Journal of Internet Services and Applications

RESEARCH Open Access

SCOPE: self-adaptive and policy-based data management middleware for federated clouds

Ansar Rafique* , Dimitri Van Landuyt, Eddy Truyen, Vincent Reniers and Wouter Joosen

Abstract

A federated cloud storage setup which integrates and utilizes storage resources from multiple cloud storage providers has become an increasingly popular and attractive paradigm for the persistence tier in cloud-based applications (e.g., SaaS applications, IoT applications, etc).

However, federated cloud storage setups are prone to run-time dynamicity: many dynamic properties impact the way such a setup is governed and evolved over time, e.g., storage providers enter or leave the market; QoS metrics and SLA guarantees may change over time; etc. In general, existing federated cloud systems are oblivious to dynamic properties of the underlying operational environment, resulting in both sub-optimal data management decisions and costly SLA violations. Additionally, due to the sheer complexity of cloud-based applications coupled with the heterogeneous and volatile nature of federated cloud setups, the complexity of building, maintaining, and expending such applications increases dramatically and therefore managing them manually is no longer simply an option.

To address these concerns, we present SCOPE, a policy-based and autonomic middleware that provides self-adaptiveness for data management in federated clouds. We have validated SCOPE in the context of a realistic SaaS application, performed an extensive functional validation, and conducted a thorough experimental evaluation. The evaluation results demonstrate (i) the ability of the middleware to perform data management decisions that take into account the run-time dynamicity (i.e., dynamic properties) of a federated cloud storage setup to meet the promised SLAs, and (ii) the self-adaptive behavior of SCOPE without the need for operator intervention. In addition, our in-depth performance evaluation results indicate that the benefits are achieved with acceptable performance overhead, and as such highlight the applicability of the proposed middleware for real-world application cases.

Keywords: Self-adaptive, Data management, Middleware, Policy-driven, Multi-cloud storage, Multi-tenant SaaS, NoSQL, Monitoring, Federated cloud storage, Resource Provisioning, Elasticity

1 Introduction

Cloud computing has become a highly attractive paradigm due to its potential to significantly reduce costs through optimization and increase operating and economic benefits [1–4]. Therefore, nowadays, a growing number of applications (e.g., Software-as-a-Service (SaaS) applications, Internet of Things (IoT) applications, etc) take maximum advantage of the flexible services such as Storage-as-a-Service offered by the cloud to minimize the high up-front cost and optimize the overall maintenance cost [5]. As such, this enables service providers to offer enhanced services in a scalable and timely manner [6].

However, service providers find it difficult to select the best candidate when faced with numerous cloud storage providers (CSPs) and their underlying heterogeneous storage systems, as well as their different promised Service Level Agreement (SLA) guarantees. In addition, applications backed by a single cloud provider are highly subject to vendor lock-in, data unavailability, provider reliability, data security, etc [7–9]. Therefore, a federated storage cloud setup, which combines different storage resources and SLA guarantees from multiple cloud providers has become an increasingly popular tactic and proven practice

*Correspondence: Ansar.Rafique@cs.kuleuven.be
Department of Computer Science, KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium

Springer Open

© The Author(s). 2019 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

for designing the storage tier of cloud-based applications (e.g., SaaS applications, IoT applications, etc) [10–13].

In practice, federated cloud storage setups are subject to dynamicity, e.g., relevant dynamic properties are performance characteristics (i.e., latency and throughput), evolving price conditions, new providers arrival, cloud provider availability (i.e., uptime), etc. Despite this run-time dynamicity, data management decisions across a federated cloud storage setup are commonly based on the static properties of the operational environment (e.g., cloud storage system X always provides significantly better overall performance than the cloud storage system Y). However, solely relying on static properties leads to sub-optimal data management decisions in highly dynamic environments and may eventually results in costly SLA violations. Additionally, as applications increasingly rely on federated cloud storage setups to offer more advanced features, the complexity of managing such applications and their development life cycle also grows in parallel. In essence, managing and engineering such complex software applications requires careful planning, continuous monitoring, optimized configurations, and various other run-time operations. Due to the sheer complexity of such cloud-based applications coupled with the heterogeneous and volatile nature of federated cloud setups, managing and maintaining such complex applications manually is no longer simply an option.

To address the above-mentioned concerns, we propose a policy-based and autonomic middleware called SCOPE, which provides self-adaptiveness for data management in federated clouds. As such, SCOPE integrates three key desired features: (i) it continuously monitors the federated cloud setup and then collects, stores, and aggregates the monitored metrics (such as write latency, read latency, uptime, free memory, etc); (ii) it selects (based on the assembled metrics) the most suitable and appropriate cloud storage provider for data management decisions in order to meet the promised SLAs, and finally (iii) it autonomously reconfigures (based on the simple, reusable, and extensible configuration policies) the federated cloud storage setup (by identifying performance degradation) and therefore it is no longer required that an operator has to manually monitor and reconfigure the federated cloud storage setup.

We have validated SCOPE in the context of a realistic SaaS application, by implementing a prototype of a document processing SaaS application that runs on top of a federated cloud storage setup. We performed an extensive functional validation and also conducted a thorough experimental evaluation. The evaluation results demonstrate (i) the ability of the middleware to perform SLA-aware data management decisions by taking into account the run-time dynamicity of a federated cloud storage setup, and (ii) the self-managing behavior of SCOPE without the need for manual operator intervention. In addition, our in-depth performance evaluation results indicate that the benefits are achieved with acceptable performance overhead, which also demonstrate the applicability of the proposed middleware for real-world application cases.

This paper extends our previous work [14], in which we have introduced an initial version of the SCOPE middleware. In comparison, the main points of extensions and improvements are fourfold: (i) we provide a more detailed description of the middleware architecture and the underlying concepts of the SCOPE middleware, (ii) we validate SCOPE with a proof-of-concept prototype implementation, on top of which we have built an industrial SaaS application case, a Document Processing SaaS offering, (iii) we carry out an extensive functional validation and the overhead evaluation, and finally (iv) we extend our review of related work and provide a more detailed comparison with existing state-of-the-art and state-of-practice systems.

The remainder of this paper is structured as follows: Section 2 motivates the paper from the context of a realistic SaaS application and further outlines the key requirements of interest in this paper. Section 3 presents the overall architecture and describes the roles and responsibilities of different components of the SCOPE middleware. In Section 4, we briefly describe the prototype implementation. An in-depth functional validation of dynamic data placement and self-adaptive aspects of SCOPE as well as incurred performance overhead is presented in Section 5. Section 6 continues with a brief discussion about different choices being made and provides pointers to future work. Section 7 compares our approach with related work. Finally, Section 8 concludes the paper.

2 Motivation

The motivation for this paper is based on our frequent interaction and experiences with industry-level Software-as-a-Service (SaaS) providers in the context of a number of collaborative research projects [15–17]. The illustrative example for this paper –a document processing SaaS application– is introduced in Section 2.1. Section 2.2 subsequently highlights three scenarios that require active monitoring capabilities of changing conditions for making suitable decisions (including data placement and reconfiguration). Finally, Section 2.3 identifies the key requirements of interest for this paper.

2.1 Document processing SaaS application

The document processing SaaS application is a business-to-business (B2B) cloud offering that provides online services to its customer organizations (i.e., tenants) to generate, archive, and search a large collection of

customized digital documents (e.g., invoices, payslips, etc). Besides the functional requirements, the application also deals with a number of different non-functional requirements, especially with respect to performance, scalability, and availability, usually expressed in Service Level Agreements (SLAs). For example, the document processing SaaS application stores and process a large set of documents (e.g., monthly bills), usually at the end of each month. Therefore, high availability and elastic scalability of storage resources is particularly relevant during seasonal peaks (e.g., at the end of each month), whereas limited resources would suffice for the remaining period.

To address these concerns, as shown in Fig. 1, the document processing SaaS application combines resources from on-premise infrastructure (i.e., limited in terms of resources) with resources from external cloud storage providers (i.e., supporting elastic scalability and higher availability for seasonal spillover) in a federated cloud storage setup. This also enables service providers to employ a whole set of advanced storage tactics (e.g., auto scale, temporary spillover, cross-provider data replication, data migration, etc) in the application, based on the requirements to cope with seasonal peak periods. A spillover strategy is a temporary measure to address seasonal peaks. It refers to a situation when some of the tasks or jobs (e.g., data processing jobs) are dynamically moved from one place (e.g., on-premise storage infrastructure) to another (e.g., public clouds) to address peaks in load and also to avoid costly over provisioning. For example, in peak periods (e.g., at the end of the month for generating pay slips for employees), the Document Processing SaaS application dynamically spills over some processing jobs to the public cloud.

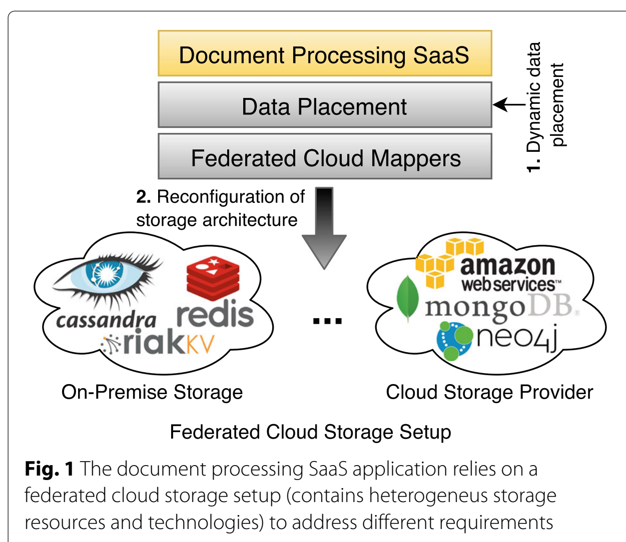


Fig. 1 The document processing SaaS application relies on a federated cloud storage setup (contains heterogeneus storage resources and technologies) to address different requirements

2.2 Scenarios

However, in reality, managing a federated cloud storage setup manually is a tedious task because such a setup involves heterogeneous resources from multiple clouds, which contain properties that may evolve quickly. For example, cloud storage providers (CSPs) may update their pricing policies or change their Quality-of-Service (QoS) guarantees. Consequently, managing such a setup requires continuous attention, careful planning, and appropriate manual intervention.

Table 1 illustrates a number of scenarios and the required interventions to realize them across a federated cloud storage setup, which is composed of two CSPs: CSP1 and CSP2. We elaborate further on three key scenarios that require continuous monitoring of changing run-time conditions of storage resources in a federated cloud setup.

Scenario #1: Performance optimization. Dynamic data placement is the strategy that involves decisions, which are in line with the changing conditions of cloud storage providers. In case of the document processing SaaS application, dynamic data placement decisions will

Table 1 An overview of federated cloud scenarios and their expected adaptation actions required to accomplish these scenarios

#	Scenarios/conditions	Adaptation actions
1	CSP2 outperforms CSP1 (i.e., performance optimization)	1. Change storage policy to use CSP2 for the future data storage requests (instead of CSP1)
		1.1. Keep existing data in CSP1, OR
		1.2. Migrate existing data from CSP1 to CSP2
2	CSP2 is suffering from ongoing performance issues in peak-load condition	1. Add more storage nodes in CSP2 (i.e., scale-out), OR
		2. Temporary spill-over to CSP1
3	CSP2 offers a discount and storage price drops below that of CSP1 (i.e., cost optimization)	1. Change storage policy to use CSP2 for the future data storage requests (instead of CSP1)
		1.1. Keep existing data in CSP1, OR
		1.2. Migrate existing data from CSP1 to CSP2
4	The SLA of CSP1 offers higher availability than that of CSP2	1. Use CSP1 for the data, which requires higher availability
		1.1. Keep existing data in CSP2, OR
		1.2. Migrate existing data from CSP2 to CSP1

entail selecting the most suited cloud storage provider by taking into account different performance characteristics of storage systems (static property), different performance profiles (dynamic property based on measurement and performance profiling/predictions). The first row of Table 1 illustrates the type of dynamically changing storage policy that accommodates such changes.

Scenario #2: Peak-load conditions. The document processing SaaS application deals with a wide variety of non-functional requirements with respect to performance and availability. However, with respect to the requirement of sending out the invoices, the application experiences congestion during peak periods (i.e., usually at the end of each month). To accommodate such peak periods where a large number of users connect simultaneously to the SaaS application, an adaptation of the storage architecture involves dynamically and temporarily including spillover resources (see #2 in Table 1 for possible adaptation actions).

Scenario #3: Cost optimization. Pricing schemes differ across CSPs and evolve over time. Consider a scenario where a new cloud storage provider enters the market that offers more cost-efficient data storage or temporary discounts. As the cost optimization is one of the core objectives of SaaS applications, there is a clear incentive to dynamically extend the storage architecture by incorporating the new provider and maximally utilizing its storage resources (see #3 in Table 1 for possible adaptation actions).

2.3 Requirements

The prime objective of the document processing SaaS application is to make data management decisions (including data placement) that are consistent with the operational environment (see **1.** in Fig. 1) and also autonomously perform actions (e.g., reconfiguration of storage architecture; implementation of advanced storage tactics such as temporary spillover, data migration, etc), eliminating the need for human intervention (see **2.** in Fig. 1).

Therefore, to achieve the above general objective, the following functional and qualitative requirements must be addressed and satisfied by the application:

R1 - Dynamic data placement. In the case of peak periods, e.g., when a data center of a cloud provider is confronted with heavy workloads, the application must determine whether switching the cloud provider yields better performance. If so, dynamically switching between cloud storage providers based on workloads in order to meet the specified Service-Level-Objective (SLO) requirements for different operations. Consequently, data management decisions based on dynamic properties will yield significant improvement in performance in comparison to the decisions based solely on static properties.

R2 - Self-adaptive behavior. Similar to **R1**, situations where the cloud provider is faced with heavy workloads, which contribute significantly to performance degradation, the application must identify the dynamically changing conditions in the operating environment (i.e., federated cloud setup) and perform actions autonomously (see **2.** in Fig. 1). For example, identifying the situations when a scale-up or scale-out is required and then performing actions accordingly.

R3 - Abstract complexity of self-adaptiveness. Federated cloud storage setups are typically heterogeneous and dynamic and the supported features are often incompatible. This diversity and dynamicity hinders the proper exploitation of the full potential of a federated cloud storage paradigm as it increases the complexity of the administration of applications. For example, providing the self-adaptiveness support (**R2**) on top of a federated cloud storage setup introduces another level of complexity in the application code. This involves writing reconfiguration logic on top of the federated cloud storage setup (e.g., to support advanced storage tactics such as auto scaling, temporary spillover, data migration, etc) in the application code. This complexity must be abstracted and concealed from the application developers to the operators and thus enabling them to employ different advanced storage tactics without invasively extending application code.

R4 - Acceptable overhead. The above-mentioned requirements must be addressed in a feasible manner, i.e., introduce only a small and negligible overhead.

However, addressing these requirements in the application leads to continuous monitoring, careful planning, optimized configurations, and various other run-time operations. This further introduces additional complexity and therefore managing this increased complexity manually in the application is not practically feasible and striving for self-adaptiveness support.

The next section presents the architecture of the SCOPE middleware and its key components and further, it discusses how the document processing SaaS application can effectively leverage the potential of the SCOPE middleware to address the above-mentioned requirements.

3 SCOPE: a self-adaptive middleware

Dynamically adapting the system behavior requires an architecture that provides active monitoring capabilities and also supports (re)configuration at run time.

Figure 2 depicts the organization of the major components in the SCOPE architecture. We have designed SCOPE around two fundamental principles. The first principle is that the data management decisions (including data placement) are executed by taking dynamic properties into account, and thus these are compatible with the underlying federated cloud storage setup (the `Data`

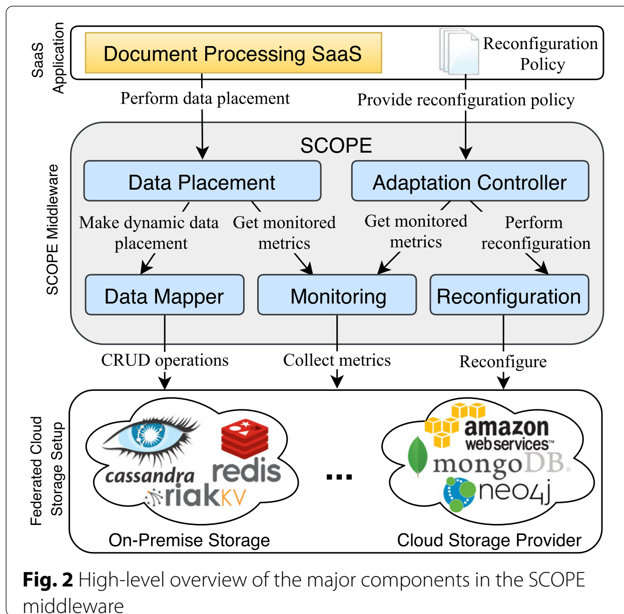


Fig. 2 High-level overview of the major components in the SCOPE middleware

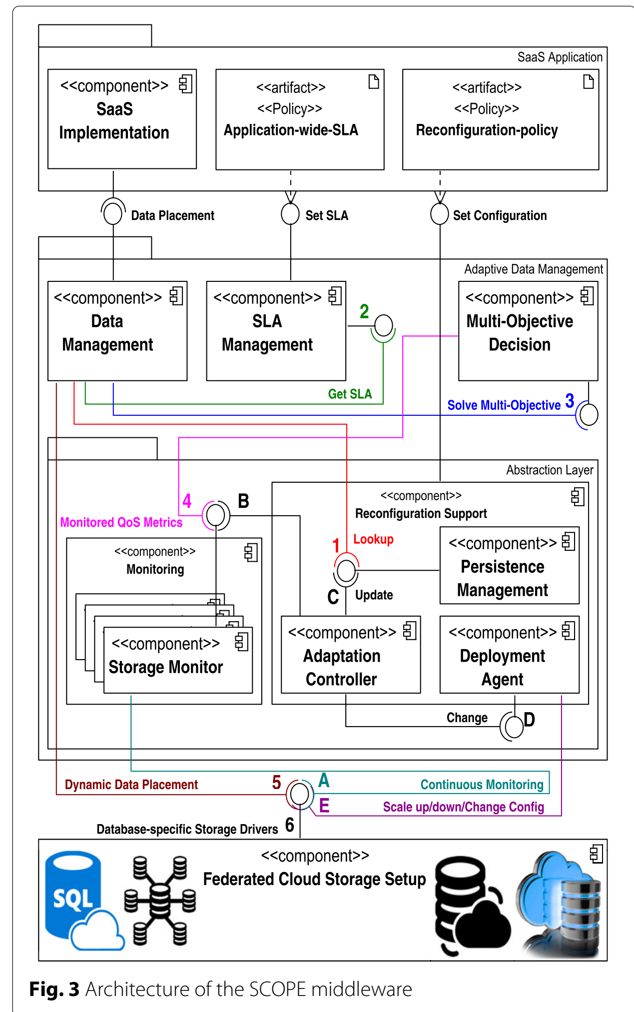


Fig. 3 Architecture of the SCOPE middleware

`Placement` component in Fig. 2). The second principle is that the reconfiguration of the underlying federated cloud storage setup is accomplished using external and reusable reconfiguration policies, and thus the complexity of performing reconfiguration is abstracted and externalized from the application (the `Adaptation Controller` component in Fig. 2). To provide a brief overview, the `Monitoring` component periodically collects metrics/statistics of the underlying federated cloud storage setup and stores them in the cache/database. These metrics are then used by (i) the `Data Placement` component to make dynamic data placement decisions, and (ii) the `Adaptation Controller` component to support adaptation actions (e.g., scale up, scale out, temporary spillover, etc).

The detailed architecture of the SCOPE middleware is presented in Fig. 3, which consists of three layers. From top to bottom they are: (i) the *SaaS Application* layer, (ii) the *Adaptive Data Management* layer, and (iii) the *Federated Cloud Storage Setup* layer. In the following subsections, we will describe in detail the different layers and components of SCOPE as well as discuss how the three key requirements (**R1** - **R3**) listed above are addressed.

3.1 SaaS application

Presented at the top of Fig. 3, the *SaaS Application* layer provides application-wide configuration and customization (e.g., reconfiguration policies and SLAs specifications). SaaS providers are given the ability to specify `Application-wide-SLA`, which is a declarative model-based description of different SLA requirements (including different QoS metrics) that the application has to satisfy. In addition, SaaS providers can also specify the `Reconfiguration-policy`, which is defined as a set of rules of which each contains a set of conditions based on specific target variables and an action part. The `Reconfiguration-policy` enables service providers to specify the reconfiguration rules for the underlying federated cloud storage setup without writing the reconfiguration logic in the application code (addresses **R3**).

3.2 Adaptive data management

The core of the SCOPE middleware is the *Adaptive Data Management* layer (positioned in the center of Fig. 3), which we describe in detail in the rest of this section. We mainly focus on the roles and responsibilities of different components of the *Adaptive Data Management* layer and how they efficiently and flexibly support the remaining requirements (**R1** and **R2**) discussed in the previous section.

The *Adaptive Data Management* layer provides adaptation capabilities for responding to changes at

run time and meeting different SLAs requirements specified by the application. The layer is comprised of five main components: (i) the `Data Management` component, (ii) the `SLA Management` component, (iii) the `Multi-Objective Decision` component, (iv) the `Monitoring` component, and (v) the `Reconfiguration Support` component as shown in Fig. 3.

Data Management Component. The dynamic data management decisions in a federated cloud storage setup must take into account the different requirements of the application, expressed in the `Application-wide-SLA`. In a simplistic case (e.g., storing invoices), multiple data stores can be elected as suitable candidates for data placement decisions. However, the question remains which properties must be considered to efficiently select the data stores for data placement decisions. In order to make this decision efficiently, the application requirements (expressed in the `Application-wide-SLA`), current state, and dynamic properties of cloud storage providers and their respective storage systems must be taken into consideration. There are a number of dynamic properties (e.g., cloud storage provider availability, performance, evolving price conditions, etc), which are supported and can be considered by SCOPE for data management decisions. However, in this paper, we mainly focus on the performance aspect of the cloud storage provider simply because this aspect is not yet sufficiently reflected in existing state-of-the-art federated cloud systems [7, 12, 18–21]. To perform data placement decisions that are compatible with the operational environment (i.e., underlying federated cloud storage setup), the `Data Management` component gets the persistence configuration details of different storage systems, distributed across multiple cloud storage providers from the `Persistence Management` component (Step 1 in Fig. 3) and the application-specific SLA requirements from the `SLA Management` component (Step 2 in Fig. 3) and passes the information to the `Multi-Objective Decision` component (Step 3 in Fig. 3). The latter component is responsible for making appropriate optimization decisions (i.e., selecting suitable candidates for data placement decisions), taking into account the different requirements of the application. The `Data Management` component, then performs data placement operations that are consistent with the operational environment (Step 5 in Fig. 3), based on the returned information from the `Multi-Objective Decision` component about the most suited cloud storage providers (addresses **R1**).

SLA Management Component. The `SLA Management` component stores SLA requirements specified by the application. The component exposes an interface that allows the application to specify SLAs, which usually are expressed in terms of different optimization objectives (e.g., performance, cost, availability, etc). Listing 1 shows an example of the SLA agreement for the document processing SaaS application. The SLO parameter values (i.e., SloID) show the quality of service required by the document processing SaaS application (e.g., response time for write operations, response time for read operations, uptime, etc). Furthermore, we also define the threshold values that guide the enforcement of these SLAs. The SLA agreement of the document processing SaaS application combines performance and availability as follows: request response time should not exceed 10 ms and atleast 97% of requests should be served.

Listing 1: An example of SLA agreement for the document processing SaaS application.

```
<slas>
  <sla serviceID = "Document-Processing">
    <slo sloID="RTW" operation="write"
    metric="response time" unit="millisecond"
    comparator="LT" threshold="10" />

    <slo sloID="RTR" operation="read"
    metric="response time" unit="millisecond"
    comparator="LT" threshold="10" />

    <slo sloID="AV" metric="availability"
    unit="% of requests" comparator="GT"
    threshold="97" />
  </sla>
</slas>
```

Multi-Objective Decision Component. The dynamic data placement decisions are influenced by many run-time factors. For example, SLA requirements, dynamic conditions of a federated cloud storage setup (e.g., performance, availability, cost, etc), and individual requirements for a certain data type may influence the decision. To make data placement decisions that are consistent with the underlying federated cloud storage setup, as part of Step 4 in Fig. 3, the `Multi-Objective Decision` component sends a request to the `Storage Monitor` sub-component of the `Monitoring` component, which continuously monitors different QoS metrics (see Listing 2 as an example of monitored QoS metrics such as write latency, read latency, and uptime) and sends the response back (i.e., monitored metrics) to the `Multi-Objective Decision` component. The `Multi-Objective Decision` component compares the monitored metrics (i.e., the QoS) with the expected performance SLAs specified by the SaaS application (see Listing 1 for an example of expected SLA policy for the document processing SaaS application). For example, as shown in Listing 2, three data stores on potentially varying cloud providers (i.e., Cassandra-Private, Cassandra-Public, and MongoDB-Private) satisfy the imposed SLA requirements for storing invoices. The

`Multi-Objective Decision` component based on the SLA requirements and monitored QoS metrics, making optimization decisions as such, selects the most efficient cloud provider suited for data storage.

Monitoring Component. Monitoring, in general, is a key component in database administration that enables service providers to gain an in-depth visibility into different metrics to optimize their data infrastructure. Thus, it also allows service providers to diagnose issues (e.g., related to health, performance, availability, etc) in their data infrastructure and then plan and implement appropriate change process.

In SCOPE, the `Monitoring` component is responsible for monitoring different QoS metrics of back-end storage systems operating at different cloud providers. Table 2 shows the list of supported monitoring metrics for different cloud storage systems (including both relational and NoSQL databases). An example of monitored QoS metrics (i.e., write latency, read latency, and uptime) for cloud storage technologies operating at different cloud providers is shown in Listing 2.

Listing 2: Monitored QoS metrics for storage technologies operating at different cloud providers.

```
                          Latency        Availability
Cloud  Provider      Write       Read       Uptime
Cassandra-Private     2ms        2ms         99%
Cassandra-Public      3ms        4ms         98%
Mongodb-Private       3ms        3ms         99%
Mongodb-Public        11ms       13ms        90%
...                   ...        ...         ...
```

This component exposes an interface to retrieve the monitored QoS metrics information. The `Storage Monitor` component continuously monitors QoS metrics (Step A in Fig. 3) and stores up-to-date QoS metrics in the database. The up-to-date monitored QoS metrics are accessed by different components of the *Adaptive Data Management* layer for different purposes. For example, as stated above, to perform data management decisions that are consistent with the operational environment (i.e., the underlying federated cloud storage setup), the up-to-date QoS metrics are accessed by the `Multi-Objective Decision` component. Similarly, to autonomously (re)configure the federated cloud storage setup or to be able to react to unusual demand situations (e.g., add more nodes, change the replication factor, etc), the monitored QoS metrics information is also accessed by the `Adaptation Controller` sub component of the `Reconfiguration Support` component.

Reconfiguration Support Component.

The `Reconfiguration Support` component provides an interface to set configuration details, and performs an initial deployment and configuration of heterogeneous storage systems distributed across multiple clouds. The component is comprised of three sub components (i) the `Persistence Management` component, (ii) the `Adaptation Controller` component, and (iii) the `Deployment Agent Service` component.

Table 2 Summary of the white-box metric types supported in the SCOPE middleware for different cloud storage technologies (including both NoSQL and relational database technologies)

	NoSQL Databases			RDBMS
Metric type	Cassandra	MongoDB	Redis	PostgreSQL
Read latency	⊠	⊠	⊠	
Write latency	⊠	⊠	⊠	
Uptime	⊠	⊠	⊠	⊠
Average object size	⊠	⊠	⊠	⊠
Total object size	⊠	⊠	⊠	⊠
Connected clients		⊠	⊠	⊠
Total connections		⊠	⊠	⊠
Keycache capacity	⊠			
Keycache hitrate	⊠			
Keycache size	⊠			
Memory allocated	⊠	⊠	⊠	
Memory used	⊠	⊠	⊠	
Rowcache capacity	⊠			
Rowcache hitrate	⊠			
Rowcache size	⊠			
Read request count	⊠	⊠	⊠	
Write request count	⊠	⊠	⊠	
Total object count	⊠	⊠	⊠	⊠

The `Persistence Management` component contains the persistence configuration details of different storage systems and for that, the component provides an interface to lookup and update these details.

The process of expansion and contraction of resources to cope with various ongoing performance, scalability, and availability issues or to even address the contradicting requirements of applications requires runtime elasticity. The run-time elasticity in the context of a cloud storage system (e.g., NoSQL system) is defined as the ability of the system to increase or decrease the storage or computing resources (e.g., virtual CPU cores, memory, etc) in response to the changing workload.

Although NoSQL systems are designed to be elastic, they are however not autonomously elastic, which means external components are specifically required for making decisions (such as when to increase or decrease the resources) and thereby taking appropriate actions (e.g., scaling up/down or scaling out/in). The component responsible to provide such a service is the `Adaptation Controller` component. The `Adaptation Controller` component is responsible

for the management of resources and triggers an appropriate action (e.g., install new instances to a database cluster, change the replication factor, etc) if the resources are suffering from various ongoing issues or if the unusual demand situations have occurred. The component reads the up-to-date monitored QoS metrics (Step B in Fig. 3) from the `Storage Monitor` sub component of the `Monitoring Component`, which provides continuous monitoring capabilities. The `Adaptation Controller` component contains a number of (re)configuration rules and based on these rules and the monitored QoS metrics, and finally make effective decisions. For example, Listing 3 shows the (re)configuration rule to keep the average latency of the Cassandra storage system below 30 ms. The appropriate (re)configuration action (i.e., adding a new storage node), specified in the (re)configuration action executes in case of service violation. To perform such an action, as part of Step D in Fig. 3, the component dispatches a notification change signal to the `Deployment Agent` component, which is responsible for providing the needed deployment support. Another example of service violation is to give an indication that the system (e.g., Cassandra, MongoDB, etc) is running out of storage or memory. In such a case, the `Deployment Agent` component acts to increase the system memory or storage.

Listing 3: The (re)configuration rule to keep the average latency of Cassandra below 30 ms. Add more nodes in case of service violation.

```
rule "Keep the average latency below 30 ms"
  when
  sla: SlaMeasurement(storage == "Cassandra"
    and latency > 30)
  then
  sla.setExtraNodeRequired(true)
end
```

The `Deployment Agent` component is responsible for performing the desired (re)configuration and deployment autonomously (Step E in Fig. 3) including: adding more nodes in a cluster (i.e., scale out), remove nodes from the cluster, changing consistency level, increase system memory and storage, etc (addresses **R2**).

3.3 Federated cloud storage setup

The *Federated Cloud Storage Setup* layer provides a uniform application programming interface (API) which underneath consists of a number of storage-specific drivers for different storage systems operating at different cloud storage providers. This layer is responsible to handle a number of requests originated from different components of the *Adaptive Data Management* layer (e.g., the `Data Management` component, the `Storage Monitor` component and the `Deployment Agent` component) and uses the right database-specific storage driver to perform an operation.

4 Prototype implementation on SCOPE

In this section, we describe the prototype implementation of SCOPE that follows the reference architecture presented in Section 3.

The proposed middleware is validated in a prototype, which is built upon and includes a number of open source tools and technologies. To address the problem of lack of standardization (i.e., each NoSQL technology exposes a different interface, different data model, and different API) and generalization (i.e., NoSQL databases exhibit an additional phenomenon: they are specialized solutions, usually tailored to specific use cases, and address the specific storage requirements), we have used Impetus Kundera [22]. The prototype is implemented on top of the Kundera platform, which is an open-source and a Java-based abstraction API for a wide range of NoSQL data stores (e.g., Apache Cassandra, MongoDB, HBase, Redis, Neo4J, etc). In our current implementation, Kundera is considered as an important foundation as it facilitates SCOPE to communicate with the data stores in a uniform way and also introduces minimal performance overhead, compared to existing abstraction APIs [3, 23, 24]. In addition to the wide range of data stores (including in-memory, SQL-based, NoSQL-based, and full-text search) supported by the abstraction API, the prototype also makes use of additional technologies such as Ehcache[1], Apache Lucene[2], etc. Ehcache is an open-source and a Java-based distributed cache, whereas Apache Lucene is a high-performance, full-featured text search engine library written in Java. Although the prototype was implemented on top of a specific data store version, the prototype also works with older and newer versions.

Similarly, to address the challenges related to monitoring a federated cloud storage setup —where different monitoring APIs need to be combined and used, which significantly complicates monitoring efforts and leads to increased development complexity—, we have created an extensible and simple-to-use abstraction API for monitoring. The API underneath uses different technology-specific APIs for monitoring different NoSQL technologies operating at different cloud storage providers. Even in the case of monitoring a single NoSQL technology, the API combines and uses multiple internals and externals APIs as well as utilities to assemble different monitoring metrics. For example, to monitor the health status of nodes (e.g., active or inactive nodes) within the Cassandra cluster, an internal *nodetool*[3] utility can be used, whereas an external monitoring API such as *JConsole*[4], a Java Management Extensions (JMX)-compliant API must be employed to monitor the memory consumption in Cassandra.

The policy execution engine of SCOPE was developed on top of JBoss Drools[5], an open source and an object-oriented rule engine written in Java. The prototype provides a Java Persistence API (JPA) and a Java Persistence Query Language (JPQL) as a uniform API to communicate with the supported data stores. Similarly, entities metadata, multi-cloud persistence configuration files, policy files, and the policy evaluation decisions are cached stored in Ehcache.

The prototype itself is also implemented in Java and operates as a service on Tomcat 8[6] with an exposed configuration dashboard for SaaS providers and their tenants. After the service starts up, it first reads the federated cloud persistence configuration file and the reconfiguration policy file. The federated cloud persistence configuration file includes the configuration details (such ip addresses, port, keyspace, credentials, etc) of the back-end storage systems used in the federated cloud setup. Similarly, the reconfiguration policy file contains a number of rules to determine when to perform a reconfiguration action (e.g., scale up or scale down). After setting these files, the monitoring and the reconfiguration services (i.e., the `Monitoring` component and the `Adaptation` Controller component) of the prototype start up and run continuously run in the background.

5 Evaluation

We performed a functional validation of SCOPE and also conducted several extensive performance evaluation experiments. The experimental setup and specific goals for each experiment are detailed throughout the following subsections.

More precisely, in Section 5.1, we first describe the application setups and then discuss the different deployment setups in which we tested SCOPE along with details on software and hardware. Then, in Section 5.2 we report on the functional validation of SCOPE in terms of dynamic data placement decisions (validate **R1** discussed in Section 2.3). Section 5.3 evaluates the impact of the SCOPE middleware in terms of additional performance overhead introduced (evaluate **R4** discussed in Section 2.3). Section 5.4 subsequently presents the functional validation of the self-managing behavior of SCOPE (validate **R2** discussed in Section 2.3).

5.1 Application/experimental setup

To quantify and evaluate the performance of SCOPE and to achieve a more accurate functional verification, we implemented two application prototypes doing the same CRUD[7] operations. These application prototypes were implemented with and without monitoring and reconfiguration capabilities. To validate our approach, in all settings, we compare the performance of prototype SCOPE (an application prototype of the document processing SaaS built on top of SCOPE) with the performance of prototype SCOPE^{-MR}, an application prototype of the document processing SaaS that implements all components of SCOPE, but without the monitoring and the reconfiguration capabilities enabled. The experiments were executed for CRUD operations, containing the data size of 100 K entries. The application that is acting as a client (i.e., running on a client node) is executed 3 times for each experiment before calculating the average values.

Both application prototypes use a federated cloud architecture, which reflects a realistic deployment configuration. The architecture is comprised of (i) the Cassandra-X deployment setup contains 2 nodes Apache Cassandra cluster (stable version 3.11.1) in which one node is active and running, whereas the other node is inactive (i.e., in the standby mode); (ii) the MongoDB-X deployment setup includes a single node MongoDB service (stable version 3.4.9); (iii) the Cassandra-Y deployment setup consists of a single node Apache Cassandra service (stable version 3.11.1); and (iv) the MongoDB-Y deployment setup contains a single node MongoDB service (stable version 3.4.9). All these services are set to the standard settings, deployed and managed in a private IaaS cloud using OpenStack[8].

During experiments, all nodes of the deployment setups (i.e., hosting databases) are deployed separately with the same specification to allow for a fair comparison. The machines running each deployment setup have an Intel(R) 4 Core @ 2.60 GHz processor, 8 GB RAM and is hosted on a compute node of OpenStack. The compute node consists of 40 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz processor with 120 GB RAM and runs the Linux/Ubuntu operating system. Additionally, we use one client node (i.e., running benchmarking application and acting as a client) to interfere with the deployment setups to perform CRUD operations. However, to ensure that the benchmarking application does not affect the performance of the deployment setups (running databases), it is hosted on a dedicated machine that is not a part of the deployment setups. The client node is equipped with Intel(R) Core(TM) i5 @ 2.60 GHz (Dual) processor with 8 GB RAM and Windows 8 installed.

Beyond these experiments, we also have executed some additional performance benchmarks as confirmatory runs where we have increased the workloads and used other deployment setups (e.g., MongoDB-X and MongoDB-Y) and the responses were again measured. These additional benchmarks lead to the same conclusions as the ones we derive later in the following section.

In order to generate the load and stress deployment setups with write-heavy and read-heavy workloads, we have used the standard Yahoo! Cloud Serving Benchmark (YCSB) [25] and the cassandra-stress tool[9]. YCSB is one

of the most popular benchmarking framework and provides a means to stress multiple databases and compare them in a fair, consistent, and effective manner. It provides data generator and a variety of workloads that are defined as a set of CRUD operations. Similarly, to stress Cassandra deployment setups, the cassandra-stress tool is used as a cassandra-specific stress testing utility for basic benchmarking and load testing.

5.2 Functional validation of dynamic data placement

As a functional validation, we illustrate the importance of continuous monitoring capability of SCOPE for making dynamic data placement decisions as well as for enforcing different performance SLA guarantees. In Section 5.2.1, we first describe the metrics that are fundamental to the experiment. Then, results are presented in Section 5.2.2.

5.2.1 Metrics

We investigate the impact of dynamic data placement (i.e., taking dynamic properties such as run-time performance into consideration) on the write completion time (WCT) and the read completion time (RCT). In particular, the impact of prototype SCOPE and prototype SCOPE^{-MR} on the WCT and the RCT when the deployment setups are stressed with write-heavy and read-heavy workloads respectively, is studied.

Write completion time (WCT). Write completion time denotes the time it takes to perform the write operation. According to the SLA agreement defined for the document processing SaaS application (see Listing 1), the response time for each write operation should always be in the range of 10 ms (cf. line #5 in Listing 1 for write response time threshold value). In this experiment, both application prototypes (prototype SCOPE and prototype SCOPE^{-MR}) are configured to use the Cassandra-X deployment setup for the write operation. Then, while performing the experiment (i.e., inserting 100 K entries), the Cassandra-X deployment setup is stressed with a write-heavy workload for both application prototypes using the standard benchmarking tools (i.e., YCSB and cassandra-stress) discussed in the previous section. The primary objective of this experiment is to illustrate the usefulness of dynamic data placement in terms of making decisions that are consistent with the run-time dynamicity of a federated cloud storage setup (i.e., decisions based on dynamic properties) and therefore meeting SLA requirements. Furthermore, the goal is also to determine the changes in the data placement decisions (i.e., data storage) on both application prototypes, caused by the stress of a write-heavy workload.

Read completion time (RCT). Read completion time represents the time it takes to perform the read operation. The response time for each read operation should also be in the range of 10 ms (cf. line #9 in Listing 1 for read response time threshold value). In this experiment, data is first replicated across both deployment setups (i.e., Cassandra-X and Cassandra-Y) with strong cross-provider data consistency. The strong data consistency guarantees that the write operation is considered as executed successfully when data is replicated in both deployment setups. Therefore, performing a read operation from any of the deployment setup will always return consistent data. As data is consistent in both deployment setups, application prototypes (prototype SCOPE^{-MR} and prototype SCOPE) can access data from any of the deployment setup to facilitate the read operation. Then, while executing read operations (i.e., reading 100 K entries), both Cassandra-X and Cassandra-Y deployment setups are stressed with a read-heavy workload, however, at different time intervals. Again, the goal is to examine the impact of stress that results from heavy workloads on the deployment setups and then observe the nature of the behavioral changes on these application prototypes.

5.2.2 Results

The results of these experiments are presented in Fig. 4 for the WCT and Fig. 5 for the RCT. The x-axis represents the latency in milliseconds (ms) and the y-axis represents the frequency of write operations (ops/ms).

As shown in Fig. 4a, write operations for prototype SCOPE^{-MR} (i.e., without monitoring and reconfiguration capabilities) are executed between 2 and 70 ms in which the majority of write operations are executed between 2 and 40 ms. On the other hand, the majority of write operations for prototype SCOPE (i.e., with monitoring and reconfiguration capabilities enabled) are executed between 2 and 4 milliseconds (see Fig. 4b). Similarly, as illustrated in Fig. 5a for the RCT, read operations are executed between 4 and 14 ms for prototype SCOPE^{-MR} (i.e., without monitoring and reconfiguration capabilities), whereas a large number of read operations are executed between 3 and 6 ms (see Fig. 5b) for prototype SCOPE.

In addition, the maximum latency of the write operation and the read operation for prototype SCOPE is always less than 10 ms (i.e., prototype SCOPE meets the SLO requirements specified in line #5 for WCT and line #9 for RCT of the Listing 1).

The dynamic behavior of these deployment setups when the stress with write-heavy and read-heavy workloads is increased, depicted in Figs. 6 and 7 respectively. The latency values presented in Figs. 6 and 7 (on the y-axis) are directly accessed from the monitoring component and therefore are lower than the actual latency values when accessed from the application prototypes[10].

As shown in Fig. 6 for the WCT, initially prototype SCOPE uses the Cassandra-X deployment setup for write

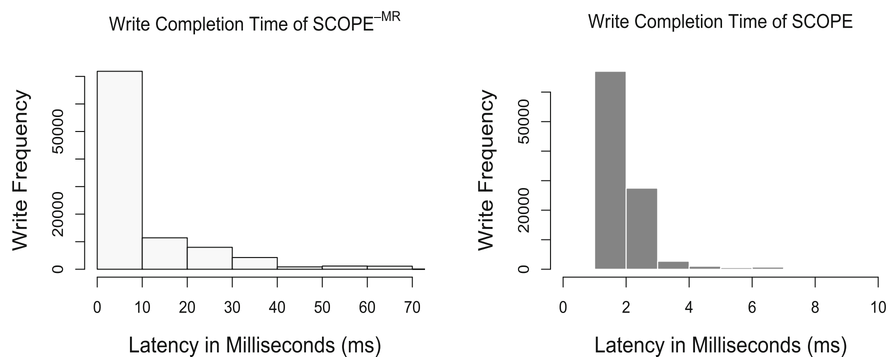
operations and the Cassandra-Y deployment setup is inactive (i.e., the write latency of the Cassandra-Y deployment setup is 0). Then, when the Cassandra-X deployment setup is stressed with the write-heavy workload (starting at the time scale of 20 min) and the latency has increased from 0.02 to 0.05 ms (at time interval between 30 and 38 min), which exceeds the specified threshold value (i.e., 0.04 ms), the SCOPE middleware observes this behavior of the deployment setup and starts redirecting all write operation requests to the Cassandra-Y deployment setup (satisfy **R1** discussed in Section 2.3). However, as shown in Fig. 6, the latency of the Cassandra-X deployment setup keeps on increasing even after SCOPE switches to the Cassandra-Y deployment setup. The increase is mainly caused by a number additional processes started to generate the write-heavy workloads on the Cassandra-X deployment setup. These additional processes are not stopped immediately after SCOPE switches to the Cassandra-Y deployment setup. As a result, the Cassandra-X deployment is stressed with write-heavy workloads even after SCOPE switches to the Cassandra-Y deployment setup.

Similarly, Fig. 7 shows the behavior of prototype SCOPE for the RCT. As for the RCT, data is replicated across the Cassandra-X and the Cassandra-Y deployment setups and is consistent, prototype SCOPE reads the data from the deployment setup, which provides lower read latency. As we can see that initially the latency of the Cassandra node deployed on the Cassandra-X deployment setup (i.e., 2.5 ms) is lower than the latency of the Cassandra node, deployed on the the Cassandra-Y deployment setup (i.e., 3.5 ms), prototype SCOPE reads data from the Cassandra-X deployment setup. However, at 10 min, the Cassandra-X setup is slowly stressed with the read-heavy workload. The read latency of the Cassandra node deployed on the Cassandra-X deployment setup has increased from 0.2 ms upto 3 ms starting at 20 min and is become higher than the read latency of the Cassandra node deployed on the Cassandra-Y deployment setup, SCOPE starts using the Cassandra-Y deployment setup to perform read requests (at the time scale of 20 min). Then, when the Cassandra-Y deployment setup is stressed with the read-heavy workload (at 25 min) and the read latency of the Cassandra node deployed on the Cassandra-Y deployment setup becomes higher than the read latency of the Cassandra node deployed on the Cassandra-Y deployment setup, the SCOPE middleware, at 30 min switches back to start using the Cassandra-X deployment setup (satisfy **R1** discussed in Section 2.3).

The evaluation results indicate that generally the monitoring approach adopted by the SCOPE middleware yields lower write and ready latencies (see Figs. 4b and 5b) and exhibits better performance (see Table 3) compared to prototype SCOPE^{-MR}, which does not support monitoring of dynamic and rapidly changing properties (e.g., performance, uptime, etc).

This significant improvement in performance is mainly due to the adaptive ability of our proposed middleware. SCOPE continuously monitors the changing properties (e.g., performance in this specific case) of all deployment setups and then data management decisions are made based on these monitored metrics. For example, if the heavy workloads increase the latency on one deployment setup, SCOPE detects this behavior and in case of detection automatically redirects the operation requests (i.e., CRUD) to the other available deployment setup (that introduces lower latency) to limit the increase in the latency and avoid costly SLA violations.

As shown in Table 3, the total execution time prototype SCOPE^{-MR} takes to process the data size of 100



a The SCOPE^{-MR} prototype takes between 1 and 70 ms to process write operations. The majority of write operations are performed between 1 and 40 ms.

b The SCOPE prototype takes between 1 and 8 ms to process write operations. The majority of write operations are performed between 2 and 4 ms.

Fig. 4 The frequency of 100 K write operations for **a** prototype SCOPE^{-MR} (without monitoring and reconfiguration) and **b** prototype SCOPE (with monitoring and reconfiguration)

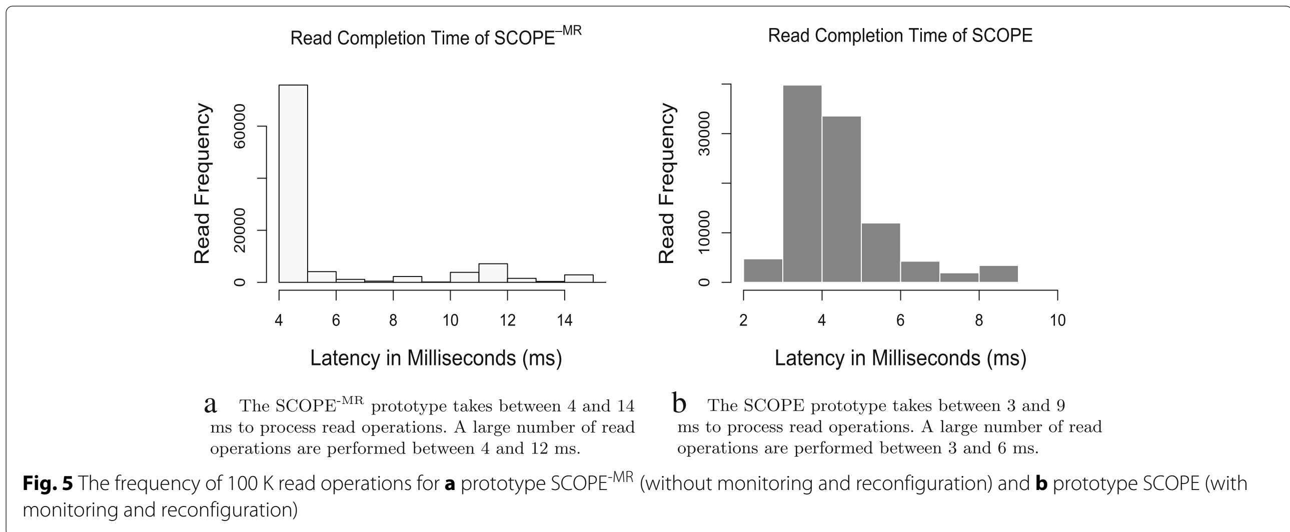


Fig. 5 The frequency of 100 K read operations for **a** prototype SCOPE^{-MR} (without monitoring and reconfiguration) and **b** prototype SCOPE (with monitoring and reconfiguration)

K entries for the WCT is 1012 s, whereas prototype SCOPE takes only 322 s to process the same data size. This corresponds to an overall performance improvement of 212% for write operations. Similarly, to process the data size of 100 K entries for the RCT, prototype SCOPE^{-MR} takes 656 s and prototype SCOPE takes 501 s as the total execution time. This contributes to the overall performance improvement of 40% for read operations (satisfy **R2** discussed in Section 2.3). These results also demonstrate that data management decisions, which are based on dynamic properties (i.e., run-time performance of deployment setups) are also in line with the run-time dynamicity of a federated cloud storage setup than the decisions based on static properties.

5.3 Performance overhead

In this section, we evaluate the performance overhead of the SCOPE middleware, caused by providing the continuous monitoring capability, taking dynamic properties into account for data management decisions (including data placement), and performing the reconfiguration action. Section 5.3.1 provides details about the experimental setup, while the results are summarized in Section 5.3.2.

5.3.1 Setup

The experiments are conducted to evaluate the cost in terms of the introduced overhead. To investigate the impact, the measurement interval t is used, where the value of t is set to 10 s, which means that after every 10 s,

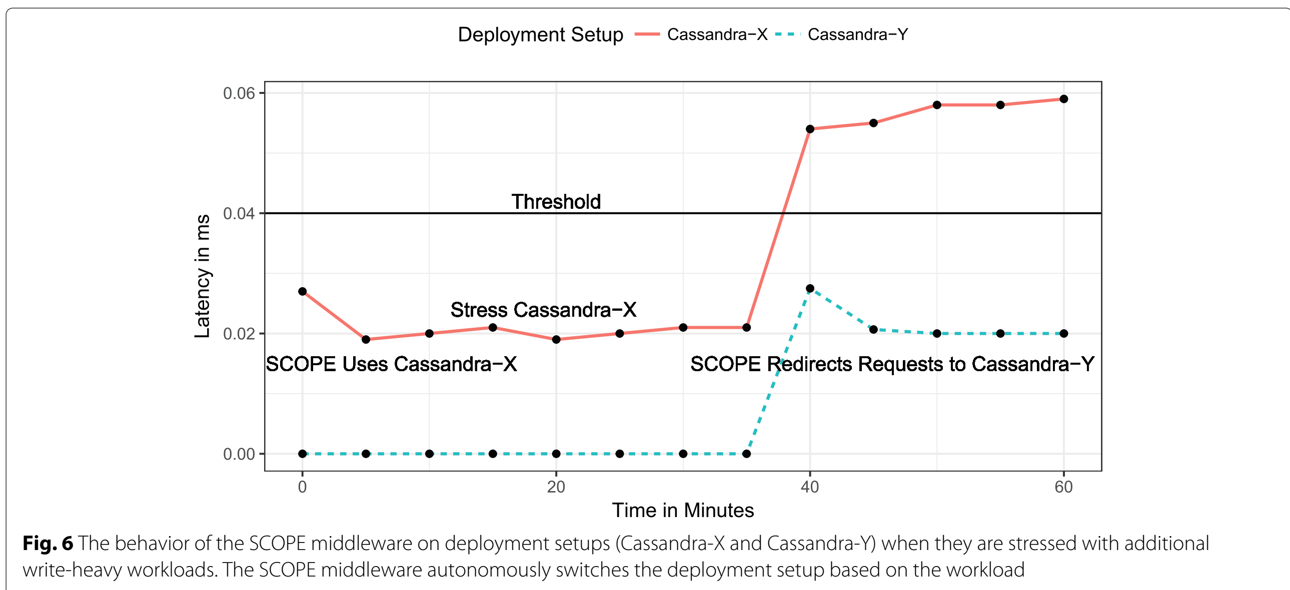


Fig. 6 The behavior of the SCOPE middleware on deployment setups (Cassandra-X and Cassandra-Y) when they are stressed with additional write-heavy workloads. The SCOPE middleware autonomously switches the deployment setup based on the workload

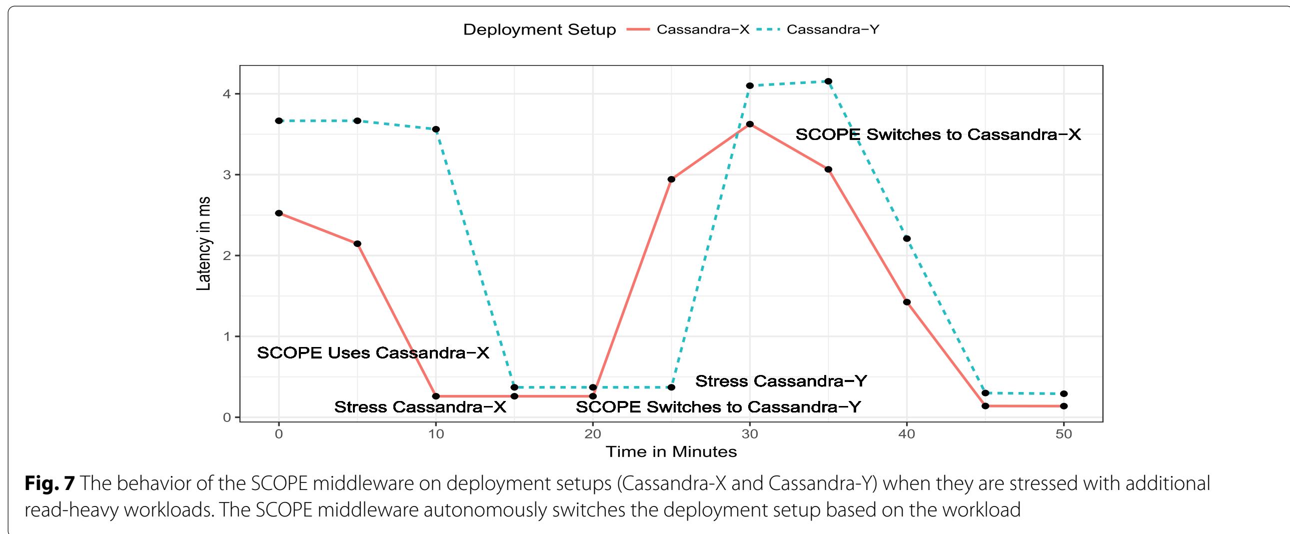


Fig. 7 The behavior of the SCOPE middleware on deployment setups (Cassandra-X and Cassandra-Y) when they are stressed with additional read-heavy workloads. The SCOPE middleware autonomously switches the deployment setup based on the workload

requests are sent out to all deployment setups and up-to-date monitored metrics (cf. Table 2 to see the supported metric types for different cloud storage systems) are gathered. The continuous monitoring underneath requires extensive interaction with the deployment setups (e.g., to perform various search queries) in order to combine different complementary metrics together to make decisions that are in line with the changing environment. Therefore, continuous monitoring has a direct impact on the read performance rather than the write performance. However, to confirm our observations, we quantify the monitoring overhead for write, write with replication (i.e., cross-provider data replication), and read operations.

In contrast to the previous experiment, in this experiment application prototypes are configured statically to use the same deployment setups, which are not stressed with additional workloads. To assess the impact of monitoring on write and read performance, first both application prototypes (i.e., prototype SCOPE and prototype SCOPE^{-MR}) are statically configured to use the Cassandra-X deployment setup and the results are obtained. Afterwards, both application prototypes are configured to use Cassandra-X and Cassandra-Y deployment setups in order to investigate the monitoring overhead on write with replication operation (i.e., data is replicated in both deployment setups). In order to allow for fair comparison for both application prototypes, deployment setups are not supplemented with additional workload stress.

Table 3 Total time in seconds prototype SCOPE^{-MR} and prototype SCOPE takes to execute the data size of 100 K entries. The decision to use deployment setups based on monitored metrics shows significant improvement in write and read performance (212% and 40% respectively) for prototype SCOPE

Application prototype	Write	Read
SCOPE^{-MR}	1012	656
SCOPE	322	501
% Performance improvement	212%	40%

5.3.2 Results

The results of this experiment are presented in Table 4.

As summarized, prototype SCOPE takes 322 s for write operations, 1089 s for write with replication operations, and 501 s for read operations compared to prototype SCOPE^{-MR}, which takes 314, 1051, 435 s for write, write with replication, and read operations respectively. The average relative performance overhead introduced by prototype SCOPE is 3% for write operations and 4% for write with replication, whereas the monitoring overhead increases to 15% when the read operations are performed.

5.4 Functional Validation of Reconfiguration Support

In the final experiment, we demonstrate the self-adaptive capabilities of SCOPE: the potential to perform monitoring and self-scaling. Section 5.4.1 describes the setup, while the results are presented in Section 5.4.2.

Table 4 Total time in seconds prototype SCOPE^{-MR} (without monitoring and reconfiguration) and prototype SCOPE (with monitoring and reconfiguration) takes to execute the data size of 100 K entries. The SCOPE prototype built on top of SCOPE introduces an additional average relative performance overhead of 3%, 4%, 15% on write, write with replication, and read operations

Application prototype	Write	Write replication	Read
SCOPE^{-MR}	314	1051	435
SCOPE	322	1089	501
Monitoring overhead	3%	4%	15%

5.4.1 Setup
In this experiment, we leverage the implemented self-adaptive capabilities of SCOPE to demonstrate auto scaling. To this end, data is stored in the Cassandra-X deployment setup in a replicated manner (i.e., data is replicated in both the Node #1 and the Node #2 with the replication_factor = 2). However, initially, only one node (i.e., Node #1) is active and running, while the other node (i.e., Node #2) is in the standby mode. Then, while performing read operations (i.e., reading 100 K entries), Node #1 is stressed with a additional read-heavy workload using the standard benchmarking tools. The sheer increase in the workload decreases the read performance, increases the read latency, and thus results in the overall degradation in the performance of Node #1. In this experiment, we examine and assess the impact of such increase in the workload —which results in performance degradation— on the behavior of the SCOPE middleware (e.g., how SCOPE will identify the performance degradation on Node #1 and what actions will be taken to deal with such scenarios).

5.4.2 Results
The results of this experiment are presented in Fig. 8.

As illustrated in Fig. 8, only one node (i.e., Node #1) is initially active and running. As data is replicated across the nodes (Node #1 and Node #2), prototype SCOPE reads data from the Node #1. Then, at the time scale of 5 min, Node #1 is stressed with the read-heavy workload (# of threads slowly increases up to 500 threads). Therefore, starting at the time scale of 10 min, the performance of the Node #1 decreases and the read latency slowly goes up until it reaches to the specified threshold value for the scale out (i.e., 1.25 ms as shown in Fig. 8).

The SCOPE prototype, built on top of the SCOPE middleware continuously monitors this behavior of the Node #1 (i.e., read latency). However, at the time scale when the read latency of the Node #1 reaches to the maximum specified threshold value (i.e., at 15 min), the scope middleware performs the necessary configuration for the Node #2 (e.g., updating the persistence configuration list). Then, the SCOPE middleware performs auto-scaling (i.e., starting Node #2, which was initially inactive). Now, the requests are load balanced across the Node #1 and the Node #2 and as also visible in Fig. 8 between the time interval of 15 and 20 min, slowly the latency of prototype SCOPE tends to decrease and remains below the threshold value (i.e., 1.25 ms). Hence, SCOPE also satisfies the requirement **R2** discussed in Section 2.3.

6 Discussion and future work
In this section, we describe some limitations of SCOPE and discuss how these can be addressed in future work.

Monitoring Interval (*t*). In general, self-adaptive systems require the support of a monitoring component/system, which provides measurements about the current system status. As briefly discussed in Section 5.3.1, to assess the impact of continuous monitoring on CRUD transactions, the monitoring interval t is used, where the value of t is set to 10 s. This implies that the monitoring component of SCOPE continuously monitors the federated cloud storage setup (i.e., after every 10 s) and up-to-date monitored metrics are collected, stored, and used.

In SCOPE, up-to-date monitored metrics are extremely useful to make decisions that are consistent with the operational environment (i.e., the federated cloud storage setup). Therefore, the inferred value of t is extremely important. To monitor up-to-date metrics, the value of t can be set to a relatively lower value. The reason is, the lower the value of t, the more accurate and up-to-date monitored metrics can be perceived and thus the more consistent decisions can be made. Inversely, setting the value of t to a relatively lower value also negatively impacts the performance of CRUD transactions. This is mainly because instead of every 10 s, monitored metrics will be collected and stored after every t seconds where the value of $t <10$ s. However, in practice, it is inherently difficult to determine the appropriate value of t.

In addition, storage technologies and resources in federated cloud storage setups are heterogeneous in nature. Each NoSQL database makes different choices and implies trade offs between the key concerns such as consistency, availability, and partition tolerance (CAP) theorem [23]. Furthermore, each class of NoSQL system employs a different data model. For example, Apache Cassandra is categorized under wide-column store family of NoSQL databases, whereas MongoDB is one of the representatives of document store family of NoSQL databases. Similarly, NoSQL databases offer different consistency guarantees (e.g., weak vs. eventual vs. strong consistency) and the storage model of these databases is also significantly different (e.g., in-memory vs. on-disk storage). Consequently, the inferred value of t can not be generalized for all NoSQL databases in a federated cloud storage setup.

Another important consideration is the frequency at which different dynamic properties of a federated cloud storage setup (e.g., performance, availability, cost, etc) should be monitored. This is a step forward because dynamic properties in a federated cloud storage setup evolve at different frequency. For example, performance in cloud storage providers changes more frequently than for example, the cost (i.e., pricing schemes). Therefore, a fine-grained approach is required in the SCOPE middleware where each dynamic property can be configured

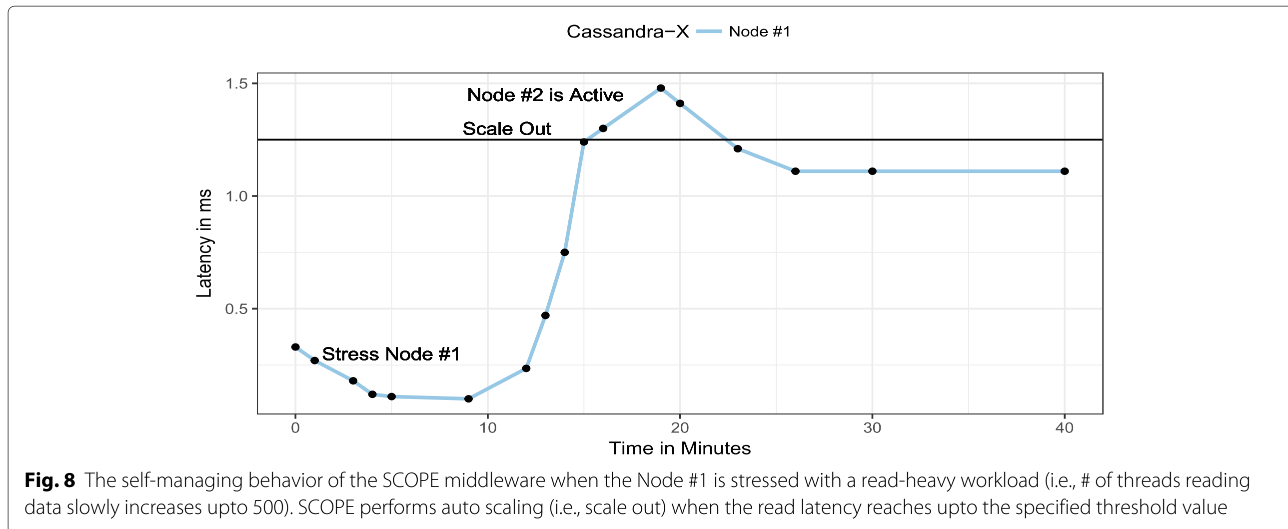


Fig. 8 The self-managing behavior of the SCOPE middleware when the Node #1 is stressed with a read-heavy workload (i.e., # of threads reading data slowly increases upto 500). SCOPE performs auto scaling (i.e., scale out) when the read latency reaches upto the specified threshold value

independently so that it can be monitored at different frequency.

To address these concerns, a self-adaptive monitoring system based on the MAPE-K loop may be a candidate solution [26] for future improvement. The self-adaptive monitoring system in SCOPE can determine the appropriate value of t by dynamically adjusting the monitoring interval t and then selecting the best monitoring interval. Addressing the concerns of heterogeneous technologies and resources in federated cloud storage setups, in a similar manner, the self-adaptive monitoring system can also determine the appropriate value of t for each NoSQL database.

Static (Re)configuration Policies. To support adaptation process and facilitate auto scaling of resources, the analysis and the planning phase of the MAPE-K reference architecture are extremely important. In practice, auto-scaling techniques are classified into two main categories: reactive and proactive. Reactive techniques typically involve making plans based on certain threshold values, while the proactive techniques try to predict and anticipate future needs.

In SCOPE, static reconfiguration policies (e.g., static rule-based policies) are an example of the reactive techniques. This involves creating a number of different rules to determine when to perform an action (e.g., scale up or scale down). For each rule, condition is defined based on a specific target variable. For example, If the load on the CPU > 90%, then perform an action (e.g., scale up, scale out, etc). Similarly, If the latency of MongoDB > 10 ms, then perform an action (e.g., scale out, switch cloud storage technology, etc). However, the key challenge is to find the right applicable method to detect the inferred value for the target variable.

Therefore, in future, our goal is to employ a hybrid reactive-proactive auto-scaling technique: combining threshold-based rules with the time series analysis. This would enable us to create reconfiguration rules based on the execution history data.

7 Related work

This section discusses two categories of related work. First, we review the related work on multi-cloud or federated cloud systems. Then, we describe recent research on self-adaptive systems with special emphasis on cloud storage and cloud provisioning.

7.1 Multi-cloud systems

The concept of multi cloud was first introduced in 2010 by Vukolic [27]. After that, numerous works have been done in this area. However, in recent years, a multi-cloud or a federated cloud setup has become a highly attractive paradigm for cloud-based applications because of its potential to provide high availability, better scalability, and protection against the vendor lock-in problem.

Recently, a number of multi-cloud storage systems such as Hybris [7], MetaStorage [12], MCDB [18], DEPSKY [19], HAIL [20], ICStore [28], SPANStore [29], NCCloud [30], SHAMC [31], TCKS [32], CDStore [33], RACS [34], CloudS [35], Scalia [21], AppScale [36], CDPort [37], CSAL [38], Cloud4SOA [39], CHARM [40] have been proposed to meet different non-functional requirements of the application. The main objective of these systems is to leverage multiple cloud providers either to enhance data availability and reliability, ensure data security, optimize storage cost, distribute the trust across clouds, or avoid the vendor lock-in problem.

However, each of these multi-cloud systems mainly focuses on specific non-functional requirements. For example, systems such as SafeStore [41] and NCCloud [30] address data reliability regarding cloud failures and the vendor lock-in problem. DEPSKY [19], a virtual cloud

storage system splits files into several parts and then sends them to different cloud providers. The goal is to avoid the frequent interruption of service, but also to address the confidentiality of data. RACS [34] on the other hand spreads the load over multiple cloud providers with RAID-like techniques to achieve the same goal as DEPSKY (i.e., high availability of data). RACS however, does not focus on the security aspects of clouds (e.g., confidentiality), rather deals with the vendor lock-in problem. CHARM [40] is a multi-cloud system, which integrates two key desired features. Firstly, it offers cost-efficient data storage with high availability. Secondly, redistributes data according to the variations of data access patterns.

Aimed at guaranteeing data security and privacy aspects of cloud computing, MCDB [18] and TCKS [32] use multiple cloud service providers to build a single cloud database. The ultimate goal of these systems is to allow an organization to outsource its data management tasks to multiple cloud providers while preserving data privacy. SHAMC [31] is based on the idea of secure multiparty computation and homomorphic encryption. Similar to DEPSKY [19], the goal of SHAMC [31] is to avoid service interruption and solve the vendor lock-in problem. However, unlike DEPSKY [19] where data is split into several parts and store in multiple clouds, SHAMC [31] stores the entire database in multiple clouds. CloudS [35], a multi-cloud storage system spreads data over multiple clouds by using a variety of combinations of compression, encryption, and coding schemes (XOR-based non-systematic erasure codes). In [42], the authors proposed ExpanStor, a multi-cloud storage system to provide security and reliability support in a multi-cloud storage setup along with the characteristics of the dynamic data distribution.

HAIL [20] is a cryptographic multi-cloud system, which combines proofs of retrievability (PORs) and proofs of data possession (PDPs) to guarantee data integrity and availability. ICStore [28] addresses CIRC attributes (confidentiality, integrity, reliability and consistency) of clouds by using data encryption, data replication, and Shamir's secret sharing scheme [43]. MetaStorage [12], a federated cloud storage system that integrates diverse cloud storage services and replicates data on these services to achieve high availability by using distributed hash tables. However MetaStorage [12] does not address the data confidentiality problem, which is later achieved by Hybris [7], a hybrid cloud storage system by distributing the encrypted data over multiple clouds via erasure coding and keeping secret keys in a private cloud.

Aimed at guaranteeing cost optimization, Scalia [21] is a cloud brokerage solution that makes data placement decisions based on the data access patterns subject to storage cost optimization. However, Scalia is a single-purpose solution that combines multiple cloud providers to optimize the cost factor, whereas SCOPE in principle is extensible and a multi-purpose solution that supports numerous run-time factors such as performance, availability, etc. Similar to Scalia, CDStore [33] provides a unified multi-cloud storage solution to guarantee cost efficiency. In addition, it also provides reliability and security guarantees, which are not reflected in Scalia, but are prominent in SCOPE. In contrast to Scalia and CDStore, SPANStore [29] focuses on unifying the use of resources from multiple clouds with the objective to minimize the cost by exploiting pricing discrepancies across providers.

However, none of the aforementioned multi-cloud systems has self-adaptive capabilities, and they are mostly influenced by decisions based on static properties. Thereby, these systems are not capable to autonomously react to changes in their environment and are required to be manually managed by the operator. In addition, none of these multi-cloud systems supports flexible configuration policies in the function of federated cloud reconfiguration.

In [3], we have presented PERSIST, a policy-based data management middleware that provides fine-grained control over data storage in a multi-cloud setup, and makes data placement decisions that are based on static properties. As discussed earlier and also demonstrated in Section 5, data management decisions, which are based on static properties, lead to sub-optimal performance when the multi-cloud setup evolves dynamically. The architecture presented in this paper as such extends such a static policy-driven setup with support for policies that are based on dynamic conditions of the operational environment.

SPACE4CLOUD (System PerformAnce and Cost Evaluation on Cloud) [44] is a tool for the optimization of QoS characteristics of cloud applications. It follows a model-driven approach with the major goal to evaluate the performance and cost estimation of cloud and multi-cloud systems. This allows applications to decide, which cloud provider is well suited (in terms of cost) and whether it will guarantee the required SLOs, before their actual development (i.e., at design time). However, the performance of cloud storage providers may vary considerably at run time depending upon the workload fluctuations. SCOPE on the other hand, considers the run-time dynamicity in a federated cloud storage setup, and therefore makes decisions at run time based on SLOs specified by the SaaS application. In addition, SCOPE is a middleware, which contains the self-adaptive capabilities and makes various decisions (e.g., dynamic data placement, add resources, remove resources, etc) at run time, while SPACE4CLOUD is a model, which focuses only on performance and cost estimation at design time.

7.2 Self-adaptive systems

Modern software applications (e.g., SaaS applications, IoT applications, etc) typically operate in a highly dynamic environment and deal with rapidly changing operational conditions. The complexity of managing such applications manually, has initiated efforts in both the industry and the academia to develop self-adaptive systems [45–48]. Self adaptation —which can be realized by means of a feedback control loop such as in the MAPE-K reference architecture [49, 50]— has been widely recognized as an effective approach to dealing with the dynamicity and the increasing complexity.

A number of self-adaptive systems have been built to serve different purposes in several different application domains such as wireless sensor networks, multi-tenant SaaS applications, databases, cyber-physical systems etc. For example, In [51] Tsoumakos et al. present TIRAMOLA, a cloud-enabled framework. The main purpose of TIRAMOLA is to automatically perform resizing of NoSQL database clusters based on user-defined policies. In another similar work, Truyen et al. have proposed K8-Scalar [52], an extensible workbench for evaluating different self-adaptive approaches to auto-scale NoSQL database clusters. Similarly, Fetai et al. [53] presents Cumulus that minimizes distributed transactions through adaptive data repartitioning.

However, these self-adaptive systems focus on specific goals and objectives in comparison to SCOPE. For example, TIRAMOLA focuses on auto-scaling NoSQL databases, while Cumulus only emphasis on minimizing transactions. Similarly, K8-Scalar is a tool to evaluate different self-adaptive approaches to auto-scale NoSQL cluster. In SCOPE, K8-Scalar can be used to determine the appropriate threshold value for the auto scale (e.g., when to scale up, scale out, etc). In addition, these systems operate in a single cloud environment and thus they do not sufficiently consolidate the heterogeneity of the underlying complex federated cloud storage setup. In contrast, SCOPE makes dynamic data management decisions across federated clouds and in principle also supports different scaling abilities (e.g., scale up, scale down, scale out, scale in, etc), which are not implied by the abovementioned systems.

In self-adaptive systems, the decisions to perform actions (e.g., resizing of NoSQL database clusters in case of TIRAMOLA, to trigger the adaptive data repartitioning in case of Cumulus, etc) are usually based on the execution history. Lorido et al. [54] proposed an auto-scaling technique, which can be classified into reactive (i.e., based on rules or using the last values obtained from the set of monitored variables) and proactive (i.e., anticipate future demands and make decisions by taking them into consideration), as well as more fine-grained techniques, resulting in: (i) threshold-based rules, (ii) reinforcement learning, (iii) queuing theory, (iv) control theory, and (v) time series analysis.

These techniques have been successfully applied in the existing self-adaptive systems. For example, threshold-based rules are applied in [51, 52] to perform resizing and autoscaling in NoSQL database clusters. In addition, self-adaptive systems for cloud management is the subject of current research. In that context, Control theory is utilised in [55] to manage uncertainty concerns in the cloud. In SCOPE, instead of only relying on threshold-based rules for performing auto scaling, other auto-scaling techniques such as reinforcement learning or time series analysis can also be used.

8 Conclusion

This paper presents SCOPE, a policy-based middleware with self-adaptive capabilities for data management in federated clouds. SCOPE (i) continuously monitors the run-time dynamicity of a federated cloud setup (i.e., changing conditions of underlying storage systems) and then collects, stores, and aggregates the monitored metrics; (ii) constantly adapts to the continuous changes in the operating environment and selects (based on the assembled metrics) the most suitable and an appropriate cloud storage provider for data management decisions; and (iii) it autonomously reconfigures the federated cloud storage architecture (e.g., spill over, scale out, scale up, etc) by identifying the behavior of individual storage systems and changing conditions in the operating environment and therefore, is no longer required to be manually managed by the operator.

We have validated the core concept in an industry-level SaaS application, a document processing prototype implementation on top of SCOPE. By performing an extensive functional validation and conducting a thorough experimental evaluation, we showed that SCOPE performs SLA-aware data management decisions. In addition, we have also demonstrated the self-adaptive capabilities of the SCOPE middleware (i.e., SCOPE performs reconfiguration actions without the need for continuous human operator intervention). Finally, we also provided evidence that the performance overhead of SCOPE is minimal.

Endnotes

[1] http://www.ehcache.org/

[2] http://lucene.apache.org/

[3] https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsNodetool.html

[4] https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

[5] https://www.drools.org/

[6] https://tomcat.apache.org/

[7] CRUD stands for create, read, update, and delete.

[8] https://www.openstack.org/

[9] https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsCStress.html

[10] These latency values are only presented to demonstrate the dynamic behavior of the deployment setups and the potential of our proposed middleware in terms of switching the deployment setups, when the load is progressively increased.

Abbreviations
API: application programming interface; B2B: business-to-business CRUD: create, read, update, and delete; CSPs: cloud storage providers; IoT: Internet of Things; QoS: Quality-of-Service; RCT: read completion time; SaaS: Software-as-a-Service; SLA: Service Level Agreement; SLO: Service-Level-Objective; WCT: write completion time

Acknowledgments
We would like to thank Luuk Raaijmakers for the initial implementation of the monitoring API as part of his master thesis.

Funding
This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS) and the SBO DeCoMAdS project.

Availability of data and materials
The prototype of the SCOPE middleware is publicly available at: http://people.cs.kuleuven.be/ansar.rafique/SCOPE.zip. The prototype is implemented in Java and is therefore platform independent.

Authors' contributions
AR was responsible for the majority of the technical work that includes implementing the prototype, conducting the experiments, collecting the results, and drafting the final manuscript. DVL supervised the overall research and made contributions to the conception and design of the proposed solution and also contributing for critically reviewing the manuscript. ET and VR have participated in the initial design of the middleware and have contributed to the positioning of the work. WJ conducted a final supervision. All authors have read and approved the final manuscript.

Authors' information
The authors are affiliated with the imec-DistriNet, a research group within the Department of Computer Science of KU Leuven, Belgium.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 1 August 2018 Accepted: 13 December 2018
Published online: 30 January 2019

References
1. Zhang Q, Cheng L, Boutaba R. Cloud computing: state-of-the-art and research challenges. J Internet Serv Appl. 2010;1(1):7–18. https://doi.org/10.1007/s13174-010-0007-6.
2. Grozev N, Buyya R. Multi-cloud provisioning and load distribution for three-tier applications. ACM Trans Auton Adapt Syst. 2014;9(3):13–11321. https://doi.org/10.1145/2662112.
3. Rafique A, Van Landuyt D, Joosen W. Persist: Policy-based data management middleware for multi-tenant saas leveraging federated cloud storage. J Grid Comput. 2018;16(2):165–94. https://doi.org/10.1007/s10723-018-9434-6.
4. Kovács J, Kacsuk P. Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures. J Grid Comput. 2018;16(1): 19–37. https://doi.org/10.1007/s10723-017-9421-3.
5. Grolinger K, Higashino WA, Tiwari A, Capretz MA. Data management in cloud environments: Nosql and newsql data stores. J Cloud Comput Adv Syst Appl. 2013;2(1):22. https://doi.org/10.1186/2192-113X-2-22.
6. Wu J, Ping L, Ge X, Wang Y, Fu J. Cloud storage as the infrastructure of cloud computing. In: 2010 International Conference on Intelligent Computing and Cognitive Informatics; 2010. p. 380–3. https://doi.org/10.1109/ICICCI.2010.119.
7. Dobre D, Viotti P, Vukolić M. Hybris: Robust hybrid cloud storage. In: Proceedings of the ACM Symposium on Cloud Computing, SOCC '14. New York: ACM; 2014. p. 12–11214. https://doi.org/10.1145/2670979.2670991. http://doi.acm.org/10.1145/2670979.2670991.
8. Singh Y, Kandah F, Zhang W. A secured cost-effective multi-cloud storage in cloud computing. In: 2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS); 2011. p. 619–24. https://doi.org/10.1109/INFCOMW.2011.5928887.
9. Subashini S, Kavitha V. A survey on security issues in service delivery models of cloud computing. J Netw Comput Appl. 2011;34(1):1–11. https://doi.org/10.1016/j.jnca.2010.07.006.
10. Gubbi J, Buyya R, Marusic S, Palaniswami M. Internet of things (iot): A vision, architectural elements, and future directions. Futur Gener Comput Syst. 2013;29(7):1645–60. https://doi.org/10.1016/j.future.2013.01.010.
11. Fazio M, Celesti A, Villari M, Puliafito A. The need of a hybrid storage approach for iot in paas cloud federation. In: 2014 28th International Conference on Advanced Information Networking and Applications Workshops; 2014. p. 779–84. https://doi.org/10.1109/WAINA.2014.162.
12. Bermbach D, Klems M, Tai S, Menzel M. Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In: 2011 IEEE 4th International Conference on Cloud Computing; 2011. p. 452–9. https://doi.org/10.1109/CLOUD.2011.62.
13. Jiang L, Xu LD, Cai H, Jiang Z, Bu F, Xu B. An iot-oriented data storage framework in cloud computing platform. IEEE Trans Ind Inform. 2014;10(2):1443–51. https://doi.org/10.1109/TII.2014.2306384.
14. Rafique A, Van Landuyt D, Reniers V, Joosen W. Towards an adaptive middleware for efficient multi-cloud data storage. In: Proceedings of the 4th Workshop on CrossCloud Infrastructures & Platforms, Crosscloud'17. New York: ACM; 2017. p. 4–146. https://doi.org/10.1145/3069383.3069387. http://doi.acm.org/10.1145/3069383.3069387.
15. D-Base: Enabling distributed business process outsourcing (BPO) in an interoperable, scalable and secure way. 2018. https://www.imec-int.com/en/what-we-offer/research-portfolio/d-base. [Last visited on July 20, 2018].
16. DMS2: Decentralized Data Management and Migration for SaaS (iMinds ICON project). 2018. https://distrinet.cs.kuleuven.be/research/projects/(DMS)2. [Last visited on June 14, 2018].
17. CUSTOMSS: CUSTOMization of Software Services in the cloud (imec ICON project). 2018. https://distrinet.cs.kuleuven.be/research/projects/CUSTOMSS. [Last visited on June 27, 2018].
18. Alzain MA, Soh B, Pardede E. Mcdb: Using multi-clouds to ensure security in cloud computing. In: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing; 2011. p. 784–91. https://doi.org/10.1109/DASC.2011.133.
19. Bessani A, Correia M, Quaresma B, André F, Sousa P. Depsky: Dependable and secure storage in a cloud-of-clouds. Trans Storage. 9(4): 12–11233. https://doi.org/10.1145/2535929.
20. Bowers KD, Juels A, Oprea A. Hail: A high-availability and integrity layer for cloud storage. In: Proceedings of the 16th ACM conference on Computer and communications security. ACM; 2009. p. 187–98.
21. Papaioannou TG, Bonvin N, Aberer K. Scalia: An adaptive scheme for efficient multi-cloud storage. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '12. Los Alamitos: IEEE Computer Society Press; 2012. p. 20–12010. http://dl.acm.org/citation.cfm?id=2388996.2389024.

22. Impetus: Kundera: Object-Datastore Mapping Library for NoSQL Datastores. 2014. https://github.com/impetus-opensource/Kundera. [Last visited on June 28, 2014].
23. Rafique A, Van Landuyt D, Lagaisse B, Joosen W. On the performance impact of data access middleware for nosql data stores a study of the trade-off between performance and migration cost. IEEE Trans Cloud Comput. 2018;6(3):843–56. https://doi.org/10.1109/TCC.2015.2511756.
24. Reniers V, Rafique A, Van Landuyt D, Joosen W. Object-nosql database mappers: a benchmark study on the performance overhead. J Internet Serv Appl. 2017;8(1):1.
25. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10. New York: ACM; 2010. p. 143–54. https://doi.org/10.1145/1807128.1807152. http://doi.acm.org/10.1145/1807128.1807152.
26. Ehlers J, van Hoorn A, Waller J, Hasselbring W. Self-adaptive software system monitoring for performance anomaly localization. In: Proceedings of the 8th ACM International Conference on Autonomic Computing. ICAC '11. New York: ACM; 2011. p. 197–200. https://doi.org/10.1145/1998582.1998628. http://doi.acm.org/10.1145/1998582.1998628.
27. Vukolic M. The byzantine empire in the intercloud. ACM SIGACT News. 2010;41(3):105–11. cited By 43.
28. Cachin C, Haas R, Vukolic M. Dependable storage in the intercloud. Technical report, Research Report RZ, 3783; 2010.
29. Wu Z, Butkiewicz M, Perkins D, Katz-Bassett E, Madhyastha HV. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13. New York: ACM; 2013. p. 292–308. https://doi.org/10.1145/2517349.2522730. http://doi.acm.org/10.1145/2517349.2522730.
30. Chen HCH, Hu Y, Lee PPC, Tang Y. Nccloud: A network-coding-based storage system in a cloud-of-clouds. IEEE Trans Comput. 2014;63(1):31–44. https://doi.org/10.1109/TC.2013.167.
31. Wang L, Yang Z, Song X. Shamc: A secure and highly available database system in multi-cloud environment. Futur Gener Comput Syst. 2017. https://doi.org/10.1016/j.future.2017.07.011.
32. Aggarwal G, Bawa M, Ganesan P, Garcia-Molina H, Kenthapadi K, Motwani R, Srivastava U, Thomas D, Xu Y. Two can keep a secret: A distributed architecture for secure database services. In: The Second Biennial Conference on Innovative Data Systems Research (CIDR 2005); 2005. http://ilpubs.stanford.edu:8090/659/.
33. Li M, Qin C, Lee PPC. Cdstore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15). Santa Clara: USENIX Association; 2015. p. 111–24. https://www.usenix.org/conference/atc15/technical-session/presentation/li-mingqiang.
34. Abu-Libdeh H, Princehouse L, Weatherspoon H. Racs: A case for cloud storage diversity. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10. New York: ACM; 2010. p. 229–40. https://doi.org/10.1145/1807128.1807165. http://doi.acm.org/10.1145/1807128.1807165.
35. Shen L, Feng S, Sun J, Li Z, Wang G, Liu X. Clouds: A multi-cloud storage system with multi-level security. In: Wang G, Zomaya A, Martinez G, Li K, editors. Algorithms and Architectures for Parallel Processing. Cham: Springer; 2015. p. 703–16.
36. Chohan N, Bunch C, Pang S, Krintz C, Mostafa N, Soman S, Wolski R. Appscale: Scalable and open appengine application development and deployment. In: Avresky DR, Diaz M, Bode A, Ciciani B, Dekel E, editors. Cloud Computing. Berlin, Heidelberg: Springer; 2010. p. 57–70.
37. Alomari E, Barnawi A, Sakr S. Cdport: A portability framework for nosql datastores. Arab J Sci Eng. 2015;40(9):2531–53. https://doi.org/10.1007/s13369-015-1703-0.
38. Hill Z, Humphrey M. Csal: A cloud storage abstraction layer to enable portable cloud applications. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science; 2010. p. 504–11. https://doi.org/10.1109/CloudCom.2010.88.
39. DAndria F, Bocconi S, Cruz JG, Ahtes J, Zeginis D. Cloud4soa: Multi-cloud application management across paas offerings. In: 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing; 2012. p. 407–14. https://doi.org/10.1109/SYNASC.2012.65.
40. Zhang Q, Li S, Li Z, Xing Y, Yang Z, Dai Y. Charm: A cost-efficient multi-cloud data hosting scheme with high availability. IEEE Trans Cloud Comput. 2015;3(3):372–86.
41. KOTLA R. Safestore : A durable and practical storage system. In: 2007 USENIX Annual Technical Conference, June; 2007.
42. Wei Y, Chen F, Sheng DCJ. Expanstor: Multiple cloud storage with dynamic data distribution. In: 2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2); 2017. p. 85–90. https://doi.org/10.1109/SC2.2017.20.
43. Shamir A. How to share a secret. Commun ACM. 1979;22(11):612–3. https://doi.org/10.1145/359168.359176.
44. Franceschelli D, Ardagna D, Ciavotta M, Di Nitto E. Space4cloud: A tool for system performance and costevaluation of cloud systems. In: Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds. MultiCloud '13. New York: ACM; 2013. p. 27–34. https://doi.org/10.1145/2462326.2462333. http://doi.acm.org/10.1145/2462326.2462333.
45. Markl V, Lohman GM, Raman V. Leo: An autonomic query optimizer for db2. IBM Syst J. 2003;42(1):98–106. https://doi.org/10.1147/sj.421.0098.
46. Kumar V, Cooper BF, Schwan K. Distributed stream management using utility-driven self-adaptive middleware. In: Second International Conference on Autonomic Computing (ICAC'05); 2005. p. 3–14. https://doi.org/10.1109/ICAC.2005.24.
47. Portocarrero JMT, Delicato FC, Pires PF, Rodrigues TC, Batista TV. Samson: Self-adaptive middleware for wireless sensor networks. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. SAC '16. New York: ACM; 2016. p. 1315–22. https://doi.org/10.1145/2851613.2851766. http://doi.acm.org/10.1145/2851613.2851766.
48. Rouvoy R, Beauvois M, Lozano L, Lorenzo J, Eliassen F. Music: An autonomous platform supporting self-adaptive mobile applications. In: Proceedings of the 1st Workshop on Mobile Middleware: Embracing the Personal Communication Device. MobMid '08. New York: ACM; 2008. p. 6–166. https://doi.org/10.1145/1462689.1462697. http://doi.acm.org/10.1145/1462689.1462697.
49. Kephart JO, Chess DM. The vision of autonomic computing. Computer. 2003;36(1):41–50.
50. Macias-Escriva FD, Haber R, del Toro R, Hernandez V. Self-adaptive systems: A survey of current approaches, research challenges and applications. Expert Syst Appl. 2013;40(18):7267–79. https://doi.org/10.1016/j.eswa.2013.07.033.
51. Tsoumakos D, Konstantinou I, Boumpouka C, Sioutas S, Koziris N. Automated, elastic resource provisioning for nosql clusters using tiramola. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing; 2013. p. 34–41. https://doi.org/10.1109/CCGrid.2013.45.
52. Delnat W, Truyen E, Rafique A, Van Landuyt D, Joosen W. K8-scalar: a workbench to compare autoscalers for container-orchestrated database clusters. In: Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '18; 2015. https://doi.org/10.1145/3194133.3194162.
53. Fetai I, Murezzan D, Schuldt H. Workload-driven adaptive data partitioning and distribution ??? the cumulus approach. In: 2015 IEEE International Conference on Big Data (Big Data); 2015. p. 1688–97. https://doi.org/10.1109/BigData.2015.7363940.
54. Lorido-Botran T, Miguel-Alonso J, Lozano JA. A review of auto-scaling techniques for elastic applications in cloud environments. J Grid Comput. 2014;12(4):559–92. https://doi.org/10.1007/s10723-014-9314-7.
55. Filieri A, Maggio M, Angelopoulos K, D'Ippolito N, Gerostathopoulos I, Hempel AB, Hoffmann H, Jamshidi P, Kalyvianaki E, Klein C, Krikava F, Misailovic S, Papadopoulos AV, Ray S, Sharifloo AM, Shevtsov S, Ujma M, Vogel T. Software engineering meets control theory. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '15. Piscataway: IEEE Press; 2015. p. 71–82. http://dl.acm.org/citation.cfm?id=2821357.2821370.