts to SDGen [75] to create realistic contents for data objects based on the file types described in the workload traces.

To assess our lambda filter for analytics, we used both real Spark SQL queries from a GridPocket —a European smart grid energy company— as well as to standard Spark k-means jobs (see Table 5). For SQL queries, we resorted to a synthetic data generator[19] that reproduces realistic data used in GridPocket but avoids disclosing private information; on the other hand, we fed the k-means job with real traces of a cloud storage system [43].

Platform: We ran our experiments in a cluster formed by Dell PowerEdge nodes with Intel Xeon E5-2403 processors. The Swift cluster has 1 proxy node (28GB RAM, 1TB HDD, 4-core) and 7 storage nodes (16GB RAM, 2x1TB HDD, 4-core). 4 nodes (32GB RAM, 1TB HDD, 24-core) were used as compute nodes to execute workloads and analytics. Also, there is 1 large node that runs the OpenStack services and the Crystal control plane (i.e., API, controllers, messaging, metadata store). Nodes in the cluster are connected via 1 GbE switched links.

13.1 Evaluating Storage Automation

Next, we present a battery of experiments that demonstrate the feasibility and capabilities of storage automation with Crystal (see Section 12.7). To this end, we make use of synthetic workloads and real trace replays (Idiada, Arctur). These experiments have been executed at the compute nodes against 1 swift proxy and 6 storage nodes.

Storage management capabilities of Crystal. Fig. 20 shows the execution of several storage automation policies on a workload related to containers `C1` and `C2` belonging to tenant `T1`. Specifically, we executed a write-only synthetic workload (4PUT/second of 1MB objects) in which data objects stored at `C1` consist of random data, whereas `C2` stores highly redundant objects.

Due to the security requirements of `T1`, the first policy defined by the administrator is to encrypt his data objects (`P1`). Fig. 20 shows that the `PUT` operations of *both containers* exhibit a slight extra overhead due to encryption, given that the policy has been defined at the tenant scope. There are two important aspects to note from `P1`: First, the execution of encryption on `T1`'s requests is isolated from filter executions of other tenants, providing higher security guarantees [30] (Storlet filter). Second, the administrator can enforce the filter at the storage node, so as not to overload the proxy with the overhead of encrypting data objects (`ON` keyword).

After policy `P1` was enforced, the administrator decided to optimize the storage space of `T1`'s

[18] Available at `http://iostack.eu/datasets-menu`

[19] `https://github.com/gridpocket/project-iostack`

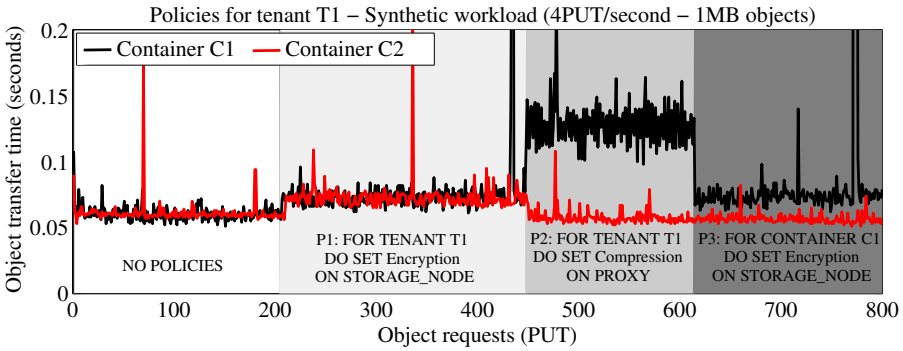


Figure 20: Enforcement of compression/encryption filters.

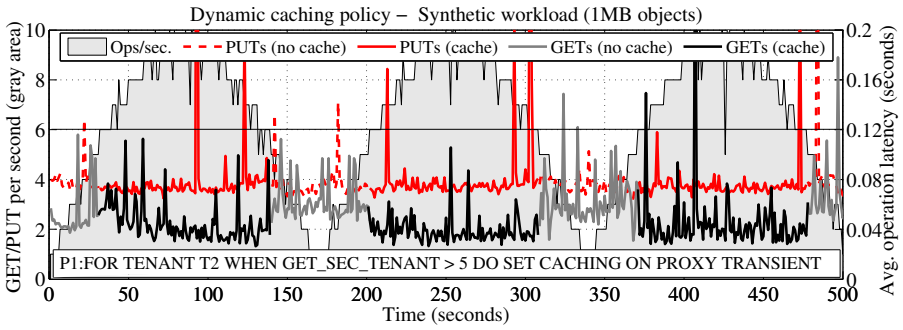


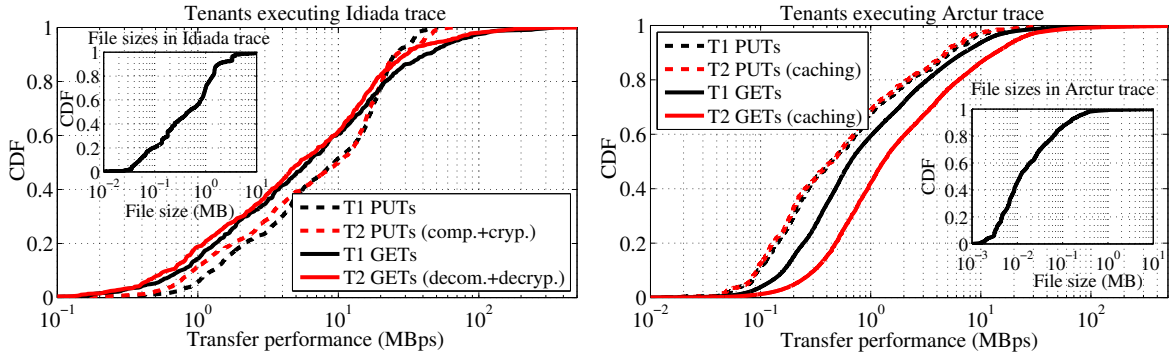
Figure 21: Dynamic enforcement of caching filter.

objects by enforcing compression (P2). P2 also enforces compression at the proxy node to minimize communication between the proxy and storage node (ON PROXY). Note that the enforcement of P1 and P2 demonstrates the filter pipelining capabilities of our filter framework; once P2 is defined, Crystal enforces compression at the proxy node and encryption at storage nodes for each object request. Also, as shown in Section 12.5, the filter framework tags objects with extended metadata to trigger the reverse execution of these filters on GET requests (i.e., decryption and decompression, in that order).

However, the administrator realized that the compression filter on C1's requests exhibited higher latency and provided no storage space savings (incompressible data). To overcome this issue, the administrator defined a new policy P3 that essentially enforces only encryption on C1's requests. After defining P3, the performance of C1's requests exhibits the same behavior as before the enforcement of P2. Thus, the administrator can manage storage at different granularities, such as tenant or container. Furthermore, the last policy also proves the usefulness of policy specialization; policies P1 and P2 apply to C2 at the tenant scope, whereas the system only executes P3 on C1's requests, as it is the most specialized policy.

Dynamic storage automation. Fig. 21 shows a dynamic caching policy (P1) on one tenant. The filter implements LRU eviction and exploits SSD drives at the proxy to improve object retrievals. We executed a synthetic oscillatory workload of 1MB objects (gray area) to verify the correct operation of automation controllers.

In Fig. 21, we show the average latency of PUT/GET requests and the intensity of the workload. As shown, the caching filter takes place when the workload exceeds 5 GETs per second. At this point, the filter starts caching objects at the proxy SSD on PUTs, as well as to lookup the SSD to retrieve potentially cached objects on GETs. First, the filter provides performance benefits for object retrievals; when the caching filter is activated, object retrievals are in median 29.7% faster compared to non-caching periods. Second, we noted that the costs of executing asynchronous writes on the SSD upon PUT requests may be amortized by offloading storage nodes; that is, the average PUT latency is in



(a) Idiada workload (file sizes in inner plot). (b) Arctur workload (file sizes in inner plot).

Figure 22: Policy enforcement on real trace replays.

median 2% lower when caching is activated. This may be because storage nodes are mostly free to execute writes, as a large fraction of GETs are being served at the proxy's cache.

In conclusion, Crystal's control loop enables dynamic enforcement of storage filters under variable workloads. Moreover, native filters in Crystal allow developers to build complex data management filters.

Managing real workloads. Next, we show how Crystal policies can handle real workloads (12 hours). We compress and encrypt documents (P1 in Fig. 17) on a replay of the Idiada trace (write-dominated), whereas we enforce caching of small files (P2 in Fig. 17) on a replay of Arctur workload (read-dominated).

Fig. 22a shows the request bandwidth exhibited during the execution of the Idiada trace. Specifically, we executed two concurrent workloads, each associated to a different tenant. We enforced compression and encryption only on tenant T2. We observed that tenant T2's transfers are over 13% and 7% slower compared to T1 for GETs and PUTs, respectively. This is due to the computation overhead of enforcing filters on T2's document objects. As a result, T2's documents consumed 65% less space compared to T1 with compression and they benefited from higher data confidentially thanks to encryption.

Fig. 22b shows tenants T1 and T2, both concurrently running a trace replay of Arctur. By executing a dynamic caching policy, T2's GET requests are in median 1.9x faster compared to T1. That is, as the workload of Arctur is intense and almost read-only, caching was enabled for tenant T2 for most of the experiment. Moreover, because the requested files fitted in the cache, the SSD-based caching filter was very beneficial to tenant T2. The median write overhead of T2 compared to T1 was 4.2%, which suggests that our filter efficiently intercepts object streams for doing parallel writes at the SSD.

Our results with real workloads suggest that Crystal is practical for managing multi-tenant object stores.

13.2 Pushing Down Lambdas for Boosting Big Data Analytics

In this section, we evaluate the benefits of Crystal filters for reducing data ingestion in Big Data analytics, especially considering a multi-tenant object store (see Section 12.8). For this battery of experiments, we executed one ssbench workload (24 threads) executing 20K 16MB GETs in one node representing an IO intensive tenant, as well as 3 compute nodes —each one contributing with 12 CPUs and 28GB of RAM— running Spark 2.1.1 in cluster mode as a tenant executing analytics. We used 3-way data replication and we configured Swift with 1 proxy/7 storage nodes to emulate a scenario with scarce outbound bandwidth accessed by multiple tenants.

Data transfer reduction. Next, we analyze the extent of data transfer reduction during the ingestion phase of analytics applications described in Table 5. To this end, Table 6 provides a quantitative notion of the data transfers savings that can be achieved thanks to our "lambda pushdown" filter.

Job name	Dataset	Data Transfer Reduction	Lambda functions written by the administrator in the policy definition
SQL Q1	GridPocket	99.96% (57MB ingested)	`map(s -> { List<String> l = new ArrayList<String>(11); String[] a = s.split(","); for (int i=0; i<11; i++) //GridPocket dataset has 11 columns separated by commas if ((i==0 || i==1 || i==5 || i==7) && i<a.length) l.add(a[i]); else l.add(""); return l; } //We only need 4 columns and rows of January 2015 and meters located at Paris filter(l -> l.get(0).startsWith("2015-01") && l.get(7).equals("Paris"))`
SQL Q2	GridPocket	99.98% (20MB ingested)	`map(s -> { List<String> l = new ArrayList<String>(11); String[] a = s.split(","); for (int i=0; i<11; i++) //GridPocket dataset has 11 columns separated by commas if ((i==0 || i==4 || i==5 || i==9 || i==10) && i<a.length) l.add(a[i]); else l.add(""); return l; } //We only need 5 columns and rows belonging to year 2012 and electric meters filter(l -> l.get(0).startsWith("2012") && l.get(4).equals("elec")) reduce((l1, l2) -> { //Only max value is needed, as they are cumulative per energy meter if (Double.valueOf(l1.get(1)) > Double.valueOf(l2.get(1))) return l1; else return l2; })`
SQL Q3	GridPocket	75.36% (34GB ingested)	`map(s -> { List<String> l = new ArrayList<String>(11); String[] a = s.split(","); for (int i=0; i<11; i++) //GridPocket dataset has 11 columns separated by commas if ((i==0 || i==1 || i==5 || i==7) && i<a.length) l.add(a[i]); else l.add(""); return l; } //We only need 4 columns and rows belonging to years between 2014-2016 filter(l -> l.get(0).startsWith("2014") || l.get(0).startsWith("2015") || l.get(0).startsWith("2016"))`
K-means	UbuntuOne Trace	96.10% (3.9GB ingested)	`filter(s -> s.startsWith("storage_done")) //The rest of trace lines are not necessary map(s -> { List<String> l = new ArrayList<String>(34); String[] a = s.split(","); for (int i=0; i<34; i++) //UbuntuOne dataset has 34 columns separated by commas if ((i==0 || i==19 || i==33) && i<a.length) l.add(a[i]); else l.add(""); return l; } //We want to classify users according to these operations (Put, Get, etc.) filter(s -> s[19].equals("PutContentResponse") || s[19].equals("GetContentResponse") || s[19].equals("MakeResponse") || s[19].equals("Unlink") || s[19].equals("MoveResponse"))`

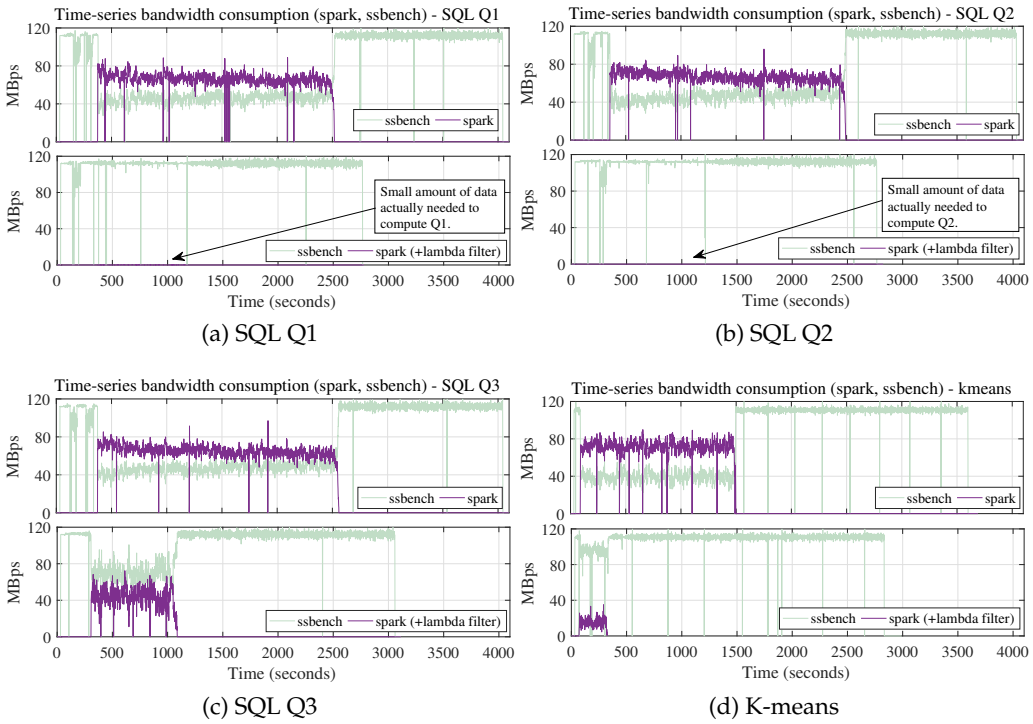
Table 6: Data transfer reduction for the executed queries/jobs and code executed in the Storlet at the storage side.

At first glance, we observe in Table 6 that Crystal filters can discard important amounts of data thanks to the execution of lambda functions at the storage side. That is, from the 3 Spark SQL queries commonly executed by GridPocket analysts, Crystal achieves bandwidth savings ranging from 75% up to 99.96%, depending on the query at hand. Similarly, the k-means batch job executed on the UbuntuOne dataset only ingested 3.9GB out of 100GB. While all analytics in Table 6 require ingesting only a small fraction of the dataset, the distinctive point is that our filter allows administrators to *write and execute lambdas on-the-fly that discard useless data at storage nodes*. For instance, computing Q1 only requires 4 dataset columns (out of 11) and rows belonging to one month of data (out of 10 years captured in the dataset). Assuming that each job is executed by a distinct tenant, the data ingestion required for all the queries/jobs in Table 6 is $\approx$ 38GB with Crystal (instead of 520GB with plain Swift).

A remarkable observation of these experiments is that the analytics application is oblivious to the fact that the data is being filtered. In other words, the submitted application does not require changes in its code depending on whether we execute lambdas at the storage side or not. This is because we can write `map` lambdas that "empty" the content of specific columns while keeping intact the data schema (e.g., number of CSV columns). Also, filtering rows (e.g., `filter`, `reduce`) does not impact the behavior of the application. This observation is important for the deployability and feasibility of such a technique in real analytics environments.

Impact on shared resources under multi-tenancy. We next analyze the benefits of data filtering on a multi-tenant scenario. Fig. 23 shows time series plots of the parallel execution of a `ssbench` workload and Spark competing for the outbound bandwidth, with/without the execution of lambdas at the storage side.

The upper plots in Fig. 23 clearly show the problems of multi-tenancy in a shared object store. We can see that the node executing `ssbench` starts first in all the experiments, thus acquiring all the bandwidth available at the Swift proxy. After a short period of time, `ssbench` bandwidth values start to show fluctuations before Spark starts the ingestion phase. This is because Spark, prior to the actual ingestion of data, executes a series of (`HEAD`) requests to infer the size of the dataset and the data partitions, which impacts on `ssbench`'s requests. Then, Spark starts the ingestion phase with an evident consequence: `ssbench` receives less than half of the bandwidth, given that it has lower parallelism (24 `ssbench` threads vs $3x12$ Spark workers). However, the worst aspect to consider is that most of the bandwidth consumed by Spark is useless data that is discarded by the tasks during



(a) SQL Q1 (b) SQL Q2

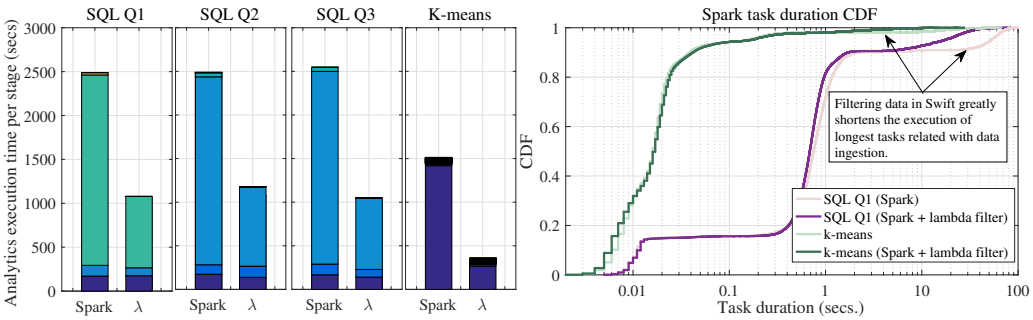
(c) SQL Q3 (d) K-means

Figure 23: Consumption of bandwidth resources at the proxy with/without the execution of our lambda pushdown filter.

the compute phase.

The lower plots in Fig. 23 show a completely different situation. Namely, while the performance interferences on `ssbench` prior Spark ingestion remain, the data ingestion phase of Spark has been greatly reduced. Visibly, `ssbench` can now benefit from almost all bandwidth resources during the experiment, specially for SQL queries Q1 and Q2, as well as for the k-means job. In Fig. 23c, Q3 still exhibits a period of time (812 seconds) in which both tenants compete for the available bandwidth as the achievable data discard for this query is lower. Crystal reduces in 63.24% the period of time in which both tenants compete for bandwidth resources compared to Swift. In all experiments in Fig. 23, `ssbench` also reduces its execution time between 21.5% and 31.8%.

Application completion times. Perhaps, the most direct consequence of reducing data ingestion of analytics in multi-tenant environments is the reduction of completion times. Fig. 24 shows the



(a) Execution times of stages in Spark vs Spark + lambda filter (λ).

(b) CDF of tasks completion times for Q1 and k-means.

Figure 24: Stage execution times of Spark and Spark with lambda filter (λ).

completion times of applications, jobs and tasks obtained thanks to the execution of lambdas at the storage side.

When bandwidth resources are scarce and shared, we can observe that executing lambda functions to discard data at the object store provides important benefits in terms of application completion times. To illustrate this, in Fig. 24a GridPocket queries exhibit a speed-up ranging from 2.12x (Q2) up to 2.43x (Q3), depending on the query at hand. In addition, the k-means job experienced a speed-up of 3.95x thanks to the lambda pushdown. As seen in Fig. 24a, applications with and without lambdas show the same or a very similar number of execution stages (drawn as stacked bars). The main difference is that the longest stages, which are related to data ingestion, are significantly shortened thanks to the data discard at the storage side. In a finer granularity, this is corroborated in Fig. 24b, which shows the duration of Spark tasks in each stage. Fig. 24b shows that in both cases the majority of tasks present a similar duration with the exception of the tail of the distribution. Such small group of longer tasks is related to data ingestion, and they also present a shorter duration due to data filtering at the storage side.

We conclude that Crystal can benefit multi-tenant object stores that serve analytics thanks to the execution of lambdas at the storage side, specifically by reducing data transfers and application completion times.

13.3 Achieving Bandwidth SLOs

Next, we evaluate the effectiveness of our bandwidth differentiation filter (see Section 12.9). We executed a `ssbench` workload (10 concurrent threads) in each of the 3 compute nodes in our cluster, one of each representing an individual tenant. As we study the effects of replication separately (in Fig. 25d we use 3 replicas), the rest of experiments were performed using one replica rings.

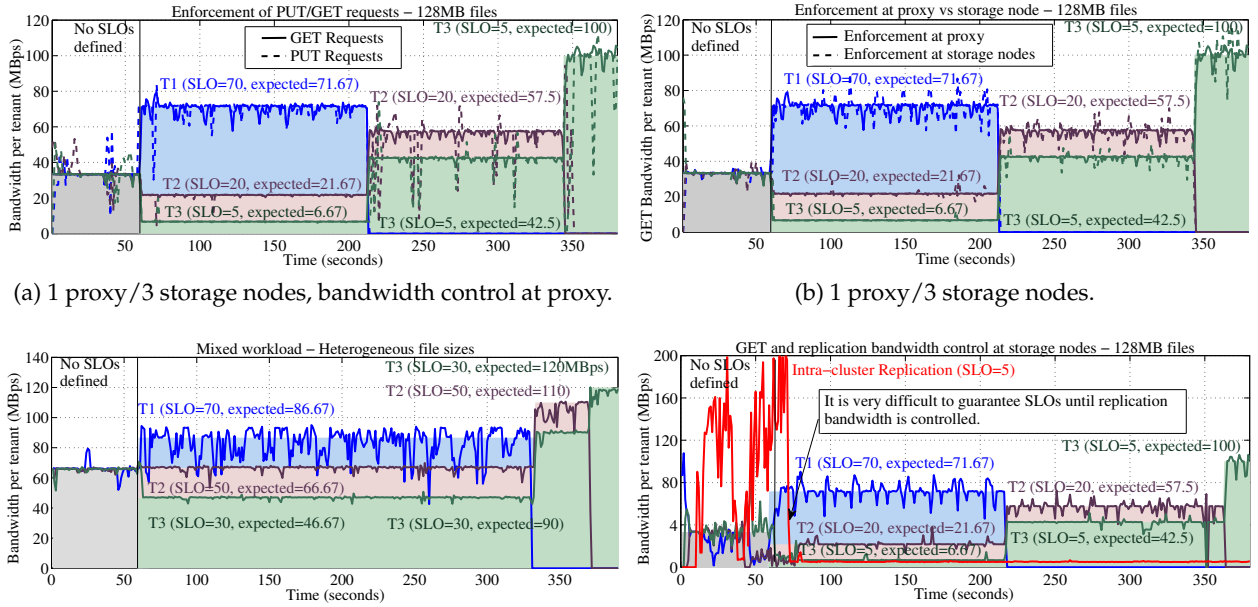
Request types. Fig. 25a plots two different SLO enforcement experiments on three different tenants for `PUT` and `GET` requests, respectively (enforcement at proxy node). Noticeably, the execution of Algorithm 1 exhibits a near exact behavior for both `PUT` and `GET` requests. Moreover, we observe that tenants obtain their SLO plus an equal share of spare bandwidth, according to the expected policy behavior defined by colored areas. This demonstrates the effectiveness of our bandwidth control middleware for intercepting and limiting both requests types. We also observe in Fig. 25a that `PUT` bandwidth exhibits higher variability than `GET` bandwidth. Specifically, after writing 512MB of data, Swift stopped the transfers of tenants for a short interval; we will look for the causes of this in our next development steps.

Impact of enforcement stage. An interesting aspect to study in our framework are the implications of enforcing bandwidth control at either the proxies or storage nodes. Fig. 25b shows the enforcement SLOs on `GET` requests at both stages. At first glance, we observe in Fig. 25b that our framework makes it possible to enforce bandwidth limits at both stages. However, Fig. 25b also illustrates that the enforcement on storage nodes presents higher variability compared to proxy enforcement. This behavior arises from the relationship between the number of nodes to coordinate and the intensity of the workload at hand. That is, given the same workload intensity, fewer nodes (e.g., proxies) offer higher bandwidth stability, as a tenant's requests are virtually a continuous data stream, which is easier to control. Conversely, each storage node receives a smaller fraction of a tenant's requests, since normally storage nodes are more numerous than proxies. This yields that storage nodes have to deal with shorter and discontinuous streams that are harder to control.

But enforcing bandwidth SLOs at storage nodes enables to control background tasks like replication. Thus, we face a trade-off between accuracy and control that may be solved with hybrid enforcement schemes.

Mixed tenant activity, variable file sizes. Next, we execute a mixed read/write workload using files of different sizes; small (8MB to 16MB), medium (32MB to 64MB) and large (128MB to 256MB) files. In addition, to explore the scalability, in this set of experiments we resorted to a cluster configuration that is double the size of the previous one (2 proxies and 6 storage nodes).

Noticeably, Fig. 25c shows that our enforcement controller achieves bandwidth SLOs under mixed workloads. Moreover, the bandwidth differentiation framework works properly when dou-



(a) 1 proxy/3 storage nodes, bandwidth control at proxy.

(b) 1 proxy/3 storage nodes.

(c) 2 proxy/6 storage nodes, bandwidth control at proxies.

(d) 1 proxy/3 storage nodes, bandwidth control at storage nodes.

Figure 25: Performance of the Crystal bandwidth differentiation service (SLOs per tenant are in MBps).

bling the storage cluster size, as the policy provides tenants with the desired SLO plus a fair share of spare bandwidth, especially for `T1` and `T2`. However, Fig. 25c also illustrates that the `PUT` bandwidth provided to `T1` is significantly more variable than for other tenants. This is due to various reasons. First, we already mentioned the increased variability of `PUT` requests, apparently due to write buffering. Second, the bandwidth filter seems to be less precise when limiting streams that require an SLO close to the node/link capacity. Moreover, small files make the workload harder to handle by the controller, as more node assignments updates are potentially needed, especially as the cluster grows. In the future, we plan to continue the exploration and mitigation of these sources of variability.

Controlling background tasks. An advantage of enforcing bandwidth SLOs at storage nodes is that we can also control the bandwidth of background processes via policies. Fig. 25d illustrates the impact of replication tasks on multi-tenant workloads. In Fig. 25d, we observe that during the first 60 seconds of this experiment (i.e., no SLOs defined) tenants are far from having a sustained `GET` bandwidth of $\approx$ 33MBps, meaning that they are significantly affected by the replication process. The reason is that, internally, storage nodes trigger hundreds of point-to-point transfers to write copies of already stored objects to other nodes belonging to the ring. Note that the aggregated replication bandwidth within the cluster reached 221MBps. Furthermore, even though we enforce SLOs from second 60 onwards, the objectives are not achieved —specially for tenants `T2` and `T3`— until replication bandwidth is under control. As soon as we deploy a controller that enforces a hard limit of 5MBps to the aggregated replication bandwidth, the SLOs of tenants are rapidly achieved. We conclude that Crystal has potential as a framework to define fine-grained policies for managing bandwidth allocation in object stores.

13.4 Crystal Overhead

Filter framework latency overheads. A relevant question is the performance costs of our filter framework to the regular operation of the system. Essentially, the filter framework may introduce overhead at i) *contacting the metadata layer*, ii) *intercepting the data stream through a filter*[20] and iii) *managing extended object metadata*. We show this in Fig. 26.

[20]We focus on isolated filter execution, as native execution has no additional interception overhead.

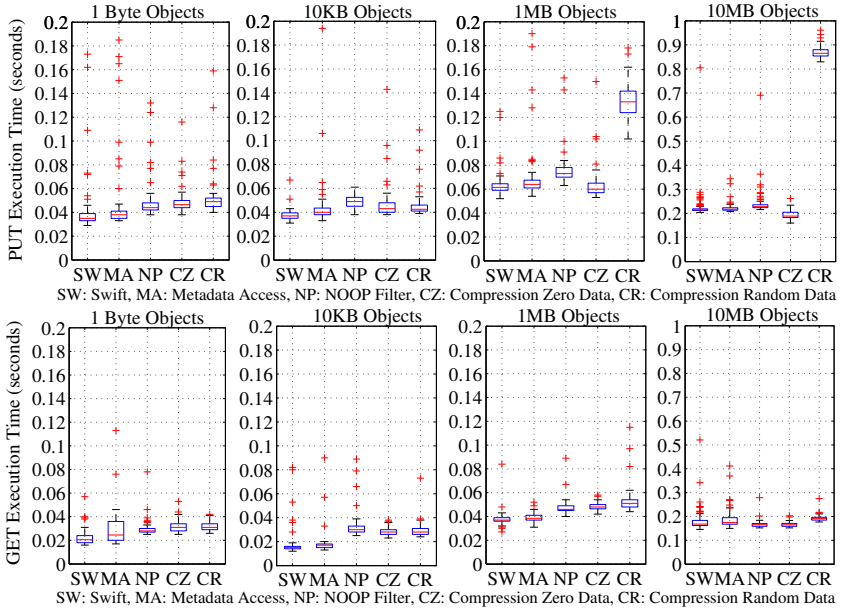


Figure 26: Performance overhead of filter framework metadata interactions and isolated filter enforcement.

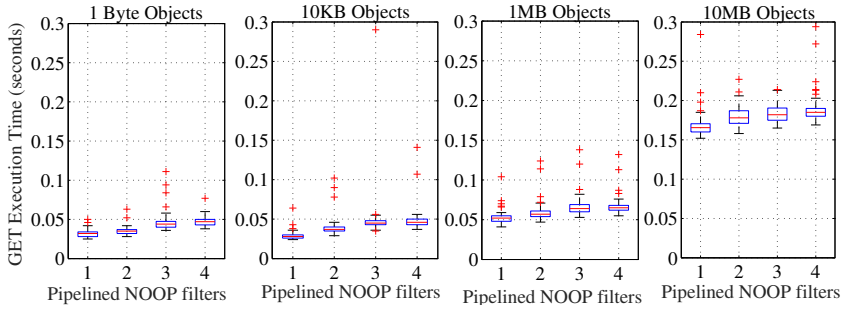


Figure 27: Pipelining performance for isolated filters.

Compared to vanilla Swift (`SW`), Fig. 26 shows that the metadata access of Crystal incurs a median latency penalty between 1.5ms and 3ms (`MA` boxplots). For 1MB objects, this represents a relative median latency overhead of 3.9% for both `GET`s and `PUT`s. Naturally, this overhead becomes slightly higher as the object size decreases, but is still practical (8% to 13% for 10KB objects). This confirms that our filter framework minimizes communication with the metadata layer (i.e., 1 query per request). Moreover, Fig. 26 shows that an in-memory store like Redis fits the metadata workload of Crystal, especially if it is co-located with proxy nodes.

Next, we focus on the isolated interception of object requests via Storlets, which trades off performance for higher security guarantees (see Section 12.5). Fig. 26 illustrates that the median isolated interception overhead of a void filter (`NOOP`) oscillates between 3ms and 11ms (e.g., 5.7% and 15.7% median latency penalty for 10MB and 1MB `PUT`s, respectively). This cost mainly comes from injecting the data stream into a `Docker` container to achieve isolation. We also may consider filter implementation effects, or even the data at hand. Columns `CZ` and `CR` depict the performance of the compression filter for *highly redundant (zeros) and random* data objects. The performance of `PUT` requests changes significantly (e.g., objects $\geq$ 1MB) as compression algorithms exhibit different performance depending on the data contents [75]. Conversely, decompression in `GET` requests is not significantly affected by data contents. Hence, to improve performance, filters should be enforced in the right conditions.

Finally, our filter framework enables managing extended metadata of objects to store a sequence

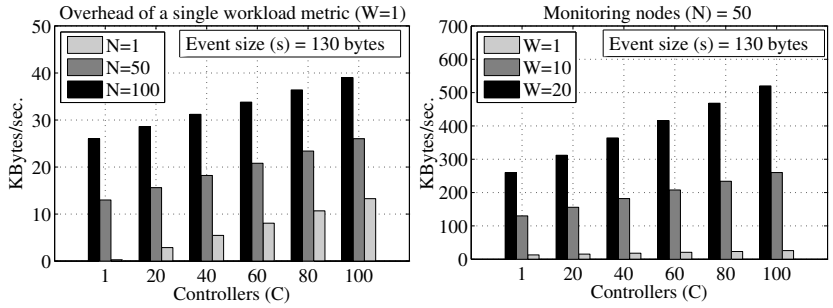


Figure 28: Traffic overhead of Crystal depending on the number of nodes, controllers and workload metrics.

of data transformations to be undone on retrievals (see Section 12.5). We measured that reading/writing extended object metadata takes 0.3ms/2ms, respectively, which constitutes modest overhead.

Filter pipelining throughput. Next, we want to further explore the overhead of isolated filter execution. Specifically, Fig. 27 depicts the latency overhead of pipelining multiple `NOOP` Storlet filters. As pipelining is a new feature of Crystal, it required a separate evaluation.

Fig. 27 shows that the latency costs of intercepting a data stream through a pipeline of isolated filters is acceptable. To inform this argument, each additional filter in the pipeline incurs 3ms to 9ms of extra latency in median. This is slightly lower than passing the stream through the `Docker` container for the first time. This is because pipelining tenant filters is done within the same `Docker` container, so the costs of injecting the stream into the container are present only once. Therefore, our filter framework is a feasible platform to dynamically compose and pipeline several isolated filters.

Monitoring overheads. To understand the monitoring costs of Crystal, we provide a measurement-based estimation of various configurations of monitoring nodes, workload metrics and controllers. Namely, the monitoring traffic overhead O related to $|\mathcal{W}|$ workload metrics is produced by a set of nodes $\mathcal{N}$. Each node in $\mathcal{N}$ periodically sends monitoring events of size s to the MOM broker, which are consumed by $|\mathcal{W}|$ workload metric processes. Then, each workload metric process aggregates the messages of all nodes in $\mathcal{N}$ into a single monitoring message. The aggregated message is then published to a set of subscribed controllers $\mathcal{C}$. Therefore, we can do a worst case estimation of the total generated traffic per monitoring epoch (e.g., 1 second) as: $O = |\mathcal{W}| \cdot [s \cdot (2 \cdot |\mathcal{N}| + |\mathcal{C}|)]$. We also measured simple events (e.g., `PUT_SEC`) to be $s = 130$ bytes in size.

Fig. 28 shows that the estimated monitoring overhead of a single metric is modest; in the worst case, a single workload metric generates less than 40KBps in a 100-machine cluster with $|\mathcal{C}| = 100$ subscribed controllers. Clearly, the dominant factor of traffic generation is the number of workload metrics. However, even for a large number of workload metrics ($|\mathcal{W}| = 20$), the monitoring requirements in a 50-machine cluster do not exceed 520KBps. These overheads seem lower than existing SDS systems with advanced monitoring [60].

Part IV
IOStack Compute/Storage Coordination

The IOStack toolkit is composed of a set of well-defined and standalone building blocks, which can be managed via an integrated dashboard. Moreover, each component has a clean communication interface that enables to modify its behavior in a policy-based manner; for instance, the Crystal API enables an administrator to define new policies and install new filters, whereas the Zoe API allows and administrator to deploy analytics with tailored configurations.

In this part of the document, we explore how an intelligent coordination of these building blocks may benefit the execution of Big Data analytics. In particular, we report our contributions related to the coordination of compute and storage components in IOStack in two ways: On the one hand, Section 14 proposes to build *hyper-controllers to orchestrate the storage service according to the needs of analytics applications*. On the other hand, Section 15 contributes a mechanisms to *automatically migrate computations within analytics applications as storage lambdas*, in order to reduce data ingestion and boost analytics execution times.

14 Hyper-controllers: Making Sense of Application Hints for Storage Customization

14.1 Introduction

In a datacenter, the infrastructure devoted for executing analytics is normally disaggregated, thus consisting of physically separated compute and storage clusters. On the one hand, the compute cluster runs a virtualization layer for analytics engines that compute. On the other hand, the storage cluster runs a storage system that serves data to applications. In this sense, the IOStack toolkit also has a compute building block (Zoe) and an object storage building block (Crystal); they are completely isolated components within the datacenter, as they are agnostic from each other.

While decoupling compute and storage is a desirable property from a software design perspective, the *workloads executed on an analytics infrastructure impact on both Zoe and Crystal*. This represents a twofold problem: i) being oblivious to this fact may be significantly sub-optimal when running analytics at scale, in terms of performance and resource usage; ii) for adapting the system to the heterogeneity of multiple analytics, system administrators should bring upon their shoulders the weight of properly managing and configuring compute and storage components within a datacenter [34, 35]. This yields that, in a complex analytics stack, building a substrate for cross-layer optimizations may be even more important that optimizing compute and storage subsystems independently [36, 37].

Zoe currently does not care about the storage system that its applications are making use of. This may lead, for example, that scheduling policies fail to meet a deadline due to the lack of control of storage resources. Similarly, one can easily infer that different analytics applications may exhibit very different storage usage patterns, which Crystal does not take into account and it makes the problem even harder. In other words, in IOStack compute and storage subsystems are now *totally uncoordinated*, which may prevent us from exploiting a wide variety of "coordinated" or "cross-layer" optimizations across compute and storage building blocks. In this work, we investigate the feasibility of introducing coordination strategies between compute and storage for improving the performance and efficiency of virtualized analytics.

The main contribution of this work is the *hyper-controller abstraction*. A hyper-controller extends the boundaries of the current controllers within the IOStack control plane architecture, by embedding optimization algorithms that take into account multiple building blocks at the same time. A hyper-controller assumes the existence of building blocks for compute and storage that offer i) a public API to control their behavior and ii) monitoring information that can be published to take decisions.

In IOStack, hyper-controllers will leverage cooperation channels across compute and storage subsystems to optimize and make more efficient the execution of analytics in the cloud. In particular, we devise that hyper-controllers have full control and visibility of both storage services and analytics instances; this will enable us to build a platform for designing dynamic algorithms that ingest monitoring information of the entire stack to better orchestrate concurrent workloads.

We also consider the exploitation of "application hints", thus including users that deploy virtualized analytics in the decision loop. In practice, application hints are user-defined meta-data items in a Zoe application that help hyper-controllers to understand the type of application that is going to be executed (batch, interactive, gold/bronze, periodic, etc.). These hints are a valuable source of information to automatically adapt the storage system to the best configuration (e.g., caching, bandwidth, etc.) for a particular application.

In summary, our contributions are the following ones:

- We propose the hyper-controller abstraction, a new control plane entity in IOStack to perform coordinated compute/storage optimizations;
- We develop hyper-controllers that dynamically adapt the storage service, including bandwidth differentiation, caching and prefetching;
- We show the flexibility of hyper-controllers by feeding them with various sources of information to take decisions, including "deployment hints" from users deploying applications;
- We demonstrate through experimentation the advantages of hyper-controllers to optimize complex analytics workloads.

In the following, we describe the role of hyper-controllers within IOStack architecture, as well as our experimental results that demonstrate their advantages.

14.2 Architecture and Design

The concept of hyper-controllers assume the existence of software-defined components to be orchestrated. Therefore, hyper-controllers can be seen as a way of augmenting the control plane of individual software-defined building blocks with behaviors and algorithms that take into account information sources of other building blocks. In the context of IOStack, both Zoe and Crystal are software-defined components: there is a control plane —in Zoe the control plane manages application descriptors and executes scheduling algorithms, and in Crystal the control plane manages storage policies and control algorithms— and a data/compute planes —in Zoe a virtualization layer that assigns resources to applications and monitors them, in Crystal a compute layer to execute storage filters close to the data and monitor storage workloads.

Hyper-controllers are distributed processes, so multiple ones can be deployed at runtime within the system and coexist. The main design point of hyper-controllers is that they embody a new platform for designing optimization within an analytics infrastructure, while keeping compute and storage building blocks isolated. This contrasts with the recently coined concept of hyper-convergence [76], in which commercial systems couple visualization, storage and network services along with specific hardware for optimization.

As can be observed in Fig. 29, hyper-controllers are decoupled from compute and storage building blocks. In other words, both Zoe and Crystal only publish monitoring or context information in the form of events to a communication bus. Note that these systems already have a monitoring system for their own control loop, so the effort required for publishing such information to hyper-controllers is low. The flexibility of hyper-controllers lies in that they can ingest any type of event and communicate with any API. That is, the developer designing a particular hyper-controller is in charge of selecting and parsing the relevant events, as well as interacting with the APIs of software-defined building blocks in an appropriate manner.

According to Fig. 29, the lifecycle of a hyper-controller is as follows: first, a hyper-controller source code is uploaded via the Crystal API. Once uploaded, the code is within the control plane of the system. Then, we can assign one or more queues to that hyper-controller, in order to receive monitoring or context information in the form of events. Then, the system administrator can deploy the hyper-controller, which essentially means to instantiate the uploaded code as a distributed process within the control plane. From this point onward, the hyper-controller will receive information events from Zoe, Crystal or any other information source publishing events in the queues of interest

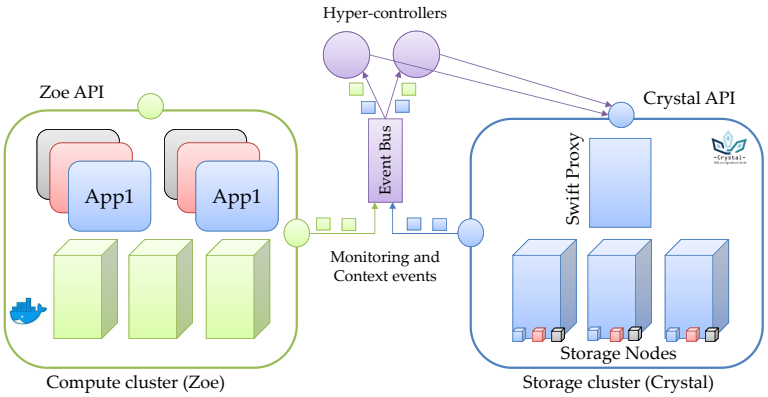


Figure 29: Architecture of IOStack with hyper-controllers orchestrating storage (Crystal) services based on compute (Zoe) and storage monitoring and context events.

of that hyper-controller. Then, depending on the logic implemented within the hyper-controller, it will execute API calls against Zoe and/or Crystal, in order to adapt the behavior of these systems in a coordinated way.

Next, we describe the implementation of two hyper-controllers in order to evaluate their feasibility via real experiments.

14.3 Implementation: Hyper-controllers that Exploit Application Hints

Hyper-controllers extend the base controllers already existing in IOStack; thus, a system developer creates the code for the hyper-controller, extending the base classes provided within IOStack that already implement communication functionality, as well as the main lifecycle stage of the hyper controller (e.g., deploy, kill). Therefore, the system developer only needs to concentrate in "what" the hyper-controller should do, reducing development efforts.

Moreover, in our implementation we make use of the Zoe plugin framework (see D5.3) to extend the Zoe application descriptor functionality with the ability to publish metadata items as events. These events are published at the communication bus where hyper-controllers are subscribed. Users can specify metadata items in a declarative way that are then exploited by hyper-controllers. For example, a user can specify a metadata item such as `priority:gold`, so a hyper-controller may decide to provide more resources to that application compared to others. Similarly, if a user deploys a Zoe application specifying `periodic:true`, a hyper-controller can infer that this application will be repeatedly executed in the future; this may lead to exploit storage techniques like caching to optimize this particular workload.

We present two sample hyper-controllers that we found useful in real situations:

Bandwidth-aware scheduling hyper-controller. Currently, Zoe is not taking into account the IO resources of virtualized analytics sharing a compute infrastructure. While for iterative, CPU-intensive algorithms (clustering, machine learning, etc.) this may be a valid assumption, for data-intensive analytics (e.g., queries, text analysis, etc.) storage becomes an important aspect of scheduling.

To this end, we developed a hyper-controller that allows to differentiate the bandwidth resources that each analytics application ingesting data receives. In our implementation, a user may define an application's priority as `gold`, in order to provide more bandwidth resources to that application than others. The actual amount of bandwidth that each application receives is determined by the hyper-controller algorithm, based on the number of concurrent applications being served, as well as their priority (e.g., `gold`, `silver`, `bronze`).

Small-file optimization hyper-controller. Another aspect related to the storage profile of applications that is not taken into account by Zoe's scheduling is data itself. That is, a data-intensive

analytics application that ingests a certain amount of data will probably exhibit significantly different execution times depending on the size of data objects that are being ingested. That is, data objects of moderate size (e.g., in the order of MBs) are more efficient to transfer than small files (e.g., in the order of KBs), specially due to the relative network overheads and the performance of HDDs.

To solve this problem, we developed a hyper-controller that is able to activate storage filters in Crystal to minimize the impact of small-file requests at the object store. To this end, a user deploying an analytics application may "hint" the hyper-controller with a metadata item such as `small-files:true`. Moreover, the hyper-controller may decide which optimization strategy to execute based on other metadata items. That is, if a user adds the tag `periodic:true` then the hyper-controller is able to optimize the storage side with at least two options, given that the ingestion of a set of small-files will be performed again in the future. The selection of the strategy depends on the size of the data at hand: if the data size fits in a proxy's memory, the hyper-controller applies a caching filter to optimize subsequent data ingestion processes. Otherwise, the hyper-controller executes a prefetching filter, that brings into the proxy's cache files before the analytics application starts its ingestion phase. The partitioning of the cache is decided by the hyper-controller depending on the amount of cacheable workloads executed at a given time.

We also created a section within the IOStack dashboard in order to manage hyper-controllers as well as to connect them with specific event information queues.

14.4 Evaluation

14.4.1 Experimental Setting

Workloads: We used a WordCount Spark application as an example of a data-intensive analytics application. This application gets the number of occurrences per word in a text-based dataset stored in an object storage and writes back the results. Zoe application descriptor is configured to use 2 spark workers with 6 cores and 20GB of RAM each. The application uses Stocator [19] as storage connector to minimize the number of requests sent to object storage.

Datasets: The first dataset used are Wikipedia Backups (57.1GB). The other dataset is composed by 2,031 ebooks from Gutenberg project (809.5MB in total).

Methodology: Spark applications are executed through Zoe command line tool. Stage and application execution time is obtained from Spark logs; bandwidth consumption data for each application is gathered from Crystal monitoring system (Elastic stack).

Platform: In our experimental setting, the object store is a deployment of OpenStack Swift (Ocata version). In the compute side, Zoe uses a Docker Swarm cluster to provision analytics applications, that in this case run on Spark 2.1.0.

We ran our experiments in a cluster formed by Dell PowerEdge nodes with Intel Xeon E5-2403 processors. The Swift cluster has 1 proxy node (28GB RAM, 1TB HDD, 4-core) and 7 storage nodes (16GB RAM, 2x1TB HDD, 4-core). 4 nodes (32GB RAM, 1TB HDD, 24-core) were used as compute nodes to execute Zoe and Spark. Also, there is 1 large node that runs the OpenStack services and the Crystal control plane (i.e., API, controllers, messaging, metadata store). Nodes in the cluster are connected via 1 GbE switched links. Swift cluster-internal replication traffic and client traffic use the same network.

14.4.2 Results

Bandwidth QoS cross-controller. Next, we describe the results obtained from the use of the bandwidth differentiation hyper-controller on two concurrent `wordcount` applications. The objective of this experiment is to demonstrate that a hyper-controller can help Zoe to control the IO resources of applications to help in scheduling tasks. We demonstrate this by executing two `wordcount` applications that share the same amount of CPU and RAM resources in the compute cluster, but giving them different priorities (`gold/bronze`) in the application descriptor. Both applications use the Wikipedia backups datasets.

Fig. 30 shows the difference between executing two `wordcount` applications with and without the hyper-controller. That is, due to the lack of IO control of Zoe resource management framework, two

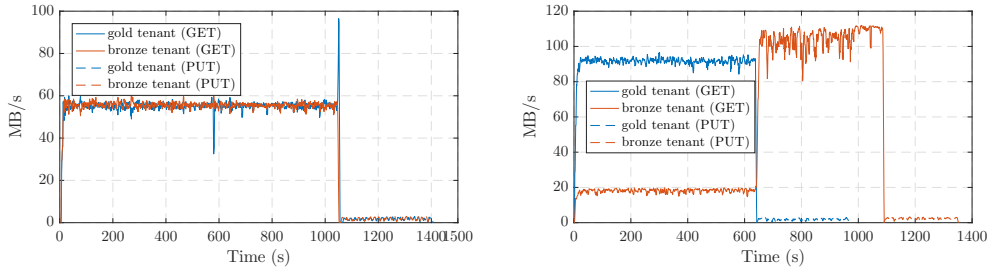


Figure 30: Consumption of bandwidth resources per tenant at the Swift proxy without managing bandwidth resources (left) and enforcing bandwidth differentiation (right).

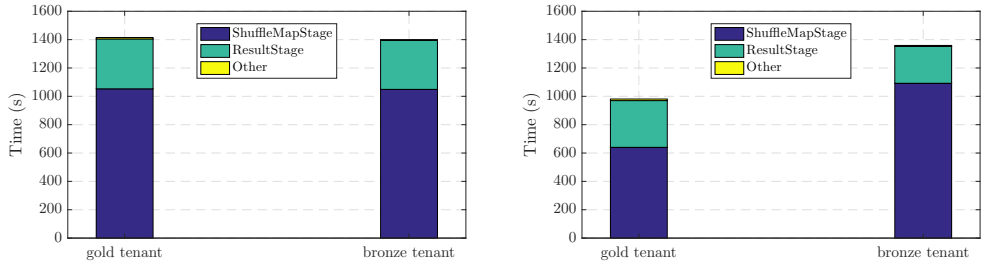


Figure 31: Execution times per stage for both applications without managing bandwidth resources (left) and enforcing bandwidth differentiation (right). Purple areas in stacked bars represent data ingestion stages.

applications with the same CPU/RAM resources but different priorities obtain the same amount of bandwidth. On the other hand, we observe that by adding to the deployment descriptor the `priority` metadata item, the hyper-controller orchestrates Crystal bandwidth differentiation service to provide different shares of bandwidth on both applications. This leads that the `gold` application receives a larger share of bandwidth resources ($\approx$ 95MBps) compared to the `bronze` application.

Fig. 31 shows the impact of managing bandwidth resources applications in terms of execution times. As expected, by default both applications take the same amount of time to execute (Fig. 31, left). This is because the execution time of both applications is directly related to the data ingestion speed (data-intensive analytics). However, thanks to the bandwidth differentiation enforced by the hyper-controller, the execution time of the `gold` application is 29% lower, while the performance for application `bronze` remains similar (actually, it exhibits a 3% shorter execution time as the output phase of the application does not collide with the output writing phase of the other application). The reason why the `gold` application finishes before is simple: a higher bandwidth share allows it to complete all its computation tasks before the `bronze` application. Thus, during the time that `gold` application is being executed, the amount of tasks executed by the `bronze` application is much lower. As soon as the `gold` application finishes, the `bronze` application obtains all the available bandwidth. That is, while the global amount of data ingested is the same in both situations, the hyper-controller gives us the flexibility to decide how to prioritize the execution of applications by controlling bandwidth.

Therefore, we observe that the use of hyper-controllers can augment the resource management capabilities of Zoe by making a coordinated use of Crystal's storage filters, thus making a more accurate scheduling. This may lead also to coordinated scheduling algorithms that are shared across Zoe and hyper-controllers.

Small-file optimization hyper-controller. Next, we describe the results obtained from the use of

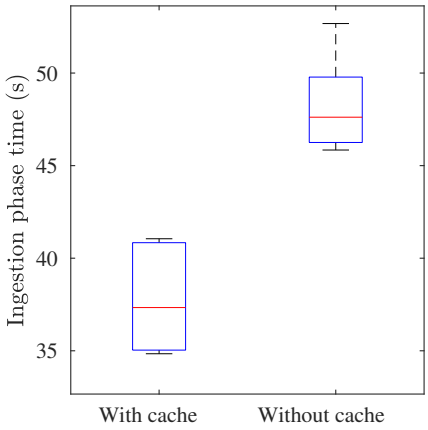


Figure 32: Ingestion stage times for subsequent application executions, with cache enabled in the proxy (left) and without applying cache (right).

the small-file optimization hyper-controller on a `wordcount` application and how user hints in Zoe application descriptors can help to minimize the impact in object store of subsequent data ingestion processes on the same dataset. We demonstrate this by executing the same application with and without user hints. This experiment uses the Gutenberg project Dataset.

Fig. 32 shows the ingestion phase duration with and without the hyper-controller. When the hyper-controller is enabled, it decides to apply a caching filter to optimize subsequent data ingestion processes. All files requested by the first execution of the application are stored in a SSD cache in the proxy server node. The caching filter intercepts the requests of subsequent application executions and returns the cached file instead of forwarding the request to the object servers. The left box plot shows the duration of the ingestion phase for those executions that benefit from the cache (considering a maximum hit ratio when the entire dataset fits in the proxy's cache). On the other hand, we observe that the ingestion phase duration is 22% longer in average without using the hyper-controller, because all requests must reach the object servers to retrieve the requested file, and thus experiment overheads caused by the performance of HDDs and the cluster-internal replication traffic. As we saw in the previous experiment (Fig. 31), the ingestion-related stages account for most of the application execution time in this kind of data-intensive applications.

Therefore, we see that a hyper-controller is able to dynamically optimize some requests depending on the size of data objects and application execution patterns.

15 Automated Migration of Dataflow Computations Close to the Data

15.1 Introduction

In the last years, we have witnessed a growing interest from both the industry and the research community in Big Data analytics. Researchers and practitioners have spent important efforts on improving the operation and limitations of the traditional MapReduce paradigm [7]. As a result, an entire ecosystem of so called *dataflow computing engines* [77] is now commonplace in organizations that perform analytics. Examples of these engines include Apache Spark [6, 78], Flink [79], Apex [80] and Tez [77], among others.

One of the great departures in the design of dataflow engines compared to the traditional MapReduce paradigm lies in their *programming expressiveness*. The MapReduce API requires all user-defined algorithms to be expressed as `map/reduce` primitives, which has been previously noted as too constraining [81, 77]. In contrast, dataflow engines enable richer programming APIs that makes it easier for data scientists to write analytics.

The expressiveness of dataflow computing engines is based on the Directed Acyclic Graph (DAG);

a data structure that represents the data processing workflow of an application. In such a data structure, *a vertex* is a (parallel) data processing step in which user-defined code within API calls —like `filter`, `map` or `reduce`— is executed on a data partition, whereas *edges* represent a data flow between producer and consumer vertexes. The DAG execution model also enables such engines to take fine-grained decisions on the execution of analytics applications, such as "where" computations will take place (e.g., in-memory, disk) and "when" data processing logic will be executed (i.e., the lazy-binding execution [77]).

But a common design aspect that both the MapReduce paradigm and dataflow engines still share is that tasks are scan-oriented [82]. That is, a dataflow system partitions the input dataset into small units of work that are assigned to tasks executing computations defined in the DAG. Inherently, this yields that dataflow engines require to ingest the entire dataset prior to the actual computation, even in the case that only a small fraction of it is actually needed.

While this is suitable when the input dataset is small or the analytics application at hand is CPU-bounded (e.g., machine learning, sorting), other types of analytics built on top of dataflow engines may be potentially data-intensive; to wit, Streaming, SQL or Graph libraries in Spark extend the basic RDD API [83]. This yields that log/text analysis batch jobs, exploratory queries or streaming analytics may blindly scan a dataset before performing any kind of computation on it. This may incur resource waste and performance limitations.

15.1.1 Problem: Getting Stuck with Dataflow Ingestion Stages

To better understand this problem, Fig. 33 shows the execution of two data-intensive applications in Spark: i) An application (`win_stats`) executed by GridPocket[21] (an energy grid company) on semi-structured energy measurements data (140GB) to get statistics on energy values per time-slot, and ii) a `wordcount` job executed on Wikipedia dumps (57GB). In these experiments, Spark ran in 3 compute nodes, whereas the data was stored in a 8-node OpenStack Swift cluster (1 proxy). Nodes were connected via 1Gbit switched links.

Fig. 33a shows in stacked bars the DAG execution stages of analytics applications. Visibly, in all these applications the ingestion-related stages (gray color) accounted for 87.6% to 60.5% of total execution time for `wordcount` and `win_stats`, respectively. At a finer granularity, Fig. 33b illustrates that the tasks involved in data ingestion are in turn the most time consuming, even being a minority. The worst side is that most of the ingested data in Fig. 33 is *useless for Spark and it is discarded later on*. From a cloud pricing viewpoint, this is important: to wit, Amazon Athena charges customers only for the data transferred from S3 per query [84].

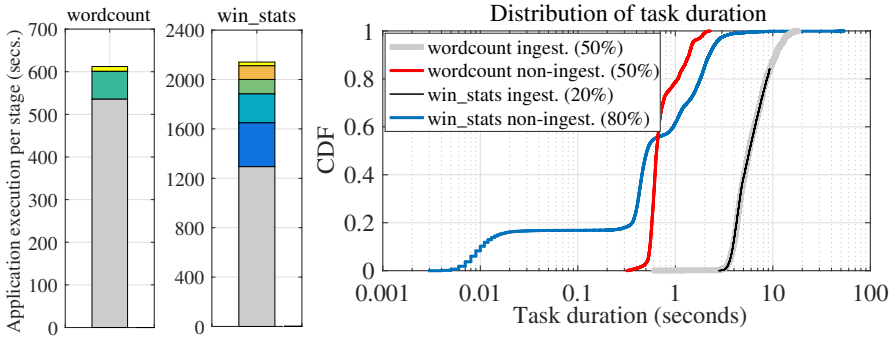
Besides, given the compute/storage disaggregation in many analytics infrastructures, multi-tenant workloads of *data-intensive analytics can induce high consumption of bandwidth resources*. This may contribute to saturate scarce or oversubscribed datacenter inter-cluster networks [85].

In this sense, the "data ingestion problem" has gained traction recently. Some efforts augmented Hadoop/Spark with indexes on structured storage (HBase) for pruning useless data partitions [82, 86]. Others proposed to execute periodic analytics in-situ during data acquisition, so further data ingestion processes are avoided [87]. Closer to the spirit of this work, authors in [85] built a data filtering service at the storage side to reduce data ingestion from object stores in Hadoop. However, we believe that the expressiveness and stream-oriented nature of dataflow primitives calls for *enacting the storage side as active entity in the DAG execution*.

15.1.2 Contributions

In this work, we identify that many operations defined in the execution DAG of dataflow engine application (`filter`, `map`, `reduce`, etc.) can be efficiently executed as synchronous lambda functions on data streams at the storage side to improve performance and reduce bandwidth usage. The research challenge we tackle is to build a framework that exploits this core idea, while keeping both the dataflow engine and the storage system oblivious to this process. In particular, we focus on unstructured data storage like object stores, which do not account with native indexing mechanisms to

[21] `http://www.gridpocket.com`



(a) Application stages (b) Task execution times

Figure 33: Experiments that show the performance bottleneck that data ingestion related stages represent in several analytics applications.

discard data.

We address these challenges with λFlow, a framework for migrating general-purpose dataflow computations as lambda functions on object data streams. First, λFlow allows to plug-in *job analyzers*, which encapsulate the logic for deciding which dataflow computations of an input job are safe to migrate to the storage side. Second, λFlow proposes a *programmable storage layer* that intercepts storage requests and dynamically compiles and executes DAG computations as lambda functions on data streams. Moreover, λFlow offers a *policy-based scheduler* that allows administrators to decide whether to migrate a set of DAG computations to the storage based on migration metrics (e.g., data filtering ratio, lambda execution cost at the storage).

In summary, our key contributions are:

- We propose to migrate parts of a dataflow analytics application's DAG to the storage for reducing execution times and ingestion-related transfers;
- We design and implement λFlow, a framework for migrating dataflow API calls as lambda functions on object data streams. In our prototype, both Spark and OpenStack Swift are oblivious to the migration process.
- We deployed λFlow in a 14-machine cluster and we demonstrated its potential by executing standard data-intensive analytics in well known benchmarks and jobs of an energy grid use-case company (GridPocket).

Our results in a 12-machine cluster demonstrate that λFlow significantly reduces data ingestion for data-intensive analytics (i.e., data transfer reduction $> 90\%$), and it achieves higher analytics execution performance (i.e., application speedups from $1.35x$ to $3.98x$). Moreover, the flexibility of λFlow migration policies facilitate its deployment in multiple scenarios.

15.2 Background and Motivation

Next, we introduce the basic concepts behind dataflow computing and data-driven lambdas in object stores. Then, we present our rationale behind λFlow: automatically combining both computing models to optimize data-intensive analytics.

15.2.1 Dataflow Computing: Spark as an Example

Dataflow computing engines —e.g., Apache Spark [78], Flink [79], Apex [80] and Tez [77]— offer simple programming APIs that let programmers manipulate distributed collections of objects across a cluster through operations. In Spark, such collections called Resilient Distributed Datasets (RDDs) [6] reside in memory to optimize iterative computations on large clusters. The programming API of

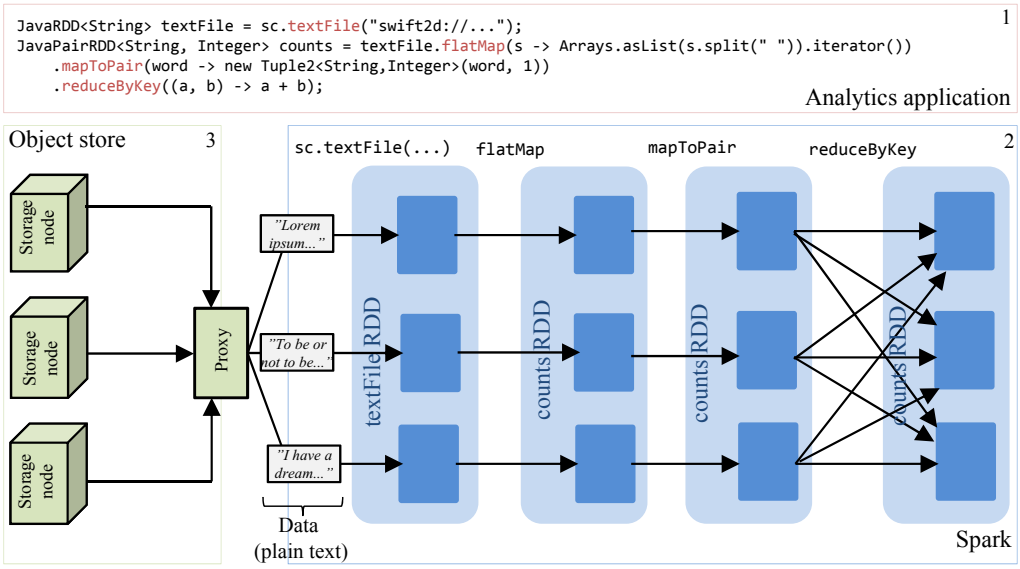


Figure 34: Example of (1) an input `wordcount` application, (2) the DAG execution of Spark and (3) the data ingestion process from an object store.

Spark RDD allows programmers to execute operations on datasets through i) *transformations*, which create a new dataset from an existing one (e.g., `map`, `filter`, `reduceByKey`, etc.); and ii) *actions* that return a value to the driver program as a result of a computation on the dataset (e.g., `reduce`, `first`, etc.).

To better understand the dataflow programming model, Fig. 34 depicts the code and execution of a `wordcount` Spark application (Java). The program first declares a RDD that contains the lines of a set of text data objects stored remotely (`textFile`). Then, a second RDD (`counts`) is aimed to actually contain the counts of each word in the dataset by applying a set of transformations on `textFile`. That is, we first split each line into words (`flatMap`), and then we `map` each word into a `(word,1)` pairs that are further aggregated via a `reduceByKey` transformation. As visible in the application DAG of Fig. 34, all these operations are executed in parallel on data partitions.

The broad adoption of such programming model across several dataflow computing engines has motivated the appearance of systems like Apache Beam [88], which allows users to write analytics applications and choose among back-ends to execute them (e.g., Spark, Flink, Apex). We believe that this outlines an opportunity: *improving data ingestion for this "programming model" may benefit several dataflow engines*.

15.2.2 Towards Data-driven Lambdas in Object Stores

Due to the relevance of unstructured data in analytics, object storage is becoming a pervasive substrate for big data storage [89, 90]. A popular example is OpenStack Swift, which is a highly scalable object storage system that can store a large amount of data through a RESTful HTTP APIs (see Fig. 34). It provides a simple API to store (`PUT`), retrieve (`GET`), and delete (`DELETE`) objects. To achieve high scalability, Swift exploits the synergy between a flat object ID space and consistent hashing via a hash-based data structure called ring. Internally, Swift exhibits a two-tier architecture that consists of *proxy and object servers*. The former are in charge of authentication and access control of storage requests. Upon reception of a valid request, a proxy server routes it to the corresponding object servers for storage and replication.

But even more interestingly, the rise of interest in large scale computing has also reached object storage services. That is, the high-level semantics of object storage makes it an ideal substrate to execute *stateless computations or lambdas on objects*. Services like AWS Lambda [91], IBM OpenWisk [92] or Google Cloud Functions [93] are pioneering in the provisioning of such services, and the research

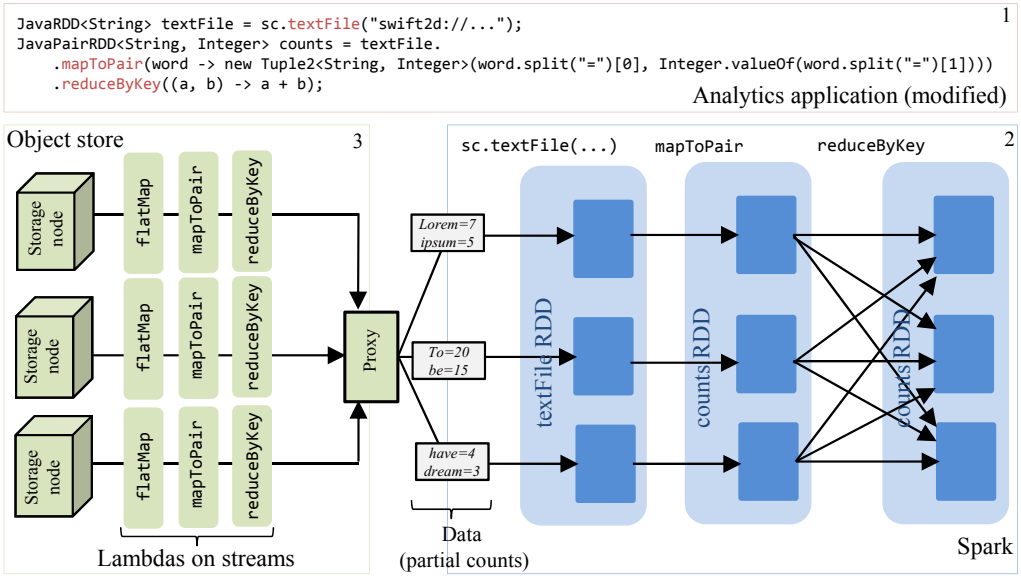


Figure 35: Rationale behind λFlow: (1) To modify the input application, and (2) migrate some dataflow computations as lambdas in object streams.

community is recently focusing on this model due to its programming simplicity and data-parallel nature [94, 95]. While most of these systems execute computations asynchronously (event-driven), recent approaches also provide *synchronous execution of computations of data flows* (data-driven) [21, 95]. In fact, synchronous lambda processing can be seen just as a simplification of the dataflow model to execute parallel computations on data partitions (objects).

15.2.3 Motivation: Getting the Best of Both Worlds

Our motivation is *to blend both dataflow computing and lambda processing models as a unified processing engine*.

To better illustrate this, let us describe the example. In Fig. 35, we observe that the input application is different from the original one (see Fig. 34). That is, Spark now assumes that the data is a set of `(word,count)` pairs instead of plain text, which are aggregated later on (`reduceByKey`). To this end, storage nodes execute a pipeline of lambda functions per data object according to a set of dataflow operations defined in the input application (`flatMap`, `mapToPair` and `reduceByKey`). In this example, storage nodes are synchronously transferring word counts of each data object that are then summed up by Spark. Such a mixture of lambda processing and dataflow computing has two main benefits: i) the application DAG in Spark is *simplified and consumes less resources* by amortizing spare computing power at the storage side; ii) and even more importantly, *data ingestion is heavily reduced* (with this approach, a `wordcount` on Wikipedia dumps reduces data ingestion in ≈ 95%).

Naturally, getting the best from lambda and dataflow processing involves challenges; for instance, it requires a framework capable of identifying dataflow computations to be safely migrated as well as a flexible execution engine at the storage side, among others. In what follows, we present the design and architecture of λFlow to realize this vision.

15.3 λFlow Design and Architecture

λFlow is a framework for migrating general-purpose dataflow computations as lambda functions on object data streams. The core goal of λFlow is to allow developers to benefit from the joint power of dataflow computing and lambda processing at the storage to boost data-intensive analytics. To this end, λFlow is designed to augment existing cloud analytics infrastructures; the object store, the dataflow engine and the user may remain oblivious to the operation of λFlow, which is key for deployability. Moreover, the design of λFlow is aimed at benefiting multiple dataflow engines (not a

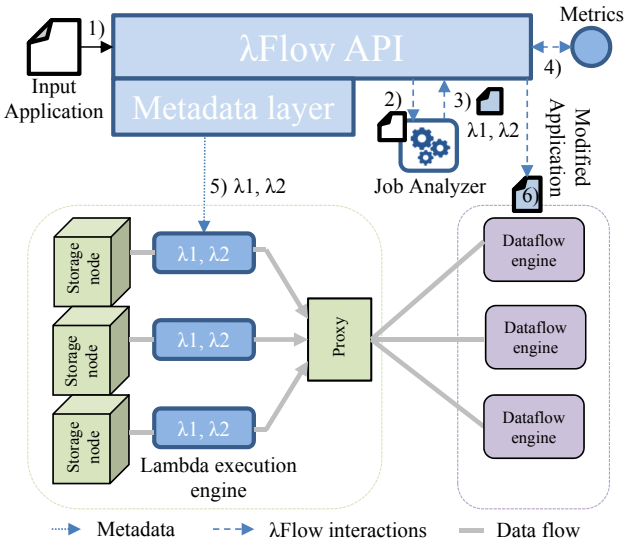


Figure 36: Architecture and lifecycle of λFlow.

specific one).

To achieve such design goals, λFlow decouples control/data planes (similarly to software-defined storage systems [25, 16]) and provides the following extension points:

- **Job analyzer**: A job analyzer receives as input an analytics application and has two main tasks: i) to identify operations within the code that are suitable for execution at the storage side in a stream fashion, and ii) modify the input application accordingly (if necessary);
- **Lambda executor**: Lambda executors are intended to receive lambda functions from the control plane and execute them on data streams of objects.
- **Workload metric**: Metrics expose to the control plane monitoring information related to either the execution of analytics or storage system resources;
- **Migration controller**: Administrators can decide whether to execute or not lambdas at the storage side based on policies that are related to the available metrics;

The architecture of λFlow is depicted in Fig. 36. At the control plane, we can find the API that exposes the available λFlow operations to administrators in a simplified manner (see Table 7). The λFlow API has 4 main groups of calls, so administrators can manage the extension points of the framework. Besides, the API manages information from the metadata layer; for instance, it stores the information about defined migration policies on specific data containers or the existing metrics and their location.

Upon the submission of an analytics application (Fig. 36, step 1), the API resorts to an executor facade that triggers the *job analyzer* defined by the administrator. Developers contributing a new job analyzer must fulfill a simple contract with the executor (Fig. 36, step 2 and 3): i) a job analyzer should expect as input the `original code` of the analytics application, and ii) it should return a tuple `<modified code, [lambdas]>`. Thus, job analyzers can be seen as pluggable pieces of logic for developing techniques like static analysis on input applications of dataflow engines [85].

Administrators may also manage *metrics* within the system. Metrics are micro-services [53, 55] that get monitoring information from a target source —e.g., storage resources events, analytics execution files— and expose them to the API via a simple contract (i.e., `get_value`). Once a metric is deployed, an administrator can write *migration policies* to decide whether or not to perform the migration of lambdas (Fig. 36, step 4). Policies are in the form of simple rules like `free_bw > 50MBps`

API Calls	Description
[add,delete]_analyzer	Management calls for job analyzer binaries.
[attach,remove]_analyzer	Assign a job analyzer to a tenant/container.
[deploy,undeploy]_metric	Management calls for metric micro-services.
[create,delete]_policy	Definition of migration policies.
[deploy,undeploy]_executor	Management of lambda executors at the storage side.

Table 7: Main API calls of λFlow.

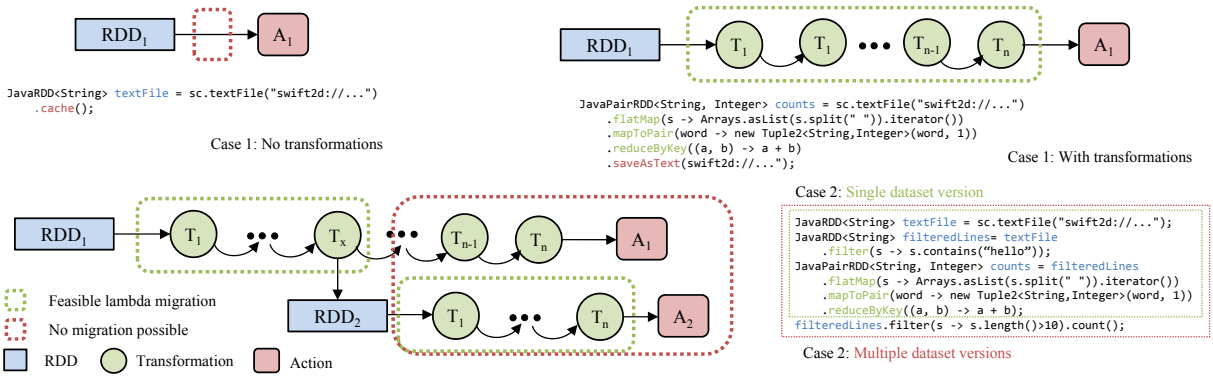


Figure 37: Flow control graph built by our job analyzer from an application to infer which transformations on RDDs can be safely migrated to the storage.

or `data_filtering > 0.8`, where `free_bw` and `data_filtering` are aliases for metric micro-services. This mechanism provides enough flexibility to the framework for handling situations where the compute power at the storage side is limited or when migrating lambdas is not beneficial due to the data contents, among others.

Following the framework lifecycle, if a policy does exist and its condition is not met, the workflow proceeds by submitting the original application code; otherwise, the system stores the lambdas at the metadata store and submits the modified application code (Fig. 36, step 5 and 6).

At the data plane, λFlow offers a stream processing engine within the object store that has one-to-one[22] relationship with storage nodes to exploit their parallel compute and/or network resources. The stream processing engine allows us to intercepting data streams from requests and inject them into a lambda executors. Lambda executors are contributed by developers and receive as input i) the input/output streams of a data object requests, and ii) a set of `<order=type/body>` tuples describing the signature and body code of the lambdas to execute and their execution order. Thus, the responsibility of the developer is to correctly apply the input lambdas on the data stream. Note that λFlow allows several lambda executors to be deployed at the storage side; an administrator defines which one should be triggered on requests of a tenant/container.

15.4 Implementation and Use-Cases

The implementation of λFlow has been carried out as an extension of Crystal [16, 96], a software-defined storage framework for object stores. We extended the Crystal API for managing job analyzers on tenants and containers that are used in analytics workloads. Moreover, we created a new micro-service template that embeds the communication logic with the API; to create a new metric, developers should only extend this micro-service and focus on gathering the data of interest. As a side effect, we also developed a PyActor [97], a minimalist actor middleware, in order to improve the communication between metrics and the control plane.

At the storage side, we resorted to OpenStack Storlets framework [30] —a former IBM project— that provides OpenStack Swift with the capability to run computations on data streams at storage nodes in a secure and isolated manner making use of Docker containers [59]. Invoking a Storlet on

[22] "One-to-one" means that the execution engine should be placed between storage nodes and proxys, but does not necessary yields co-location.

a data object is done in an isolated manner so that the data accessible by the computation is only the object's data and its user metadata. A Docker container only runs filters of a single tenant for higher security. We exploit the raw stream processing capabilities of Storlets and its supported runtimes (Java, Python) to develop lambda executors. Moreover, we make use of interception middleware (WSGI) i) to contact the metadata layer (e.g., once per request to get the lambdas to execute, pick the correct lambda executor), ii) and to inject data streams into the Storlets framework, so Swift remains oblivious to the execution of stream computations.

15.4.1 Job Analyzer for Java Applications: Spark

A developer contributing a job analyzer should take into account a subtle but important nuance: *there is a mismatch between in-memory (dataflow) and stream processing (lambdas)*. That is, whereas a Spark RDD is persistent and may be subject to several transformations (even jointly with other RDDs), lambdas at the storage side can be only executed once during data ingestion. Anyway, our job analyzer implementation[23] shows that even though this mismatch, both processing models can be combined as a practical optimization technique.

First, our job analyzer resorts to well-known control flow analysis in order to build a tractable data structure of the input application code. In particular, we build a graph (`FlowControlGraph`) per RDD that directly points to a data container or that is derived from a RDD that does. The root node of such graph corresponds to the RDD name and type, whereas the rest of nodes represent transformations and actions executed on it. We discard declared RDDs that are unrelated with the external data containers, as they cannot benefit from storage-side optimizations.

Then, the analyzer builds each `FlowControlGraph` node by extracting the signature, type and body of the operations invoked on its RDD; this information is necessary for compiling lambdas at the storage side. Also, a non-derived RRD has a pointer to the root node of a derived RDD at the operation node from which it was derived (see Fig 37, case 2). As shown later on, this helps us to describe data dependencies across RDDs and identify lambdas that can be safely migrated.

The next phase for the job analyzer to execute is the translation phase. That is, our implementation exploits the great synergy between the Spark RDD API and the Java 8 Stream API, which is the platform used to executed lambda functions at the storage side (see Section 15.4.2). For many RDD operations that can be executed in a stream fashion, there is a direct equivalent in the Stream API (e.g., `map`, `filter`, `distinct`, etc.); for those operations without a direct equivalent operation, our job analyzer contains an extensible set of translation rules that can encapsulate the translation logic (e.g., `reduceByKey` can be achieved using the `Collector` API). Thus, during the translation phase such rules are applied on each node of the available `FlowControlGraph`, so RDD operations become Stream equivalents for the lambda executor.

Subsequently, we infer which lambdas are safe to migrate at the storage side. According to Fig. 37, we may face two main situations: i) a single RDD invokes a (possibly empty) set of transformations and an action; ii) from a single RDD, one or several other RDDs are derived, and each one will in turn fall into one of these two cases. In former case, all RDD transformations that are executable in a stream fashion can be migrated to the storage. However, the migration possibilities for the latter case depends on whether the application works on a single or multiple versions of the original dataset.

To better understand this, let us describe the examples in case 2 (Fig. 37). In the first version of the code (green dotted box), there are 3 RDDs (`textFile`, `filteredLines` and `counts`) that are derived from each other after applying some transformations and reside in memory. But most importantly, although all these RDDs are persistent the result of the application could be also expressed as a set of transformations on the original RDD. Thus, this is just another example of case 1, so transformations can be migrated as lambdas as the storage side. Conversely, if we consider the last line of the code (red dotted box), we realize that applying all the lambdas in `counts` will alter the result of `filteredLines`. That is, to obtain the same result with and without migrating lambdas, we only can execute `filter` at the storage side.

Finally, the job analyzer removes migrated operations from the original Spark code. Moreover, in

[23] `https://github.com/Crystal-SDS/spark-java-job-analyzer`

Application	Dataset	Description	λFlow optimization opportunities
Countwords (`countwords`)	Wikipedia Backups (57.1GB) [98]	Count the number of words of a text-based dataset.	Compute the number of words per object instead of transferring text.
Wordcount (`wordcount`)	Wikipedia Backups (57.1GB) [98]	Get the number of occurrences per word in a text-based dataset.	Compute `(word,count)` pairs for each data object instead of transferring text.
Exploratory query (`query`)	GridPocket IoT Data (138.3GB) [99]	Get the top 10 meters from Paris that consumed most energy in 2012.	Filter the rows that do not belong to 2012/Paris, build `(meter,energy)` tuples and select max values.
Windowed statistics (`win_stats`)	GridPocket IoT Data (138.3GB) [99]	For records within a given period (e.g., 30 mins) in time-series data, get (avg, min, max) triples.	Compute tuples with maximum meter energy measurement per time-slot at the storage side.
Joint log failure correlation (`log_corr`)	NASA Apache logs (95.6GB) [100]	Compute per-hour time-series correlation for bad HTTP requests ($40X/50X$ codes) across two log containers.	Filter non-error lines from logs and compute partial time-series with errors per time-slot.
DNA sequence similarity (`genomics`)	1000 Genome project FastQ DNA (177.5GB) []	Compute cosine similarities across DNA sequences containing the pattern `GATTACA`.	Select only sequences containing `GATTACA` and transform them into numeric vectors.
K-means (`kmeans`)	UbuntuOne Trace (97.7GB) [43]	Classify UbuntuOne users according to their types of storage operations (k=5, iterations=20).	Discard useless data dimensions/trace lines, and perform partial counts of user operations per object.

Table 8: Analytics applications used in our evaluation.

some cases the input data does not match with the expected type in the original application (e.g., the modified `wordcount` receives `(word, count)` pairs from the storage as strings). As with translation rules, our job analyzer has an extensible set of migration rules that developers may use to introduce the necessary logic to map the input data to the appropriate type according to the first transformation executed in Spark.

15.4.2 Lambda Executor Exploiting Java Stream API

We implemented a lambda executor[24] as a Storlet that allows to execute (Java 8) lambdas passed by parameter on object data streams. It gets as input from the metadata layer a set of `<order, type/body>` pairs describing the lambda functions to apply on a data stream and their execution order. To this end, the lambda executor compiles on runtime the input code for the lambda input pairs. In the case of a successful compilation, the lambda executor encodes the byte-level stream into a Java 8 Stream and applies the lambda(s) on each record. Naturally, in the case the lambda compilation raises an error, the data object is retrieved to the user as regular.

Moreover, our lambda executor contains two optimizations for minimizing overheads: first, the encoding overhead related to transform byte-level data streams into text is only performed when there are lambda functions to execute. Second, we implemented a cache of lambda functions, so a lambda requires to be compiled only the first time it reaches the filter; subsequent executions of a given lambda function reuse the compiled function object stored at the cache. Overall, we confirmed that the compilation overhead of simple lambdas ranges between 3ms-9ms in most cases. We consider this overhead to be affordable, specially taking into account the data transfer and job execution time gains reported in Section 15.5.

15.4.3 Deploying Migration Controllers

To demonstrate our framework, we deployed several metrics that enable administrators to decide when to migrate lambdas to the storage upon the submission of an analytics application:

Available bandwidth: It is plausible to consider that the compute power at the storage side could be limited, as it cannot be provisioned on demand [101] (e.g., active storage). For this reason, an administrator may want to migrate lambdas to the storage only when there is no available bandwidth when submitting a new application (e.g., `free_bw < 80`MBps).

Variable price of stream computations: The prices of cloud computing services may vary with time (e.g., AWS Spot Instances), for which migrating lambdas to the storage may exhibit variable costs. Thus, we created a metric that periodically tracks the price of a defined computing service and de-

[24] `https://github.com/Crystal-SDS/filter-samples`

cides whether to execute lambdas on objects based on a threshold (e.g., `instance_price < 0.4$/hr.`).

15.4.4 Applicability and Current Limitations

As for now, our lambda executor is limited to the runtime compilation of object types available in the JDK 8 (e.g., primitives, String, collections, etc.) or any other library linked during the lambda executor packaging. However, we do not dynamically handle cases in which lambdas operate on user-defined objects, as this would require not only to migrate and compile lambdas, but also classes as well. Moreover, our job analyzer does not yet support code expansion; that is, the body of the lambda to be migrated should contain the actual logic, instead of a function reference.

We also defer for future work the study of our framework with RDDs that are shared in-memory across several applications, as systems like Apache Ignite recently offer [102].

In any case, with our current implementation we successfully tested 30 different analytics applications from real use-cases and benchmarks. Note some of these limitations are in the current development roadmap of the project.

15.5 Validation

Next, we evaluate a prototype of λFlow with an OpenStack Swift deployment and Spark.

15.5.1 Experimental Setting

Objectives: Our evaluation addresses the following objectives: i) Demonstrate the end-to-end lambda migration capabilities of λFlow in various analytics applications; ii) Understand the potential benefits of λFlow in terms of data transfer reduction and resource usage; iii) Verify the flexibility of our policy-driven migration policies; iv) Show the applicability of λFlow in both public and private analytics infrastructures.

Analytics: Table 8 summarizes the analytics applications used in our evaluation[25]. Applications in Table 8 are real-world use-cases for which the λFlow transparently inferred the computations to be executed at the storage side, as well as the modifications required in the input code.

Table 8 also depicts the public datasets used in our evaluation. These datasets are composed by sets of objects; to explore the impact of object sizes, we test data objects from 20MB (GridPocket) up to 196MB (Apache logs), and even heterogeneous object sizes within a dataset (DNA sequences). In some cases (e.g., NASA Apache logs), we replicated the original dataset to obtain more significant data volumes[26].

Platform: In our experimental setting, the object store is a deployment of OpenStack Ocata version with OpenStack Storlets [30] as a computing platform for our lambda executor, whereas we ran analytics applications on Spark 2.1.1. We also make use of the Spark listener system to obtain fine-grained monitoring information of an application lifecycle [103].

We ran our experiments in a 12-machine cluster formed by 8 Dell PowerEdge 320 nodes (Intel Xeon E5-2403 processors); 1 of them acts as Swift proxy nodes (28GB RAM, 1TB HDD, 4-core) and the rest are Swift storage nodes (16GB RAM, 2x1TB HDD, 4-core). There are 3 Dell PowerEdge 420 (32GB RAM, 1TB HDD, 24-core) nodes that were used as compute nodes to execute Spark. Also, there is 1 large node that runs the OpenStack services and the Crystal control plane (i.e., API, controllers, messaging, metadata store). Nodes in the cluster are connected via 1 GbE switched links.

We ran our experiments in a cluster formed by Dell PowerEdge nodes with Intel Xeon E5-2403 processors. The Swift cluster has 1 proxy node (28GB RAM, 1TB HDD, 4-core) and 7 storage nodes (16GB RAM, 2x1TB HDD, 4-core). 3 nodes (32GB RAM, 1TB HDD, 24-core) were used as compute nodes to execute Spark. Also, there is 1 large node that runs the OpenStack services and the Crystal control plane (i.e., API, controllers, messaging, metadata store). Nodes in the cluster are connected via 1 GbE switched links.

Application	Lambda pipeline migrated to Swift	Transfer reduction	Data ingested
`countwords`	map → reduce	99.978%	12MB
`wordcount`	flatMap → filter → map → collect	95.096%	2.8GB
`query`	filter → map → filter → map → collect	99.997%	4.2MB
`win_stats`	filter → map → collect	89.877%	14GB
`log_corr`	map → filter → map → map → map → collect (x2)	99.940%	57MB (x2)
`genomics`	filter → filter → map	97.697%	4.1GB
`kmeans`	filter → map → filter → map → collect	99.916%	82MB

Table 9: Data transfer reduction per application achieved by λFlow.

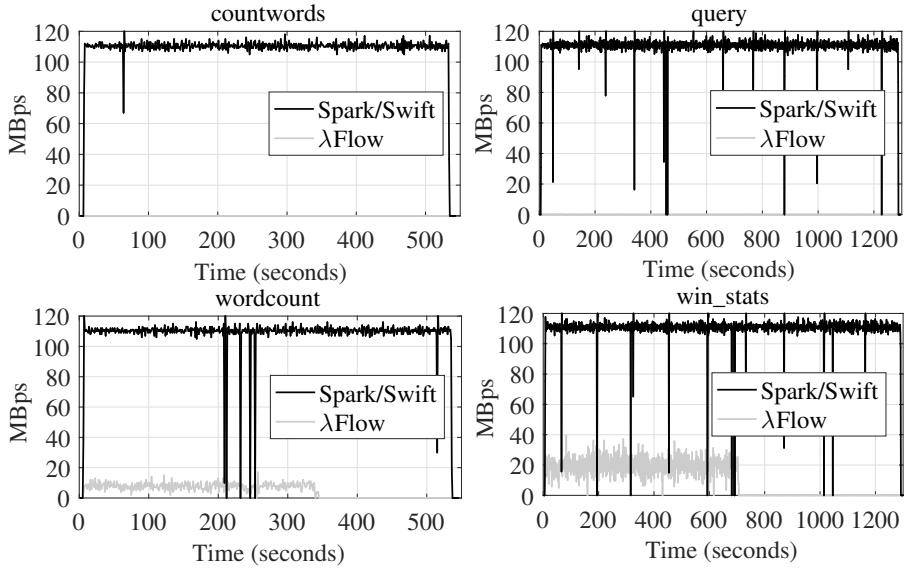


Figure 38: Time-series view of bandwidth utilization during data ingestion of vanilla Spark/Swift and λFlow for several applications.

15.5.2 Results

We compare the operation of vanilla Spark/Swift against these systems augmented with λFlow in terms of data transfer reduction, application execution times, including multi-tenant scenarios. We also demonstrate the benefits of migration policies under different situations.

Data transfer reduction. Next, we show how λFlow improves bandwidth usage in analytics infrastructures. Table 9 depicts the pipeline of lambdas per application that our job analyzer (Section 15.4.1) migrated to the storage and the data transfer reduction of executing such lambdas on requests.

Table 9 demonstrates that augmenting dataflow computing engines with synchronous lambdas at the storage side is a practical approach to reduce data ingestion for data-intensive analytics. That is, all the applications in our evaluation experience a data ingestion reduction $\geq$ 90%, considering either one or multiple containers. This yields that the storage cluster may deliver the resulting spare bandwidth to other tenants.

Interestingly, the applications in Table 9 are heterogeneous in terms of data transfer reduction. While applications like `countwords` or `query` only require a very small amount of data from the whole dataset (order of MBs), other applications require from all the requests an output from the lambda pipeline of a non-negligible size, leading to a larger data ingestion. This can be clearly noticed in Fig. 38, which illustrates time series plots of bandwidth consumption of several applications. All in all, we observe that λFlow greatly reduces data ingestion transfers compared to vanilla Spark/Swift.

In Table 9, we also observe that our λFlow prototype allows to configure complex lambda pipelines

[25] The analytics used in Table 8 are available at `https://github.com/Crystal-SDS/spark-java-job-analyzer/tree/master/src/test/resources/test_jobs`

[26] Note that augmenting a dataset does not benefit λFlow as it can filter data within object but not across data objects.

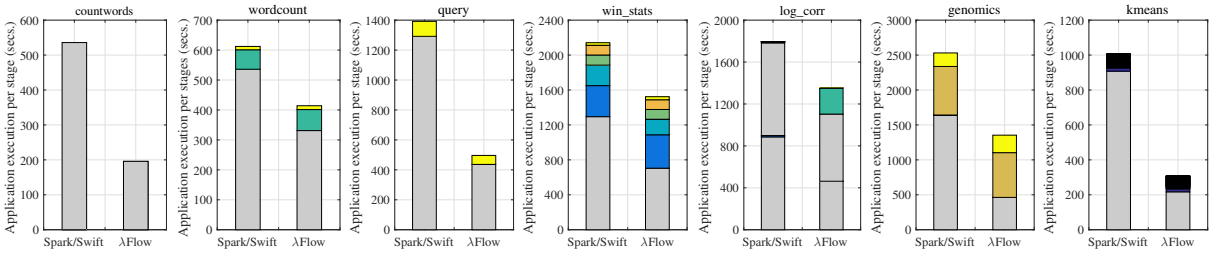


Figure 39: Execution times per stage for all applications considered w/wo λFlow. Grey areas in stacked bars represent data ingestion stages.

at the storage side. Therefore, the data reduction for each application is achieved in a different way; for instance, most applications benefit from filtering useless rows (`filter`), others select a particular part of a record (`map`), and there are other applications that directly compute partial results —e.g., sums, counts, maximum finding— at each object (`reduce`). As we discuss later on, this opens the door to investigate the feasibility of richer types of lambdas in λFlow.

Application completion times. Next, we examine how the lambda pipelines migrated to the storage side, as well as the changes in the original application code, impact on application execution times. In this sense, Fig. 39 show the execution times per-stage of all the applications in Table 8.

At first glance, Fig. 39 shows that λFlow clearly improves execution times in all the applications considered in our experiments. Specifically, we measure the speedup S for an application A as $S = \frac{T^A_{vanilla}}{T^A_{\lambda Flow}}$, where T^A stands for an A's execution time. Given this metric, λFlow achieves speedups that range from $S = 1.35$x (`log_corr`) to $S = 3.13$x (`genomics`). For the real use-case that motivates this work, GridPocket `query` and `win_stats` applications obtain a spedup of $S = 2.80$x and $S = 1.41$x, respectively. We observe that there is an evident correlation between the percentage of data transfer reduction (Table 9) and the speedup achieved by λFlow; however, other aspects such as the computational complexity of lambda pipelines and the compute power at the storage side are also very relevant for speedup gains.

The main cause for λFlow speedup gains lies on boosting heavy data ingestion stages within an application's execution. Such data ingestion stages are depicted in Fig. 39 as grey areas in stacked bars. In this sense, the modified code version of most applications presents the same sequence of execution stages than the original one. However, the case `log_corr` suggests that the modified version of an application, which yields a different set of transformations applied on RDDs, may lead Spark to build a different DAG with different stages.

Fig. 40 describes the behavior of application at the task level. Conversely to what happens with stages, the execution times of tasks seem to exhibit greater differences among applications. This mainly depends on the number of tasks related to data ingestion stages and applications exhibit different profiles. First, in applications like `kmeans` most of the tasks are short and related to iterative in-memory computations (algorithm convergence); this yields that λFlow only improves the execution tasks of a small group of long tasks (distribution tail in Fig. 40) related to data ingestion. Second, applications like `wordcount` show a differentiated behavior between data ingestion tasks, which exhibit shorter execution times, and tasks related to in-memory computations (shuffle, reduce of `(word,count)` pairs) that are similar to the original application. Finally, we observe that executing lambdas at the storage side in other applications (`genomics`, `win_stats`) shortens most of the task execution times.

Multi-tenant workloads. Next, we explore the benefits of λFlow in shared analytics infrastructures. Fig. 41 shows the execution times and bandwidth usage of 3 applications in parallel (one application per compute node).

First, in Fig. 41 (left) we observe that the sharing resources across applications in a multi-tenant scenario leads to significantly higher application execution times. Specifically, `query`, `wordcount` and `kmeans` execution times are 1.94x, 2.92x and 2.83x larger compared with when they ran alone with the

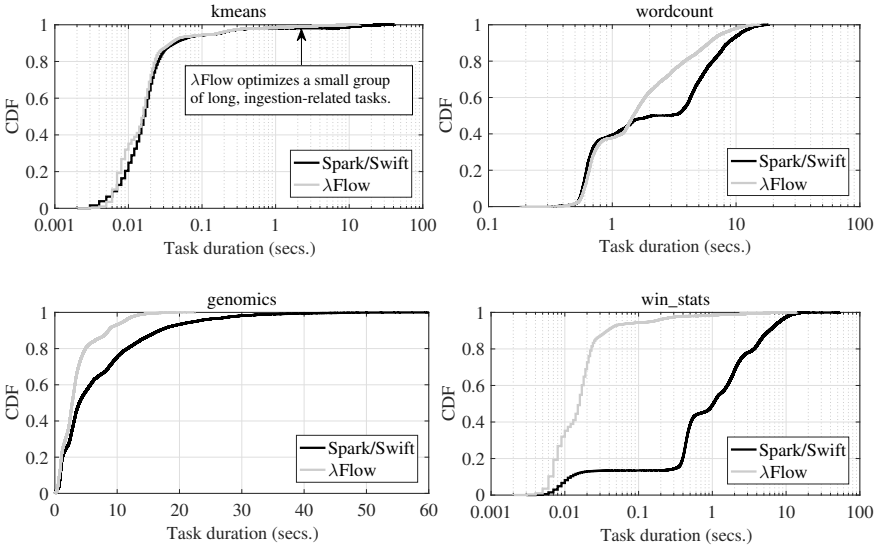


Figure 40: Distribution of application task execution times w/wo λFlow.

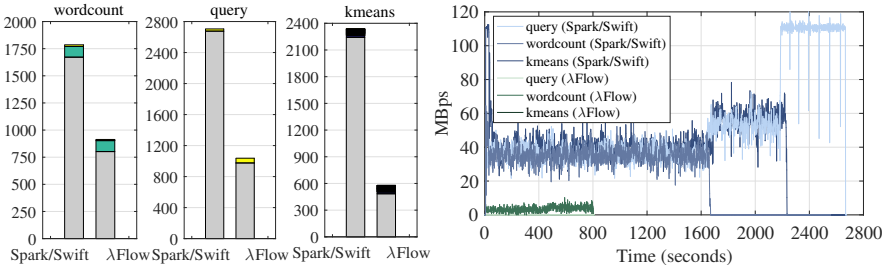


Figure 41: Execution times per stage and time-series bandwidth for 3 applications in parallel w/wo λFlow.

cluster. Despite this slowdown is in part motivated given that compute resources per application are only a fraction of the total available, the most important bottleneck is sharing bandwidth resources. This can be confirmed by inspecting Fig. 41 (right), as most of an application's execution time is related to ingest data in a situation of scarce bandwidth.

Fig. 41 also demonstrates that λFlow constitutes a solution to minimize data ingestion in multi-tenant scenarios, providing important application speedups. That is, in Fig. 41 λFlow achieves execution time speedups of $S = 2.61$x, $S = 1.95$x, and $S = 3.98$x in `query`, `wordcount` and `kmeans` applications, respectively. In our cluster, these speedups are similar or better than the ones obtain when running applications alone. The reason is that executing lambdas at the storage side is both reducing bandwidth consumption and offloading computation tasks, which benefits the performance of data-intensive analytics.

Exploiting migration policies. We evaluate how the feedback loop of λFlow (i.e., metrics, migration policies) enables us to migrate lambdas upon specific conditions (Section 15.4.3).

First, we present a scenario where the execution engine for synchronous lambdas is co-located with storage nodes. In this situation, an administrator may decide to execute lambdas that exploit the spare compute power of storage nodes only when bandwidth is scarce (`available_bw>80MBps`). To demonstrate this, Fig. 42 (left) shows the parallel execution of `query` and `wordcount` applications. The first execution of `wordcount` is done as usual, as at the beginning of the experiment all the bandwidth is available. Then, the `query` application starts its execution, and both are sharing bandwidth for a period of time; that is, the decision of migrating lambdas or not is done when the application is submitted and cannot be changed on runtime. Then, a second execution of the `wordcount` application

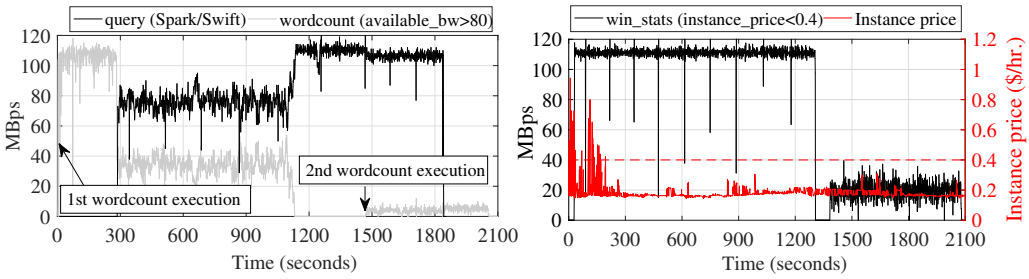


Figure 42: Enforcement of λFlow migration policies on different multi-tenant workloads.

occurs when the `query` application is still running and consuming most of the available bandwidth. In this situation, the controller enables the migration of lambdas to optimize the parallel execution of applications.

Besides, migration policies can be useful for the deployment of λFlow in the cloud. In this case, the execution engine for synchronous lambdas may be deployed in an elastic compute platform [95] and data is stored in an object storage service. A tenant of such deployment pays for both bandwidth and compute resources, so the migration of lambdas may be subject by the variable price of compute resources, as in AWS Spot Instances (`instance_price < 0.4`). Thus, Fig. 42 (middle) shows the execution of `xxx` application in two points in time, as well as "price events" of AWS Spot Instances (large instance, Linux) that the migration controller receives as input. As can be noted, the first execution of the application is done as regular, given that the price of the instance at that time was higher than 0.4. Instead, in the second execution the controller decides to perform migration of lambdas, given that the price was low enough to meet the condition.

15.6 Related Work

The "data ingestion problem" has gained traction in the last years [87]. From the analytics viewpoint, some works optimize the workflow of MapReduce programs to minimize data movements [104]. In fact, a role of λFlow is to act as an optimization mechanism on input applications to migrate dataflow computations that reduce data ingestion. Other works augmented Hadoop/Spark with indexes on structured storage (HBase) for pruning useless data partitions [82, 86]. Interestingly, indexing mechanisms in object stores could also help to filter out entire data objects [18]. This technique is compatible with our work, as λFlow computes on data streams of the data objects that are required by the index.

Other efforts attacked the data ingestion problem purely from a storage perspective. To wit, recent works have focused on interfacing Hadoop with enterprise file systems to bridge the gap between legacy data stores and compute clusters [105, 106, 107], or even replacing HDFS by optimized file systems to minimize the impact of disaggregation [108, 109]. In this sense, λFlow exploits the new stateless or lambda computing paradigm at the storage side as a building block to optimize data intensive analytics [95, 94]. Despite performing computations at the storage side is a well-known technique (e.g., active storage, co-processors) [56, 57], λFlow is the first framework to automatically orchestrate end-to-end computations across dataflow engines and object stores.

The closest related work we are aware of is that of Gkantsidis et al. [85]. Authors present a system called Rhea that aims at filtering data at the storage side to minimize data ingestion in Hadoop. Even more, Rhea presents a static analysis tool to infer the filters to apply at the storage side automatically. Despite sharing the same spirit, our work presents important differences compared to Rhea: first, Rhea focuses on optimizing SQL filters/selects in Hadoop, whereas λFlow uses richer programming APIs on data streams, thus coping with diverse analytics applications and engines. λFlow exploits parallel stream processing close to the data, but Rhea instead relies on a single proxy to perform filtering tasks. Conversely to Rhea, λFlow provides a migration policy-based scheduling mechanism for wisely executing filters at the storage side.

In summary, λFlow greatly extends our previous efforts [21] to become the first framework that

automatically combines stream processing and dataflow computing to optimize data intensive analytics. We believe that the emergence of systems like Apache Beam [88] supports the ideas behind λFlow, motivating its use for optimizing multiple data flow engines.

Part V
Conclusions

In this deliverable, we presented the release of the IOStack toolkit for month 36. At the end of the project, we achieved to deliver an integrated SDS solution for Big Data analytics with several deployments running (URV, Arctur, Idiada). To demonstrate the achievements of the toolkit, we provided a summarized overview of the uses-cases and their problems, the toolkit KPIs that solve such problems, and the exploitation plans of use-cases for IOStack.

Second, we described the IOStack toolkit. It consists of three main building blocks for analytics virtualization (Zoe), block storage (Konnector) and object storage (Crystal), as well as an administration/monitoring dashboard and other components that complete the toolkit (Storlets, Stocator, FUSE Client). We also presented that the different software components of IOStack are adhered to design principles described in D2.2 and integrated into a single toolkit. Moreover, this does not prevent that each software component of the IOStack toolkit can be exploited in a standalone manner.

In the third part of the deliverable we described the architecture of Crystal: The IOStack SDS building block for object storage. Specifically, Crystal pursues an efficient use of multi-tenant object stores that need to support non-anticipated requirements. Crystal addresses unique challenges for providing the necessary abstractions to add new functionalities at the data plane that can be immediately managed at the control plane. For instance, it adds a filtering abstraction to separate control policies from the execution of computations and resource management mechanisms at the data plane. Also, extending Crystal requires low development effort. We demonstrate the feasibility of Crystal on top of OpenStack Swift through two use cases that target automation and bandwidth differentiation making use of benchmarks and real workloads from our use case companies (Arctur, Idiada). Our results show that Crystal is practical enough to be run in a shared cloud object store.

Finally, the extensible and software-defined architecture of IOStack building blocks facilitates the implementation of "cross-layer" optimizations; that is, control algorithms and mechanisms that orchestrate multiple building blocks (e.g., compute and storage components) to optimize workloads. We demonstrate this kind of optimization strategies by: i) designing hyper-controllers that orchestrate storage services (Crystal) based on the type of applications deployed at the compute cluster (Zoe); and ii) adding to the Crystal architecture the capability of analyzing an input analytics application and identify the computations that can be executed at the storage side more efficiently. These examples show that IOStack is still an interesting substrate to develop research in the field of storage and analytics in the future.

Appendices

A Crystal Controller API

The next table summarizes the REST methods of Crystal Controller API final release. For further details and examples, refer to the documentation[27].

REST Call Description	HTTP Method	URL
Filters		
List filters	GET	`/filters`
Create a filter	POST	`/filters`
Upload filter data	PUT	`/filters/:filter_id/data`
Delete a filter	DELETE	`/filters/:filter_id`
Get filter metadata	GET	`/filters/:filter_id`
Update filter metadata	PUT	`/filters/:filter_id`
Deploy a filter to a project	PUT	`/filters/:project_id/deploy/:filter_id`
Deploy a filter to a project and a container	PUT	`/filters/:project_id/:container/deploy/:filter_id`
Undeploy a filter from a project	PUT	`/filters/:project_id/undeploy/:filter_id`
Undeploy a filter from a project and a container	PUT	`/filters/:project_id/:container/undeploy/:filter_id`
Dependencies		
Create a dependency	POST	`/filters/dependencies`
Upload dependency data	PUT	`/filters/dependencies/:dep_id/data`
Delete a Dependency	DELETE	`/filters/dependencies/:dep_id`
Get Dependency metadata	GET	`/filters/dependencies/:dep_id`
List Dependencies	GET	`/filters/dependencies`
Update Dependency metadata	PUT	`/filters/dependencies/:dep_id`
Deploy Dependency	PUT	`/filters/dependencies/:project_id/deploy/:dep_id`
Undeploy Dependency	PUT	`/filters/dependencies/:project_id/undeploy/:dep_id`
List deployed Dependencies of a project	GET	`/filters/dependencies/:project_id/deploy`
Workload metrics		
Add a workload metric	POST	`/metrics`
Get all workload metrics	GET	`/metrics`
Get activated workload metrics	GET	`/metrics/activated`
Update a metric module	PUT	`/metrics/:metric_module_id`
Upload a metric module	PUT	`/metrics/:metric_module_id/data`
Get metric metadata	GET	`/metrics/:metric_module_id`
Delete a metric module	DELETE	`/metrics/:metric_module_id`
Crystal Projects		
List all Crystal enabled projects	GET	`/projects`
Enable a project	PUT	`/projects/:project_id`
Disable a project	DELETE	`/projects/:project_id`
Check if a project exists or is enabled	POST	`/projects/:project_id`
Projects group		
Add a projects group	POST	`/projects/groups`
Get all projects groups	GET	`/projects/groups`
Get projects of a group	GET	`/projects/groups/:group_id`
Update members of a projects group	PUT	`/projects/groups/:group_id`
Delete a projects group	DELETE	`/projects/groups/:group_id`
Delete a member of a projects group	DELETE	`/projects/groups/:group_id/projects/:project_id`
Object type		
Create an object type	POST	`/policies/object_type`
Get all object types	GET	`/policies/object_type`
Get extensions of an object type	GET	`/policies/object_type/:object_type_name`
Update extensions of an object type	PUT	`/policies/object_type/:object_type_name`
Delete an object type	DELETE	`/policies/object_type/:object_type_name`

[27] `https://github.com/Crystal-SDS/controller`

DSL Policies		
List all static policies	GET	`/policies/static`
Add a static policy	POST	`/policies/static`
Get a static policy	GET	`/policies/static/:project_policy_id`
Update a static policy	PUT	`/policies/static/:project_policy_id`
Delete a static policy	DELETE	`/policies/static/:project_policy_id`
List all dynamic policies	GET	`/policies/dynamic`
Add a dynamic policy	POST	`/policies/dynamic`
Delete a dynamic policy	DELETE	`/policies/dynamic/:policy_id`
SLO Info		
Get SLO info about all projects	GET	`/policies/slos`
Establish a SLO for the selected filter and target	POST	`/policies/slos`
Get SLO info for a filter and target	GET	`/policies/slos/:dsl_filter/:slo_name/:target`
Edit a SLO for the selected filter and target	PUT	`/policies/slos/:dsl_filter/:slo_name/:target`
Delete a SLO for the selected filter and target	DELETE	`/policies/slos/:dsl_filter/:slo_name/:target`
Controllers		
Get the controllers list	GET	`/controllers`
Upload a new controller	POST	`/controllers/data`
Get controller metadata	GET	`/controllers/:controller_id`
Update controller metadata	PUT	`/controllers/:controller_id`
Delete a controller	DELETE	`/controllers/:controller_id`
Upgrade an existing controller	POST	`/controllers/data/:controller_id`
Download an existing controller	GET	`/controllers/data/:controller_id`
Swift calls		
Get the storage policies list	GET	`/swift/storage_policies`
Create a storage policy	POST	`/swift/storage_policies`
Get storage policy details	GET	`/swift/storage_policies/:storage_policy_id`
Get the nodes list	GET	`/swift/nodes`
Get node details	GET	`/swift/nodes/:server_type/:node_id`
Update node data	PUT	`/swift/nodes/:server_type/:node_id`
Restart a node	PUT	`/swift/nodes/:server_type/:node_id/restart`
Get the regions list	GET	`/swift/regions`
Create a new region	POST	`/swift/regions`
Get region details	GET	`/swift/regions/:region_id`
Update region data	PUT	`/swift/regions/:region_id`
Delete a region	DELETE	`/swift/regions/:region_id`
Get the zones list	GET	`/swift/zones`
Create a new zone	POST	`/swift/zones`
Get zone details	GET	`/swift/zones/:zone_id`
Update zone data	PUT	`/swift/zones/:zone_id`
Delete a zone	DELETE	`/swift/zones/:zone_id`

B Crystal Development VM

There is a development virtual machine image available for Crystal. This VM emulates running a four node Swift cluster together with Keystone, Storlets and Crystal controller and middlewares.

The VM is accessible at `ftp://ast2-deim.urv.cat/s2caio_vm/`.

C JIT prefetching

The prefetch system is able to preload objects just before they are used in order to reduce the quantity of memory used for caching. Just-In-Time prefetching creates a Markov chain to provide the prediction capabilities, pre-loading the object taking into account different temporal parameters as the download time or number of hits. Finally, the prefetch system is provided as a filter and uses a Software Defined Storage called Crystal to provide an easy way to activate or deactivate it in OpenStack Swift.

In terms of normal behaviour of the Filter, there is no learning phase and prefetching phase, both are happening at the same time (feedback system or reinforcement learning). When an object arrives to the filter, we predict which objects need to be prefetched, we prefetch them and update the Markov chain, all in the same iteration. However, in order to evaluate the filter and to explain it better we are going to divide those processes into two subsections: prediction, or how the Markov chain is built, and prefetch, or how we prefetch the objects.

In order to be able to predict which objects need to be prefetched, the first iteration of the filter creates a Markov chain that stores the order in which the objects have been requested but also the minimum time difference (the worst case) between requests. Due to the Crystal implementation,

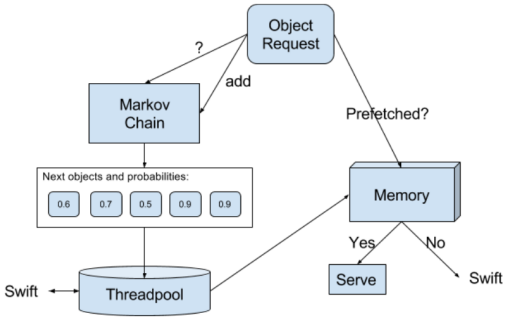


Figure 43: JIT diagram.

we just have control of each object before the actual GET happens, we cannot know in advance it's download time, so, in a second iteration (and in successive, through a feedback system) the chain is also filled with download time in MB/s for each object when prefetched (in this case, we store the maximum one, also the worst case).

The information of the Markov chain uses a special format that allows us to distribute it over a memcached server or other key-value database, if necessary.

When an object arrives to the filter we check for its entry in the Markov chain (Figure 43) and we retrieve the stats for that current object and also the stats for the next objects. So to say, we gather two levels or steps of the Markov chain: the current node and all the next ones. This is configurable, but having a lot of levels will not be aligned with the JIT objective. Once we have all this data we compute the probability of each of the objects by dividing each number of hits by the total number of hits. If an object appears twice (once in each Markov step) we just add both hits in a single number. Before prefetching an object we also check if it has been prefetched and not served (already in memory) so we avoid recursively ask the Object Storage for the same objects.

If the time difference between two objects is larger than 60 seconds we don't consider those two objects as consecutive so we won't store that relation to the chain. Also in order to keep a clean cache we have a maximum cache size and we autoremove prefetched objects that have not been served.

In order to not loose this chain we have a thread that saves it every X seconds as a pickle file in the same server than executes the filter. This chain is then loaded when the filter is called for the first time. The chain can also be saved and loaded to a memcached-like service.

Results. In this experiment, we use a real trace provided by Arctur that can be found in IOStack website. The Arctur trace captures 2.97TB of a read-dominated (99.97% read bytes) Web workload consisting of requests related to 228K small data objects (mean object size is 0.28MB) from several Web pages hosted at Arctur datacenter for 1 month. All the tests presented in this work are executed in the IOStack testbed (`http://testbed.iostack.eu`) provided by Arctur. Swift (Kilo version) installation consists of three single HDD storage nodes connected through 3 GbE plus one proxy node connected with 1 Gigabit Ethernet to the clients. We executed a workload replay process (SwiftWorkloadExecutor) a client node requesting to the proxy node, thus outside the 4 Swift machines cluster.

As a preparatory phase for the experiment, we need to execute the trace but pushing the objects to Swift in order to be able to perform `GETs` afterwards. Using this SwiftWorkloadExecutor we are also able to modify the trace in order to increase the file size, sort the trace, introduce some errors (changing requests times, order, etc.).

The results running without modification are not satisfactory (Table 10) as the objects are served faster from the object server as we have 3 GbE links, and the proxy only communicates to the client with 1 GbE link. In order to better understand the results, we filter them by object size. Small objects refer to those smaller than percentile 0.25, large objects refer to those bigger than percentile 0.75 and medium those in between 0.25 and 0.75.

The second experiment tries to increase the filesize in order to reduce the benefits from the faster links. With a 10x increase of the file size, we obtain a mean benefit of a 5% (Table 11).

Table 10: Improvement with JIT original workload.

Download Time improvement	100Mbit/s	200Mbit/s	1Gbit/s
small	14%	15%	15%
medium	14%	15%	15%
large	-5%	-5%	-5%
Total	7%	0%	-7%

As the problem is the particular hardware and network setup, we decided to decrease the link speed of the object servers/proxy. This speed decrease also implies that the prefetched data is obtained slower from the proxy and the Markov chain needs to adapt (automatically) to the new speeds.

Table 11: Improvement with JIT 10x workload.

Download Time improvement	100Mbit/s	200Mbit/s	300Mbit/s	1Gbit/s
small	22.2%	23.5%	0%	5.5%
medium	44.8%	32%	15%	15%
large	69.5%	59.8%	31.7%	10.2%
Total	37.5%	23.07%	4.76%	4.76%

As a side effect of the filter and the network shaping we also obtain an improvement in throughput as the prefetched objects (served directly from the proxy) are much faster than requesting an object directly through the shaped network (Table 12).

Table 12: Throughput improvement.

Throughput improvement	100Mbit/s	200Mbit/s	300Mbit/s	1Gbit/s
small	29%	17.2%	13.3%	23.1%
medium	45.2%	22.7%	1.18%	8.47%
large	227.8%	89.5%	14.16%	-1.1%
Total	45.9%	28.6%	1.18%	8.7%

D KPI Questionnaires

Next, we include the questionnaires that IOStack toolkit end users and system administrators filled in order to help evaluate Key Performance Indicators classified as *management costs* in Sec. 3. The questions cover technical, usability and performance scopes, and are related to KPIs. 1, 2, 8, 9, and 12. Overall, answers show improvements in all scopes for all three partners (Eurecom, Gridpocket, Idiada).

Eurecom questionnaire is focused around Zoe. The end user of this IOStack component is both the data scientist that want to run analytics applications and the system administrator that configures and manages the cluster. At Eurecom, a teaching and research institution, Zoe is being used both to run analytics experiments as well as in laboratory sessions of the Algorithmic Machine Learning course. From the questionnaire answers, we observe that Zoe improves all the analyzed items. From technical scope, it requires a minimum adaptation and provides a high level of abstraction as well as flexibility. On the performance part, Zoe shows a good system responsiveness and efficiency and allows the apps to scale up and use all available resources dynamically and automatically.

Gridpocket end users are also data scientists and system administrators. They have used IOStack toolkit to improve the speed and efficiency of their analytics workloads. From the questionnaire, we see that one of the more outstanding improvement is the speedup of data ingestion. Gridpocket data scientists also highlight that IOStack project was key to release MeterGen tool as open source. This tool allows to generate anonymous data to be used in research. Regarding performance, IOStack made it possible to increase the performance of analytics on large scale, mainly with specific smart grid queries and geospatial analytics. Gridpocket also remarks that IOStack toolkit is natural and efficient and could be adopted in an easy way.

Idiada end users are system administrators that use the IOStack toolkit to adapt their legacy systems to new and varying requirements, mainly related to reduce storage costs and to increase storage management flexibility. Overall, we observe that IOStack toolkit is able to improve most of the evaluated items, in particular the scalability of the system, the usability, and the management costs.

KPI Questionnaire

Partner: Eurecom

Partner contact: venzano@eurecom.fr

End user: **1.) data scientist** or **2.) system administrator**

TECHNICAL KPI:

1. level of adaptation needed

		1	2	3	4	5	
Without IOStack	A lot of adaptation needed	x					Minimum adaptation required
With IOStack	A lot of adaptation needed				x		Minimum adaptation required

Comment: Zoe, a component of IOStack, makes access to analytics tools easier and more transparent, requiring less adaptation than learning the full systems stack involved.

2. level of abstraction (e.g. zoe <-> hadoop)

		1	2	3	4	5	
Without IOStack	Little to no abstraction, geared towards underlying technology	x					High level of abstraction
With IOStack	Little to no abstraction, geared towards underlying technology				x		High level of abstraction

Comment: Zoe abstracts the systems details of deploying and running distributed analytics frameworks.

3. flexibility

		1	2	3	4	5	
Without IOStack	System is not flexible	x					System is highly flexible
With IOStack	System is not flexible					x	System is highly flexible

Comment: Zoe's high level description language let the user describe arbitrary analytics applications and deploy them easily. Support for new versions of existing frameworks or even new software can be added very quickly to Zoe.

4. Quality of analytics data science

		1	2	3	4	5	
Without IOStack	No analytics data is available	x					Adequate amount of analytics data is available
With IOStack	No analytics data is available					x	Adequate amount of analytics data is available

Comment: The user is able to concentrate on the data science part of his job and leave all the low-leave, systems details to Zoe (the system administrator is not really interested in the analytics).

5. status of execution

		1	2	3	4	5	
Without IOStack	No info on status	x					Status info readily available
With IOStack	No info on status					x	Status info readily available

Comment: Zoe provides status info at any given moment and makes it also actionable so that you are able to react on the fly.

PERFORMANCE KPI:

1. system responsiveness (time in queue)

		1	2	3	4	5	
Without IOStack	Idle time more than execution time	x					Minimum idle time
With IOStack	Idle time more than execution time				x		Minimum idle time

Comment: see https://arxiv.org/abs/1611.09528

2. resource utilization

		1	2	3	4	5	
Without IOStack	No scale-up	x					App scales linearly
With IOStack	No scale-up					x	App scales linearly

Comment: With Zoe applications can scale and use all available resources dynamically and automatically.

3. overall system efficiency - lower time to insight

		1	2	3	4	5	
Without IOStack	High deployment time and data set size needed for insight	x					Minimum deployment time and data set size needed for insight
With IOStack	High deployment time and data set size needed for insight					x	Minimum deployment time and data set size needed for insight

Comment: Zoe takes care of automatic deployment of software and cluster making so that the user can start to work on his case immediately without needing to install or configure anything. The user also can work on full data sets, instead of having to test his algorithm on a small subset first on his laptop.

KPI Questionnaire

Partner: Gridpocket

Partner contact: filip.gluszak@gridpocket.com

End user: Data scientist / System administrator

TECHNICAL KPI:

1. Speedup of data ingestion

		1	2	3	4	5	
Without IOStack	data ingestion is unchanged		x				data ingestion is seamless
With IOStack	data ingestion is unchanged					x	data ingestion is seamless

Comment:

2. Anonymization on data generation (data privacy)

		1	2	3	4	5	
Without IOStack	Data cannot be anonymized		x				Data is fully anonymous
With IOStack	Data cannot be anonymized				x		Data is fully anonymous

Comment:

3. Scalability (nr. of users)

		1	2	3	4	5	
Without IOStack	System accepts only one user				x		System scales to indefinite amount of users
With IOStack	System accepts only one user					x	System scales to indefinite amount of users

Comment:

USABILITY KPI:

1. release MeterGen as OS with documentation feasibility

		1	2	3	4	5	
Without IOStack	Low probability to deliver MeterGen tool to the community		x				High probability to deliver MeterGen tool to the community
With IOStack						x	

Comment:

2. market readiness/viability

		1	2	3	4	5	
Without IOStack	System is not applicable to the market			x			System is market ready
With IOStack	System is not applicable to the market				x		System is market ready

Comment:

PERFORMANCE KPI:

1. implementation of GridPocket specific queries

		1	2	3	4	5	
Without IOStack	Implementation of smart grid specific queries difficult on large scale			x			Implementation of smart grid specific queries easy and efficient on large scale
With IOStack						x	

Comment:

2. geospatial data analytics in IOStack -

		1	2	3	4	5	
Without IOStack	Geospatial analytics complex to integrate on large scale			x			Geospatial analytics easy and efficient to integrate on large scale
With IOStack						x	

Comment:

3. batch & interactive query management

		1	2	3	4	5	
Without IOStack	The system is not interactive			x			The system is fully interactive
With IOStack	The system is not interactive				x		The system is fully interactive

Comment:

4. overall cost efficiency

		1	2	3	4	5	
Without IOStack	System is time consuming and expensive to run			x			System is natural and efficient
With IOStack	System is time consuming and expensive to run					x	System is natural and efficient

Comment:

5. case of future adoption - capacity of the systems to be adopted by users and the market

		1	2	3	4	5	
Without IOStack	System difficult to adopt in the medium future			x			System easy to adopt
With IOStack	System difficult to adopt in the medium future					x	System easy to adopt

Comment:

KPI Questionnaire

Partner: IDIADA

Partner contact: saxa.egea@idiada.com, aleix.ramirez@idiada.com

End user: System administrator

TECHNICAL KPI:

1. flexibility

		1	2	3	4	5	
Without IOStack	The system does not support adaptation				x		All parameters can be (re)configured
With IOStack	The system does not support adaptation				x		All parameters can be (re)configured

Comment:

2. compression rate

		1	2	3	4	5	
Without IOStack	Compression not possible				x		Compression is adequate
With IOStack	Compression not possible					x	Compression is adequate

Comment:

3. security of info

		1	2	3	4	5	
Without IOStack	System is badly secured				x		System is secure
With IOStack	System is badly secured					x	System is secure

Comment:

4. integration

		1	2	3	4	5	
Without IOStack	The system does not allow for integration with other systems			x			Integration is available
With IOStack	The system does not allow for integration with other systems				x		Integration is available

Comment:

5. scalability

		1	2	3	4	5	
Without IOStack	System is not scalable at all			x			System scales linearly
With IOStack	System is not scalable at all					x	System scales linearly

Comment:

6. percentage of features accomplished

		1	2	3	4	5	
Without IOStack	Low percentage of features accomplished			x			High percentage of features accomplished
With IOStack	Low percentage of features accomplished				x		High percentage of features accomplished

Comment: Each required feature adds 1 point to a total of 5 for the KPI:

- Encryption: Files encrypted depending its content (1 point)
- Compression: Compression depending file type (1 point)
- Access control depending on source network, Business Unit or location. (1 point)
- Access control depending on file type. (1 point)
- Removal of files based on rules: file type, period of time…(1 point)

USABILITY KPI

1. Ease of use

		1	2	3	4	5	
Without IOStack	Lack of user interface, confusion about system parameterization, lack of documentation		x				Solution provides Graphical User Interface, a system configuration interface, documentation
With IOStack	Lack of user interface, confusion about system parameterization, lack of documentation				x		Solution provides Graphical User Interface, a system configuration interface, documentation

Comment:

PERFORMANCE KPI:

1. time of response (from submission to response)

		1	2	3	4	5	
Without IOStack	Response time is relatively long		x				Response time is negligible
With IOStack	Response time is relatively long			x			Response time is negligible

Comment:

2. cost and/or management

		1	2	3	4	5	
Without IOStack	System is time consuming and expensive		x				System is inherent and economic
With IOStack	System is time consuming and expensive				x		System is inherent and economic

Comment:

3. bandwidth

		1	2	3	4	5	
Without IOStack	Connection bandwidth is not enough optimized			x			Connection bandwidth is optimized and more than acceptable
With IOStack	Connection bandwidth is not enough optimized				x		Connection bandwidth is optimized and more than acceptable

Comment:

Glossary

Container (object storage): A container organizes and stores objects in Object Storage. Similar to the concept of a Linux directory but cannot be nested.

Domain-specific language (DSL): A computer language specialized to a particular application domain.

Logical Volume Manager (LVM): A device mapper target that provides logical volume management for the Linux kernel.

MD RAID: Also called Linux software RAID, makes the use of RAID possible without a hardware RAID controller.

Message Oriented Middleware (MOM): A software or hardware infrastructure supporting sending and receiving messages between distributed systems.

MOM broker: An intermediary program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. MOM brokers typically provide content and topic-based message routing using the publish/subscribe pattern.

Storage provisioning: The process of assigning storage, usually in the form of server disk drive space, in order to optimize the performance of a storage area network.

Storage tiering: the process of assigning different categories of data to various types of storage media to reduce total storage cost. Tiers are determined by performance and cost of the media, and data is ranked by how often it is accessed. Tiered storage policies place the most frequently accessed data on the highest performing storage. Rarely accessed data goes on low-performance, cheaper storage.

Tenant: represents the base unit of "ownership" in OpenStack, in that all resources in OpenStack should be owned by a specific tenant.

References

[1] S. LaValle, E. Lesser, R. Shockley, M. S. Hopkins, and N. Kruschwitz, "Big data, analytics and the path from insights to value," MIT sloan management review, vol. 52, no. 2, p. 21, 2011.

[2] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: current state and future opportunities," in Proceedings of the 14th International Conference on Extending Database Technology, pp. 530–533, 2011.

[3] T. White, Hadoop: The definitive guide. O'Reilly Media, Inc., 2012.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in IEEE MSST'10, pp. 1–10, 2010.

[5] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.," in USENIX OSDI'08, vol. 8, pp. 1–14, 2008.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in USENIX NSDI'12, pp. 2–2, 2012.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[8] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," Proceedings of the VLDB Endowment, vol. 2, no. 2, pp. 1626–1629, 2009.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al., "Spark sql: Relational data processing in spark," in ACM SIGMOD'15, pp. 1383–1394, 2015.

[10] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al., "Mllib: Machine learning in apache spark," The Journal of Machine Learning Research, vol. 17, pp. 34:1–34:7, 2016.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," ACM SIGOPS operating systems review, vol. 37, no. 5, pp. 29–43, 2003.

[12] "Openstack swift." `http://docs.openstack.org/developer/swift`.

[13] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44, no. 2, pp. 35–40, 2010.

[14] Google, "Kubernetes." http://kubernetes.io/, 2017.

[15] R. Gracia-Tinedo, P. García-López, M. Sanchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortes, and W. Oppermann, "IOStack: Software-defined object storage," IEEE Internet Computing, 2016.

[16] R. Gracia-Tinedo, J. Sampé, E. Zamora-Gómez, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom, "Crystal: Software-defined storage for multi-tenant object stores," in USENIX FAST'17, pp. 243–256, 2017.

[17] F. Pace, D. Venzano, D. Carra, and P. Michiardi, "Flexible scheduling of distributed analytic applications," in IEEE/ACM CCGrid'17, pp. 100–109, 2017.

[18] P. Ta-Shma, "Using pluggable spark sql filters to help gridpocket users keep up with the jones' (and save the planet)." https://spark-summit.org/eu-2017/events/using-pluggable-apache-spark-sql-filters-to-help-gridpocket-users-keep-up-with-the-jones-

[19] G. Vernik, M. Factor, E. K. Kolodner, E. Ofer, P. Michiardi, and F. Pace, "Stocator: a high performance object store connector for spark," in ACM SYSTOR'17, p. 27, 2017.

[20] G. Vernik, M. Factor, E. K. Kolodner, E. Ofer, P. Michiardi, and F. Pace, "Stocator: An object store aware connector for apache spark," in ACM SoCC'17, 2017.

[21] Y. Moatti, E. Rom, R. Gracia-Tinedo, D. Naor, D. Chen, J. Sampe, M. Sanchez-Artigas, P. Garcıa-Lopez, F. Gluszak, E. Deschdt, et al., "Too big to eat: Boosting analytics data ingestion from object stores with scoop," in IEEE ICDE'17, pp. 309–320, 2017.

[22] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Mos: Workload-aware elasticity for cloud object stores," in ACM HPDC'16, pp. 177–188, 2016.

[23] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Taming the cloud object storage with mos," in Proceedings of the 10th Parallel Data Storage Workshop, pp. 7–12, 2015.

[24] M. Factor, G. Vernik, and R. Xin, "The perfect match: Apache spark meets swift." https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/the-perfect-match-apache-spark-meets-swift, November 2014.

[25] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: a software-defined storage architecture," in ACM SOSP'13, pp. 182–196, 2013.

[26] M. Murugan, K. Kant, A. Raghavan, and D. H. Du, "Flexstore: A software defined, energy adaptive distributed storage framework," in IEEE MASCOTS'14, pp. 81–90, 2014.

[27] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in ACM SoCC'15, pp. 174–181, 2015.

[28] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska, "sRoute: treating the storage stack like a network," in USENIX FAST'16, pp. 197–212, 2016.

[29] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom, "A pipelined framework for online cleaning of sensor data streams," in IEEE ICDE'06, pp. 140–140, 2006.

[30] "OpenStack Storlets." https://github.com/openstack/storlets.

[31] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in Proceedings of the 8th ACM European Conference on Computer Systems, pp. 351–364, ACM, 2013.

[32] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "Tr-spark: Transient computing for big data analytics," in Proceedings of the Seventh ACM Symposium on Cloud Computing, pp. 484–496, ACM, 2016.

[33] F. Pace, M. Milanesio, D. Venzano, D. Carra, and P. Michiardi, "Experimental performance evaluation of cloud-based analytics-as-a-service," in IEEE CLOUD'16, pp. 196–203, 2016.

[34] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics," in ACM SoCC'11, p. 18, 2011.

[35] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: all for one, one for all in data processing systems," in ACM EuroSys'15, p. 2, 2015.

[36] H. Alkaff, I. Gupta, and L. M. Leslie, "Cross-layer scheduling in cloud systems," in IEEE IC2E'15, pp. 236–245, 2015.

[37] S. Rizvi, X. Li, B. Wong, F. Kazhamiaka, and B. Cassell, "Mayflower: Improving distributed filesystem performance through sdn/filesystem co-design," in IEEE ICDCS'16, pp. 384–394, 2016.

[38] "IBM Cloud Object Storage." https://www.ibm.com/cloud/object-storage.

[39] J. M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates," in ACM SIGMOD'93, pp. 267–276, 1993.

[40] E. Friedman, P. Pawlowski, and J. Cieslewicz, "SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," VLDB Endowment, vol. 2, no. 2, pp. 1402–1413, 2009.

[41] "Openstack innovation center." https://osic.org.

[42] "Java 8 stream api." https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

[43] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, and M. Vukolic, "Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end," in ACM IMC'15, pp. 155–168, 2015.

[44] "Sdgen." https://github.com/iostackproject/SDGen.

[45] "meter_gen." https://github.com/gridpocket/project-iostack/tree/master/meter_gen.

[46] Eurecom, "Zoe." https://github.com/DistributedSystemsGroup/zoe.

[47] Jupyter, "Jupyter." http://jupyter.org/.

[48] Apache, "Spark." http://spark.apache.org/.

[49] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in ACM EuroSys'15, p. 18, 2015.

[50] Google, "Tensorflow." https://www.tensorflow.org/.

[51] "IBM Cloud (formerly known IBM Bluemix)." https://www.ibm.com/cloud.

[52] "Ifttt." https://ifttt.com.

[53] G. A. Agha, "Actors: A model of concurrent computation in distributed systems," tech. rep., The MIT Press, 1985.

[54] J. Armstrong, Programming Erlang: software for a concurrent world. Pragmatic Bookshelf, 2007.

[55] R. Stutsman, C. Lee, and J. Ousterhout, "Experience with rules-based programming for distributed, concurrent, fault-tolerant code," in USENIX ATC'15, pp. 17–30, 2015.

[56] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia applications," in VLDB'98, pp. 62–73, 1998.

[57] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in ACM/IEEE Supercomputing'07, p. 28, 2007.

[58] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," ACM Computing Surveys (CSUR), vol. 44, no. 3, p. 15, 2012.

[59] "Docker." `https://www.docker.com`.

[60] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in USENIX NSDI'15, 2015.

[61] A. Gulati and P. Varman, "Lexicographic qos scheduling for parallel i/o," in ACM SPAA'05, pp. 29–38, 2005.

[62] Y. Wang and A. Merchant, "Proportional-share scheduling for distributed storage systems.," in USENIX FAST'07, 2007.

[63] A. Gulati, A. Merchant, and P. J. Varman, "mclock: handling throughput variability for hypervisor io scheduling," in USENIX OSDI'10, pp. 1–7, 2010.

[64] E. Zamora-Gómez, P. García-López, and R. Mondéjar, "Continuation complexity: A callback hell for distributed systems," in LSDVE@Euro-Par'15, pp. 286–298, 2015.

[65] J. Sampé, M. Sánchez-Artigas, and P. García-López, "Vertigo: Programmable micro-controllers for software-defined object storage," in IEEE CLOUD'16, 2016.

[66] A. Gulati, I. Ahmad, C. A. Waldspurger, et al., "Parda: Proportional allocation of resources for distributed storage access.," in USENIX FAST'09, pp. 85–98, 2009.

[67] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: enabling high-level slos on shared storage systems," in ACM SoCC'12, p. 14, 2012.

[68] J. C. Wu and S. A. Brandt, "Providing quality of service support in object-based file system.," in IEEE MSST'07, vol. 7, pp. 157–170, 2007.

[69] N. Li, H. Jiang, D. Feng, and Z. Shi, "Pslo: enforcing the x th percentile latency and throughput slos for consolidated vm storage," in ACM Eurosys'16, p. 28, 2016.

[70] "Swift performance tunning." `https://swiftstack.com/docs/admin/middleware/ratelimit.html`.

[71] D. Shue, M. J. Freedman, and A. Shaikh, "Fairness and isolation in multi-tenant storage as optimization decomposition," ACM SIGOPS Operating Systems Review, vol. 47, no. 1, pp. 16–21, 2013.

[72] "The Panasas activescale file system (PanFS)." `http://www.panasas.com/products/panfs`.

[73] "PVFS Project." `http://www.pvfs.org/`.

[74] Intel, "SSBench benchmark." `https://github.com/swiftstack/ssbench`.

[75] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, and A. Zuck, "SDGen: mimicking datasets for content generation in storage benchmarks," in USENIX FAST'15, pp. 317–330, 2015.

[76] "Why Hyperconverged Infrastructure is so Hot." `http://www.datacenterknowledge.com/archives/2015/12/10/why-hyperconverged-infrastructure-is-so-hot`.

[77] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in ACM SIGMOD'15, pp. 1357–1369, 2015.

[78] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in USENIX HotCloud'10, vol. 10, p. 10, 2010.

[79] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol. 36, no. 4, 2015.

[80] "Apache apex." `http://apex.apache.org`.

[81] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in ACM SIGOPS operating systems review, pp. 59–72, 2007.

[82] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," Proceedings of the VLDB Endowment, vol. 3, no. 1-2, pp. 515–529, 2010.

[83] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al., "Apache spark: A unified engine for big data processing," Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016.

[84] "Aws athena pricing." `https://aws.amazon.com/en/athena/pricing`.

[85] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. I. Rowstron, "Rhea: Automatic filtering for unstructured cloud storage.," in USENIX NSDI'13, vol. 13, pp. 2–5, 2013.

[86] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," Proceedings of the VLDB Endowment, vol. 6, no. 11, pp. 1009–1020, 2013.

[87] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ mapreduce for log processing," in USENIX ATC'11, p. 115, 2011.

[88] "Apache beam." `https://beam.apache.org`.

[89] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," in IEEE MSST'05, pp. 119–123, 2005.

[90] "Amazon s3." `https://aws.amazon.com/en/s3`.

[91] "Amazon lambda." `https://aws.amazon.com/en/lambda`.

[92] "Ibm openwhisk." `https://developer.ibm.com/openwhisk`.

[93] "Google cloud functions." `https://cloud.google.com/functions`.

[94] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in ACM SoCC'17, 2017.

[95] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in ACM/IFIP/USENIX Middleware'17, 2017.

[96] "Crystal GitHub." `https://github.com/Crystal-SDS`.

[97] "PyActor." `https://github.com/pedrotgn/pyactor`.

[98] "Wikimedia downloads." `https://dumps.wikimedia.org/`.

[99] "Gridpocket iot data." https://github.com/gridpocket/project-iostack.

[100] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," ACM SIGMETRICS Performance Evaluation Review, vol. 24, no. 1, pp. 126–137, 1996.

[101] C. Chen, Y. Chen, and P. C. Roth, "Dosas: Mitigating the resource contention in active storage systems," in IEEE CLUSTER'12, pp. 164–172, 2012.

[102] "Apache Ignite." https://ignite.apache.org.

[103] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks.," in USENIX NSDI'15, vol. 15, pp. 293–307, 2015.

[104] E. Jahani, M. J. Cafarella, and C. Ré, "Automatic optimization for MapReduce programs," VLDB'11, vol. 4, no. 6, pp. 385–396, 2011.

[105] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari, "Cloud analytics: Do we really need to reinvent the storage stack," in USENIX HotCloud'09, 2009.

[106] E. H. Wilson, M. T. Kandemir, and G. Gibson, "Will they blend?: Exploring big data computation atop traditional hpc nas storage," in IEEE ICDCS'14, pp. 524–534, 2014.

[107] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the duality of data-intensive file system design: reconciling HDFS and PVFS," in ACM SC'11, p. 67, 2011.

[108] C. Xu, R. Goldstone, Z. Liu, H. Chen, B. Neitzel, and W. Yu, "Exploiting analytics shipping with virtualized MapReduce on HPC backend storage servers," IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 1, pp. 185–196, 2015.

[109] M. Mihailescu, G. Soundararajan, and C. Amza, "Mixapart: decoupled analytics for shared storage systems.," in USENIX FAST'13, pp. 133–146, 2013.