J Grid Computing (2018) 16:165–194
https://doi.org/10.1007/s10723-018-9434-6

PERSIST: Policy-Based Data Management Middleware for Multi-Tenant SaaS Leveraging Federated Cloud Storage

Ansar Rafique · Dimitri Van Landuyt · Wouter Joosen

Received: 30 June 2017 / Accepted: 6 February 2018 / Published online: 14 March 2018

Abstract NoSQL data stores are often combined to address different requirements within the same application. The implication of this trend is particularly important and relevant in the context of multi-tenant SaaS applications where tenants commonly have different storage- and privacy-related requirements and thus they desire to customize the storage setup according to their specific needs. Consequently, application developers are increasingly combining storage resources: on-premise and public cloud resources in a hybrid cloud setup, different external public cloud storage resources and providers in a federated cloud storage setup, etc. The consequences of these trends are twofold: (i) application developers and SaaS providers have to deal with heterogeneous technologies, different APIs, and implement complex storage logic (to address different requirements of tenants), all within the application layer; and (ii) storage architectures have become less rigid, and techniques are required to flexibly change the storage configuration of running applications, up to the level of individual service requests. To address these challenges, we present PERSIST, a middleware architecture that (i) externalizes the complexity of a federated cloud storage architecture and the complex storage logic from the SaaS application to storage policies, allows tenants to enforce different storage- and privacy-related requirements at a fine-grained level; and (ii) supports the dynamic (re)configurability of the underlying federated cloud storage architecture. Application-specific policies can be customized by individual tenants at run time, and PERSIST offers support for run-time cross-provider polyglot persistence and the confidentiality of sensitive data through encryption. We have validated PERSIST in a working prototype implementation. Our extensive evaluation efforts show (i) the accomplished reduction in the required development effort to support complex storage policies, (ii) the reduction in cost/effort to change the data storage architecture itself, and finally (iii) the acceptability of the performance overhead (around 6% for insert, and 2% for read, update and delete transactions).

Keywords Policy-based middleware · NoSQL data stores · Polyglot persistence · Multi-tenant SaaS · Federated cloud storage · Data encryption

1 Introduction

Cloud computing in the recent years has shown great success as a service-oriented computing paradigm that enables convenient and on-demand network access to a shared pool of computing resources [29, 30]. It

A. Rafique (✉) · D. Van Landuyt · W. Joosen
Department Computer Science, imec-DistriNet Research Group, Celestijnenlaan 200A, 3001 Heverlee, Belgium
e-mail: Ansar.Rafique@cs.kuleuven.be

scales on demand to support fluctuating workloads and eliminates the cost of maintaining an expensive on-premise infrastructure [17, 20, 48], as such allowing for significant cost savings. For these reasons, an increasing number of applications are built upon cloud platforms.

The well-known limitations —in terms of performance, scalability, and availability— of traditional databases in a cloud environment has lead to the emergence of a family of cloud-friendly databases, which are commonly referred to as *NoSQL* [6, 19, 28, 33, 45]. The term *NoSQL*, as an abbreviation of *not only SQL*, as such relaxes a number of traditional constraints (such as the ACID transaction properties) in the favor of high availability, elastic scalability, better performance, etc. — concerns that are particularly relevant in the context of Big Data and Cloud computing [11, 13, 16].

There is a wide variety of NoSQL database systems that can be used in cloud applications [12]. However, NoSQL systems are not general-purpose: they are tailored to specific use cases, address specific storage requirements, and the selection of a certain NoSQL technology impacts different functional and non-functional requirements [4, 34, 35]. As a consequence, NoSQL systems are in practice increasingly combined and used in the context of a single application, a scheme commonly referred to as *polyglot persistence* [9, 43].

In the specific context of multi-tenant Software-as-a-Service (SaaS) applications —in which a single instance of an application is shared among many tenants— different tenants often have different functional and non-functional requirements, e.g., a tenant may demand for specific SLA stipulations with regards to data confidentiality, availability, performance, etc. However, considering contrasting storage requirements of different tenants and the constant growth of computation and storage power required by multi-tenant SaaS applications, committing to a single cloud storage provider (e.g., a Database-as-a-Service provider) is often too restrictive and introduces new concerns related to cloud availability, security, data privacy, and adequate performance (e.g., response time), etc.

This creates strong opportunities and incentives for application providers to combine resources from different cloud storage providers. One of the main drivers for seeking support from multiple cloud storage providers is to avoid technology, vendor or service lock-in, which is a key hurdle against cloud storage adoption. In addition, working with more than one cloud storage provider can make a multi-tenant SaaS application more robust in the presence of failures and maximizes the anticipated needs of tenants. Application providers are in practice increasingly combining different storage resources, combining cloud storage with on-premise resources in so-called *hybrid* setups [14], and mixing different external cloud storage resources in so-called *multi-cloud* or *federated cloud* storage setups [7].

As a consequence of these trends, designing, implementing, operating, and customizing a multi-tenant SaaS application on top of a federated cloud storage setup has become significantly more difficult. Firstly, the reality of complex federated storage architectures leads to a complex federated cloud storage logic that is commonly implemented in the application layer, which makes it hard to change over time, for example, when data storage requirements or when the storage requirements of different tenants are considered. Secondly, SaaS providers and tenants of the SaaS application may want to impose certain data storage requirements that act at a fine-grained level, e.g., on the basis of concrete data elements of certain data types for specific tenants. For example, the data elements of certain data types may be considered sensitive and therefore, appropriate confidentiality-preserving measures must be taken before storing. Thus, supporting different storage requirements of a multi-tenant SaaS application at this level of granularity is highly challenging and increasingly demanding. Thirdly and finally, tenants may want to change the underlying storage architecture according to their specific requirements and therefore techniques are required to deal with such challenging situations.

In this paper, we present PERSIST, a policy-based middleware that confronts these challenges by (i) externalizing the complexity of a federated cloud storage architecture and a federated cloud complex storage logic —which is commonly implemented in the application and ultimately serves to address different storage- and privacy-related requirements of multiple tenants— from the SaaS application layer; (ii) facilitating tenants to enforce these requirements at a more fine-grained level, i.e. up to the level of individual data elements of certain data types for different data service requests as such also allows them to attain the benefits of dynamic federated cloud storage setups; and (iii) allowing different tenants of the SaaS application to flexibly and consistently (re)configure the underlying federated cloud storage architecture, according to their own specific needs, concerns, and priorities. In addition to addressing these challenges, PERSIST also provides an abstraction API to enable partitioning of application data across multiple data stores in a federated cloud storage setup. Furthermore, it offers support for run-time polyglot persistence and confidentiality of sensitive data at various degrees of granularity through data encryption. Moreover, PERSIST supports tenant customization by allowing tenants to refine storage policies at run time (in accordance to the principle of self-service [30]).

Our extensive prototype implementation —based on existing platforms and technologies such as Impetus Kundera [27], JBoss Drools [40], and Ehcache [15]— validates PERSIST in the context of a realistic multi-tenant SaaS application. In addition, our evaluation efforts focus on three different dimensions: (i) the cost/effort to implement complex storage policies, (ii) the cost/effort to change the back-end storage architecture, and finally (iii) the performance overhead introduced by the PERSIST middleware. These evaluations confirm that PERSIST reduces the development effort and increases the customizability, compared to existing systems, and that these benefits are attained with acceptable performance overhead (ranges between [1.7 - 6.5%], [0.5 - 2.4%], [0.4 - 2.4%], and [0.4 - 2.2%] for insert, read, update, and delete transactions respectively).

The remainder of the paper is organized as follows: Section 2 illustrates the above-mentioned challenges from the context of a realistic multi-tenant SaaS application case and further elaborates and exemplifies the key challenges of interest in this paper. Section 3 provides an in-depth discussion of the overall architectural design of the PERSIST middleware. In Section 4, we discuss the PERSIST prototype implementation, while the Section 5 presents our evaluation. Section 6 connects and contrasts PERSIST to related work. Section 7 continues with a brief discussion about the applicability of PERSIST for different categories of applications. Finally, Section 8 concludes the paper and outlines directions for the future work.

2 Motivation

The motivation for this paper is based on our experiences with a number of industrial SaaS application cases, which are obtained in the context of several applied research projects in collaboration with industry-level SaaS application providers [24–26, 41].

Table 1 illustrates a number of concrete use-case scenarios from various real-world applications. The last column of the Table 1 provides information about scenarios that require a combination of different cloud storage providers and technologies as well as illustrates the complexity of the respective data storage policies.

For illustration purposes, we focus on one such application case from the log management application domain, a multi-tenant Log Management-as-a-Service (LMaaS) offering, which we introduce in Section 2.1. More specifically, we highlight the nature of its dynamic storage- and privacy-related requirements for which support is lacking in current data access middleware platforms [23, 27, 44] and academic systems [1, 8, 10, 32]. Then, in Section 2.2, we briefly describe the motivation for a federated cloud storage architecture, especially in the context of multi-tenant SaaS applications. Based on that, in Section 2.3, we describe how the multi-tenant LMaaS application leverages various cloud storage resources and technologies in a federated cloud storage architecture. Section 2.4 subsequently outlines the key challenges for this paper.

2.1 Multi-Tenant Log Management-as-a-Service

The multi-tenant Log Management-as-a-Service (LMaaS) offering provides log management facilities —the collection, aggregation, reporting, processing, and archiving of log data— to its customer organizations (tenants). As such, it allows organizations to gain insight into their IT infrastructure without having to support or implement such functionality in-house. The tenants of this SaaS application are customer organizations of all sizes from very different application domains (e.g., banks, supermarkets, hospitals, etc.).

The application relies extensively on storing large volumes of heterogeneous data (4 TB/day), such as *raw log* entries, *archived logs*, *log meta-data*, *historical logs*, and *incident reports* from multiple tenants.

Table 1 Overview of different federated cloud scenarios from various real-world application domains

Application domain	Description	Example tenants	Federated cloud scenarios
Log management	Collecting and analyzing logs of the IT infrastructure	From SMEs to large corporations	Confidential *raw log* entries are stored in Cassandra, deployed in a private cloud, whereas non-confidential *historical logs* are stored in MongoDB, deployed in a public cloud (e.g., Amazon AWS).
Log analysis	Analysis and detection of anomalies and security breaches		
Document processing and generation	Generation of large sets of documents (PDFs) from raw data and delivery of these PDFs	Service providers (invoices), employers (payslips), banks (transaction overviews and bank statements)	PDFs $<$ 20MB are stored in MongoDB (deployed in a private cloud), while the PDFs $>$ 20MB are stored in MongoDB (deployed in a public cloud).
Patient image storage and analysis	Storage and semi-automated analysis of patient image	Hospitals, health service providers	Patient images, which are smaller in size are stored in a private cloud, while the large images are stored in a public cloud. The meta-data of images must always be indexed in an Elasticsearch (deployed in a private cloud), while the images are stored in an encrypted form.
Patient monitoring	AAL system, collection and analysis of extramural patient sensor data	Patient networks, hospitals, GPs	
Model-driven engineering workflows and simulations	On-line, on-demand model-based simulation as part of an engineering process	Engineering companies, car manufacturers, avionics, etc	Proprietary data (e.g., physical features of an aircraft) must not leave the tenant premise, while other data can be stored in public clouds.

For each log type, tenants may have ***varying storage requirements***, usually related to functional and non-functional aspects. For example, the *raw log* entries arriving at the LMaaS application from a financial agency (i.e. a bank) require high write throughput. Moreover, the *raw log* entries also require high read throughput as well as high availability for post-processing. In contrast, the *archived logs* and *historical logs* do not require high write throughput and availability as they can be processed at a slower pace, but high read throughput is a key requirement. Furthermore, storing *log meta-data* for all tenants requires only high write throughput, but at the same time high availability is a prerequisite.

Additionally, for different tenants, the application has to deal with data of ***varying degrees of confidentiality***. For example, *raw log* entries generated by tenants in the banking industry will involve more stringent confidentiality requirements and constraints than the *raw log* entries generated by a medical institution (i.e. a hospital) and a supermarket (i.e. an SME). Moreover, some tenants prefer to use their own ***on-premise storage infrastructure*** for storing data which has strict confidentiality requirements than relying on the storage infrastructure of the SaaS provider or the external cloud storage providers, while other tenants have no objections against having their data stored by external third-party cloud storage providers.

2.2 Federated Cloud Storage Architecture

In practice, the selection of a single cloud storage provider is often too restrictive and eventually leads to a costly decision-making process due to the following reasons (i) application providers confront with a lack of trust in the cloud storage provider and are thus reluctant to share confidential application data; (ii) application providers and their tenants may require (a) different SLA guarantees (with respect to performance, availability, and responsiveness) and (b) various storage technologies for addressing requirements of different types of application data, which might not be supported by the selected cloud storage provider and finally (iii) as market conditions evolve, application providers may inevitably want to switch cloud storage providers, but are in practice confronted with a situation of provider lock-in [37]. Therefore, it is extremely difficult to select the most appropriate cloud storage or the Database-as-a-Service provider for addressing the diverse storage- and privacy-related requirements of tenants.

To address these concerns, SaaS providers are increasingly leveraging different cloud storage resources, technologies, and providers in a so-called *federated cloud storage architecture*. More specifically, the storage tier of multi-tenant SaaS applications is increasingly being structured as a federated cloud storage architecture. This enables data storage requirements of a single application to be realized by combining different cloud storage resources, technologies, and providers — a concept known as *cross-provider* polyglot persistence.

2.3 LMaaS on Federated Cloud Storage Architecture

In case of the multi-tenant LMaaS application, the SaaS provider wants to minimize the up-front costs and reduce the maintenance costs by maximizing the utilization of external infrastructure services (e.g., third-party cloud storage services or providers). As each cloud storage provider has different feature strengths, a federated cloud storage architecture enables to mix and match these different features and services based on different requirements of the application or tenants. As an example, the multi-tenant LMaaS offering can take advantage of third-party external public cloud storage providers to reduce CapEx and achieve better scalability and continuous availability, while still keeping the sensitive data at the provider's or tenant's location.

Consequently, the multi-tenant LMaaS application as shown in Fig. 1 combines various cloud storage technologies, services, and providers to satisfy the respective storage- and privacy-related requirements of its tenants. The best-of-many-worlds approach of a federated cloud not only lets the multi-tenant LMaaS offering to take advantage of different feature strengths of multiple storage resources or cloud providers, but also helps to use the right technology

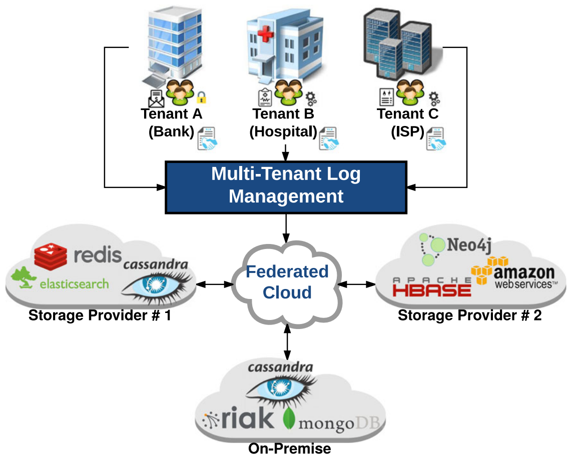


Fig. 1 The multi-tenant log management application uses a federated cloud storage architecture to satisfy contrasting requirements of tenants

for the right task. That is, the use of an appropriate data store for specific data elements —having different requirements— taking into account the nature of data elements and the data store (e.g., unstructured documents must be stored in document stores such as MongoDB or high throughput write storage requests must be propagated to the Cassandra data store). However, cloud storage providers may not be equally trustworthy. For example, European laws may consider data storage providers from the US to be untrustworthy. Therefore, additional precautions (e.g., encrypt confidential data before storing it) must be taken by the multi-tenant LMaaS application to comply with such demands and regulations.

2.4 Key Challenges

Combining an on-premise storage infrastructure with external cloud storage resources in a so-called federated cloud storage architecture has emerged as a promising yet a challenging solution to a number of these storage- and privacy-related requirements.

However, developing and maintaining a multi-tenant SaaS application that leverages the benefits of a federated cloud storage architecture essentially involves dealing with the following key challenges:

C1- Decouple complexity from SaaS applications

C1.A- Abstraction of federated cloud storage. A multi-tenant SaaS application that leverages the benefits of a federated cloud storage has to deal with the complexity of various heterogeneous database systems (both SQL and NoSQL) and technologies in the big data landscape (e.g., batch processing, stream processing, search engine, etc.) usually operating at different cloud storage providers or Database-as-a-Service providers, while also considering their respective data models, APIs, and query languages. The application needs to provide sufficient abstractions to hide the complexity involved in managing (i.e. partitioning, integrating, updating, deleting, searching, replicating, indexing, etc.) application data across federated cloud storage architectures. In addition, the application must be able to interact uniformly with a wide variety of different vendor-specific database systems and technologies in the big data landscape.

C1.B- Externalization of federated cloud storage logic. When building a multi-tenant SaaS application on top of a federated cloud storage setup, hard-coding specifics, which lead to the development of a complex storage logic for addressing different types of the application data and the dynamic requirements of tenants is clearly not a good idea. It drastically impedes the ability of the multi-tenant SaaS application to satisfy contrasting requirements of various tenants, work with different cloud storage providers and their respective storage systems, as well as evolve over time. Hence, managing and operating the federated cloud storage logic is ideally an operational concern (deployment and/or configuration) and not a development concern. Therefore, hard-coding specifics of the federated cloud storage logic must be externalized from the SaaS application layer.

C2- Address dynamic requirements of tenants

C2.A- Data storage requirements. In the case of multi-tenant SaaS applications, different tenants have slightly different data storage requirements with respect to performance, scalability, availability, etc., for different types of application data. For example, data elements of certain data types that require better performance must be stored in a data store that sufficiently complies with such storage requirements. Similarly, data elements of certain data types require higher availability and they must be treated accordingly. This form of diversity that needs to be supported for each tenant of the SaaS application at the level of individual data elements makes the SaaS application less manageable and scalable as well as inadequately adaptable for future changes. The multi-tenant SaaS application needs to accommodate these diverse and dynamic data storage requirements of tenants in an efficient, scalable, and continuous manner.

C2.B- Data confidentiality requirements. Similar to the data storage requirements, tenants also have different confidentiality requirements, which may differ considerably among them. For example, a banking industry encounters stricter regulations and constraints on data confidentiality than a supermarket. In addition, different tenants of the multi-tenant SaaS application may impose confidentiality requirements at varying levels of granularity (e.g., from data types to data elements and up to the finest level, i.e., at the level of individual data properties of an object), which can be applied at run time. The application has to fulfill these contrasting confidentiality requirements of tenants to be inline with these regulations and constraints.

C3- Tenant self-service/customization

C3.A- Data storage configuration. The application must allow customer organizations (tenants) to configure the service to use their own on-premise storage infrastructure. For example, a medical institution (i.e. a hospital) may not want to disclose sensitive patient data to the service provider or may not even want the physical storage to leave the hospital premises.

C3.B- Data storage logic. There is a growing need to customize the multi-tenant SaaS application and more specifically, its storage decisions to accommodate the requirements of different tenants. Therefore, the middleware needs to be customizable and configurable for and by individual tenants of the SaaS application to enable self-service offering [30] and meet the varying requirements of individual tenants. More concretely, the middleware should allow tenants to express their data storage- and privacy-related requirements and enforce these requirements at run time (i.e. allowing tenants to customize these storage decisions) up to the level of individual requests.

C4- Back-end portability of multi-tenant SaaS applications. Portability is the ability to easily use applications on different environments (e.g., storage systems operating at different cloud storage providers in a federated cloud storage setup) as such, it enables SaaS providers to port an existing SaaS application to different back-end environments with minimal changes. The application needs to be sufficiently flexible and adaptable to allow dynamic (re)configuration of the underlying back-end storage architecture (i.e. cloud storage providers and thus also the storage systems), without requiring modifications to the application source code.

C5- Practical feasibility. The solution that addresses the above-mentioned challenges must ensure that these challenges are addressed in a feasible and acceptable manner, i.e., the performance overhead should be minimal.

The next section presents the middleware architecture and its key components and further, it discusses how these challenges are addressed by different components of the PERSIST middleware.

3 PERSIST: a federated cloud middleware

PERSIST provides a uniform API to enable the integration of services and resources in an on-premise infrastructure with resources from external cloud storage providers (e.g., Database-as-a-Service providers). Thus, it makes it easy for tenants to achieve the composite benefits of both worlds: strong data security in the private cloud, while the unlimited resources and different feature strengths in the public cloud without having to be concerned about the underlying technical details.

This section provides an in-depth overview of the design decisions and the architecture of our PERSIST middleware. At its foundation, PERSIST relies upon the following design principles:

- It makes abstraction of the underlying federated cloud storage architecture, allowing the definition of separate storage configuration files (**C1.A**).
- Federated cloud storage complexity is externalized from the application source code in separate external data storage policies (**C1.B**).
- It addresses dynamic data storage requirements of tenants at a fine-grained level, i.e. tenants can enforce different and often conflicting storage- and privacy-related requirements for different data types at the level of individual data elements (**C2.A**).
- It supports run-time cross-database and cross-provider polyglot persistence on a per-object level, i.e. tenants can use multiple and any combination of data storage technologies, resources, and cloud storage providers for different data elements in a federated cloud setup and the decision to use such a combination can be specified at run time (**C2.A**).
- It offers dynamic support for encryption of sensitive data at differing levels of granularity and thus allows the definition of metadata and policies that involve such measures (**C2.B**).
- It enables tenants to configure and use their own storage resources (i.e. an own on-premise storage infrastructure) for performance (e.g., latency and throughput) and security reasons (**C3.A**).
- It supports multi-tenancy, more specifically tenant customization, i.e. allowing tenants to customize both data storage policies and storage configurations —at the level of individual data elements— to their specific requirements (**C3**).
- Policy decisions and tenant configurations are enabled at run time, at the finest level of detail (**C3**).
- It ensures back-end portability of SaaS applications to different cloud storage providers without any changes in the application code (**C4**).

Figure 2 depicts the architecture of the PERSIST middleware. The remainder of this section presents in further detail the different layers, subsystems, and components as well as discusses how the five key requirements (**C1 - C5**) listed above are addressed.

3.1 Tenant Configuration

Presented at the top of Fig. 2, this collection of tenant-specific artifacts logically groups the tenant customizations that are supported by the middleware. Tenants can customize the data storage policies and data storage configurations to their own specific needs. As an example, tenants can configure the middleware to use their own on-premise storage infrastructure by providing an external storage configuration file in the form of deployment descriptor that contains the deployment description of the back-end storage setup. In addition, tenants can provide their own cryptographic keys that can be used by one of the main components of the PERSIST middleware called the secure data management component (cf. Section 3.4.2), which provides a support for encryption of sensitive data at different levels of granularity. These customizations can be provided to the middleware at run time.

Multi-Tenant LMaaS Illustration In order to address varying and individual requirements of different tenants of the SaaS application, tenants of the LMaaS application can refine SaaS provider policies (see Listing 2 for a simple example of the application-wide policy) by specifying tenant-specific data storage policies, as exemplified in Listing 1. PERSIST takes these tenant-specific policies into consideration for addressing different requirements of tenants by overriding the policies of the SaaS application (e.g., the application-wide policies).

As shown in Listing 1, tenants can effectively refine policies by specifying some additional constraints tailored to their own specific requirements. For example, to enable high availability, better disaster recovery (where a single cloud provider might anytime fail or become unavailable), and the cross-provider polyglot persistence (i.e. where a specific data element is stored across multiple cloud storage providers, the tenant specifies cross-provider replication constraint as the tenant wants to replicate the specific data element across 2 cloud storage providers as shown in line 11 (partially addresses **C2.A**). Similarly, to ensure confidentiality of sensitive data before storing to external untrusted cloud storage providers, the tenant can enable data encryption support by setting the corresponding property value to true (as shown in line 7). In addition, as shown in line 10, the tenant prefers to use public cloud storage providers (e.g., Database-as-a-Service providers) over maintaining an expensive own on-premise storage infrastructure and also prioritizes search strategies that first act on private storage infrastructure for performance reasons (line 12).

3.2 SaaS Application

The *SaaS Application*, as shown in Fig. 2, the second layer from the top, logically groups the SaaS application as well as its global data storage policies and configurations. SaaS providers are given the ability to specify the application-wide persistence configuration in the form of deployment descriptor, which is a declarative model of the back-end federated cloud storage architecture. This includes meta-data information about data stores such as their deployment environment (e.g., data stores are deployed in a private cloud or external public cloud storage providers) and technical capabilities (e.g., data stores optimized for write vs. read throughput, data stores best suited for storing specific types of application data, etc).

In addition, SaaS providers can specify application-wide data storage policies, which are defined as sets of rules, each containing a set of conditions and the requirements about the technical capabilities of data stores deployed in different cloud storage providers (both private and public clouds) to satisfy these conditions. These data storage policies and the persistence configuration details are application-wide and therefore are global and apply to all tenants of the SaaS application.

Multi-Tenant LMaaS Illustration Listing 3 shows an example of an application-wide persistence configuration, which contains the configuration details of the back-end data stores (MongoDB and Cassandra), deployed in a federated cloud storage setup (both in a private cloud storage infrastructure as well as public cloud storage providers). As shown, the persistence configuration details are specified for the Apache Cassandra data store from lines 1 – 16 (deployed and managed in a private cloud) and lines 17 – 32 (deployed in a public cloud storage provider). The persistence configuration details for the MongoDB data store deployed in a private cloud storage environment and the public cloud provider are specified from lines 33 – 48 and lines 49 – 64 respectively. In addition to configuration details, each storage configuration also includes additional properties, which specify meta-data information about the data store (deployment environment and technical capabilities). For example, the meta-data information about the Cassandra data store deployed in an on-premise infrastructure is specified from lines 8 – 15 and the MongoDB data store deployed in the public cloud is specified from lines 56 – 63. These additional properties play a key role, as they describe characteristics and the deployment environment of each storage system and enable PERSIST to make appropriate data placement decisions.

Similarly, a simplified example of an application-wide data storage policy, which is global to all tenants is shown in Listing 2. The policy contains a set of rules for storing both confidential and non-confidential data of types *raw log* entries, *historical logs*, and *log meta-data*. Each rule defines a number of conditions and

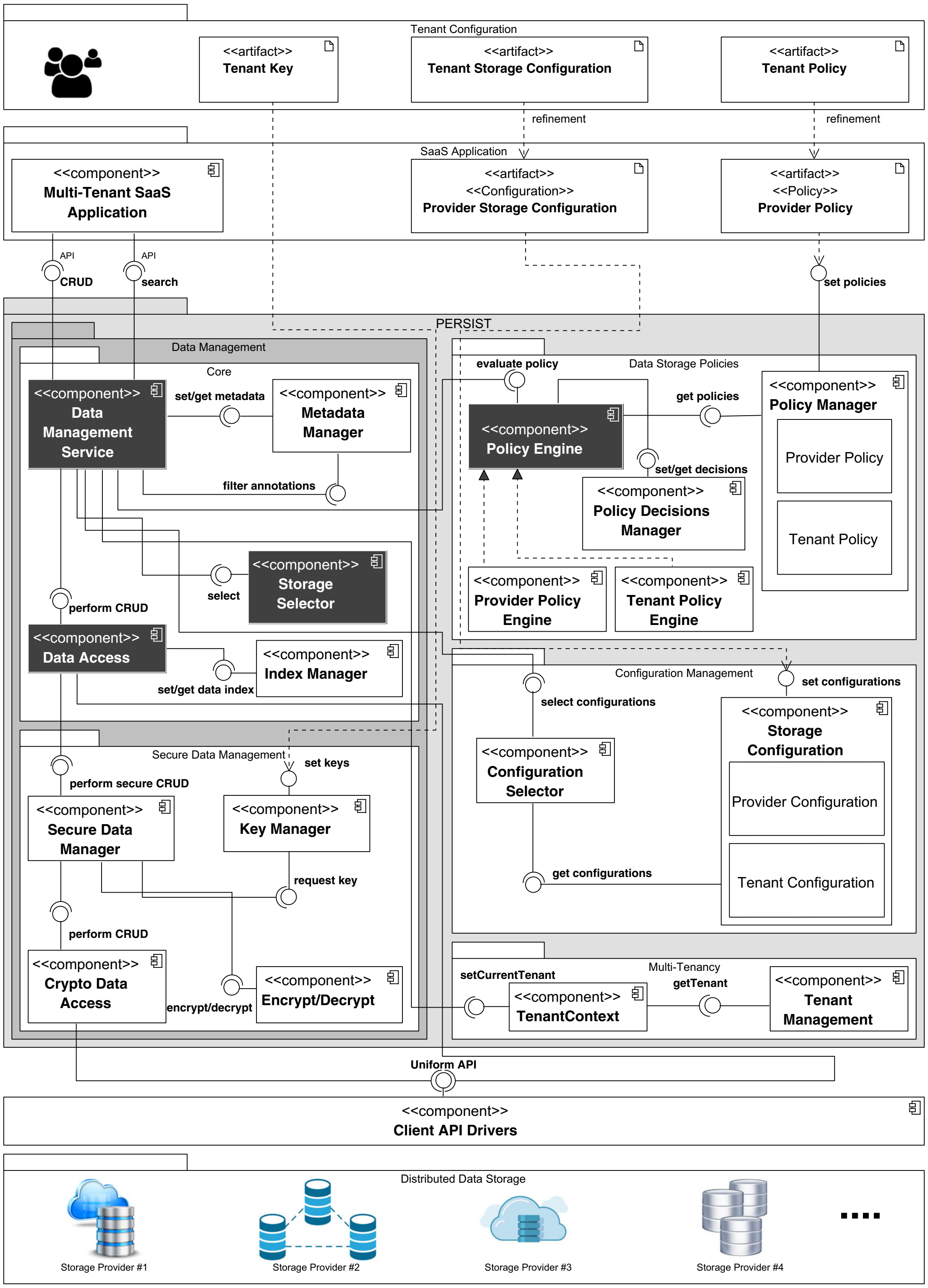

Tenant Configuration
<<artifact>> Tenant Key
<<artifact>> Tenant Storage Configuration
<<artifact>> Tenant Policy
refinement
refinement
SaaS Application
<<component>> Multi-Tenant SaaS Application
<<artifact>> <<Configuration>> Provider Storage Configuration
<<artifact>> <<Policy>> Provider Policy
API CRUD
API search
set policies
PERSIST
Data Management
Core
<<component>> Data Management Service
set/get metadata
<<component>> Metadata Manager
filter annotations
select
<<component>> Storage Selector
perform CRUD
<<component>> Data Access
set/get data index
<<component>> Index Manager
Secure Data Management
set keys
perform secure CRUD
<<component>> Secure Data Manager
<<component>> Key Manager
request key
perform CRUD
<<component>> Crypto Data Access
encrypt/decrypt
<<component>> Encrypt/Decrypt
Data Storage Policies
evaluate policy
get policies
<<component>> Policy Engine
set/get decisions
<<component>> Policy Decisions Manager
<<component>> Policy Manager
Provider Policy
Tenant Policy
<<component>> Provider Policy Engine
<<component>> Tenant Policy Engine
Configuration Management
set configurations
select configurations
<<component>> Configuration Selector
<<component>> Storage Configuration
Provider Configuration
Tenant Configuration
get configurations
Multi-Tenancy
setCurrentTenant
getTenant
<<component>> TenantContext
<<component>> Tenant Management
Uniform API
<<component>> Client API Drivers
Distributed Data Storage
Storage Provider #1
Storage Provider #2
Storage Provider #3
Storage Provider #4
....

Fig. 2 Architecture of the PERSIST middleware, which combines an on-premise infrastructure with external cloud providers and offers a single uniform API for data management across a federated cloud storage architecture

```
...
rule "Dealing with the confidential data of
    type raw log"
when
dsSelector:Entity(data == "Confidential",
    type == "Rawlog")
then
...
dsSelector.supportsCondidentiality(true);
dsSelector.writeThroughput(high);
dsSelector.readThroughput(high);
dsSelector.storagePreference("public");
dsSelector.crossProviderReplication(2);
dsSelector.searchPreference("private");
...
end
...
```

Listing 1 A tenant-specific data storage policy overrides the SaaS provider policy which specifies additional preferences tailored to its own requirements for dealing with confidential *raw log* entries

different requirements that have to be met for storing data of such data type. These data storage policy rules partition different types of the application across different cloud storage providers based on different data storage requirements without the need to (re)write the storage logic in the application source code (partially addresses **C1.B**). Furthermore, changing the back-end storage architecture to address specific requirements of new customer organizations (tenants) only induces updating the technical capabilities in storage configurations (partially addresses **C1.B**). Finally, supporting dynamic requirements within the application only involves defining additional policy rules.

3.3 Multi-Tenancy

The *Multi-Tenancy* subsystem depicted in the bottom right supports tenant-aware execution and tenant customization of the SaaS application, similarly to [49].

The `TenantContext` component enables the `Data Management Service` component to set information about tenant linked to or piggybacked on data requests (via the unique tenant ID). The component communicates with the `Tenant Management` component to authenticate the tenant.

3.4 Data Management

The *Data Management* subsystem represents the core part of the PERSIST middleware. As shown in Fig. 2, this subsystem primarily consists of two subsystems:

```
rule "Dealing with the confidential data of
    type raw log"
when
dsSelector:Entity(data == "Confidential",
    type == "Rawlog")
then
dsSelector.supportsCondidentiality(true);
dsSelector.writeThroughput(high);
dsSelector.readThroughput(high);
dsSelector.datastoreReplication(1);
end

rule "Dealing with the non-confidential data
    of type raw log"
when
dsSelector:Entity(data == "Nonconfidential",
    type == "Rawlog")
then
dsSelector.supportsCondidentiality(false);
dsSelector.writeThroughput(high);
dsSelector.readThroughput(high);
dsSelector.datastoreReplication(1);
end

rule "Dealing with the confidential data of
    type historical log"
when
dsSelector:Entity(data == "Confidential",
    type == "Historicallog")
then
dsSelector.supportsCondidentiality(true);
dsSelector.readThroughput(high);
end

rule "Dealing with the non-confidential data
    of type historical log"
when
dsSelector:Entity(data == "Nonconfidential",
    type == "Historicallog")
then
dsSelector.supportsCondidentiality(false);
dsSelector.readThroughput(high);
end

rule "Dealing with the confidential data of
    type log meta-data"
when
dsSelector:Entity(data == "Confidential",
    type == "Meta-data")
then
dsSelector.supportsCondidentiality(true);
dsSelector.writeThroughput(high);
dsSelector.datastoreReplication(1);
end

rule "Dealing with the non-confidential data
    of type log meta-data"
when
dsSelector:Entity(data == "Nonconfidential",
    type == "Meta-data")
then
dsSelector.supportsCondidentiality(false);
dsSelector.writeThroughput(high);
dsSelector.datastoreReplication(1);
end
```

Listing 2 Application-wide data storage policy to deal with multi-tenant LMaaS application data

```
<storage name="Cassandra-Private">
  <datastore>Cassandra</datastore>
  <nodes>private-ip-address</nodes>
  <port>port-no</port>
  <keyspace>LMaaS</keyspace>
  <username>username</username>
  <password>password</password>
  <properties>
    <location>private</location>
    <trusted>true</trusted>
    <writethroughput>high</writethroughput>
    <readthroughput>high</readthroughput>
    <data>Rawlog & Meta-data</data>
    ...
  </properties>
</storage>
<storage name="Cassandra-Public">
  <datastore>Cassandra</datastore>
  <nodes>public-ip-address</nodes>
  <port>port-no</port>
  <keyspace>LMaaS</keyspace>
  <username>username</username>
  <password>password</password>
  <properties>
    <location>public</location>
    <trusted>false</trusted>
    <writethroughput>high</writethroughput>
    <readthroughput>high</readthroughput>
    <data>Rawlog & Meta-data</data>
    ...
  </properties>
</storage>
<storage name="MongoDB-Private">
  <datastore>MongoDB</datastore>
  <nodes>private-ip-address</nodes>
  <port>port-no</port>
  <keyspace>LMaaS</keyspace>
  <username>username</username>
  <password>password</password>
  <properties>
    <location>private</location>
    <trusted>true</trusted>
    <writethroughput>low</writethroughput>
    <readthroughput>high</readthroughput>
    <data>Archievedlog&Historicallog</data>
    ...
  </properties>
</storage>
<storage name="MongoDB-Public">
  <datastore>MongoDB</datastore>
  <nodes>public-ip-address</nodes>
  <port>port-no</port>
  <keyspace>LMaaS</keyspace>
  <username>username</username>
  <password>password</password>
  <properties>
    <location>public</location>
    <trusted>false</trusted>
    <writethroughput>low</writethroughput>
    <readthroughput>high</readthroughput>
    <data>Archievedlog&Historicallog</data>
    ...
  </properties>
</storage>
```

Listing 3 Configuration details of back-end data storage systems distributed in a federated cloud setup

(i) The `Core` subsystem, which is the heart of the PERSIST middleware and deals with data storage requests, matches these requests to application-wide and tenant-specific data storage policies, and finally performs a lookup of the appropriate storage node(s) to which the request is then propagated.

(ii) The `Secure Data Management` subsystem transparently extends the behavior of the `Core` subsystem with a support for encryption-enabled CRUD transactions (i.e. encrypting confidential data before storing), when this is called for as a result of policy evaluation.

The detailed behavior of both these subsystems is further discussed in the following section.

3.4.1 Core

The `Data Management Service` component provides an API for SaaS applications to interact with the PERSIST middleware, enabling the execution of insert, read, update, delete (CRUD) transactions, encryption-enabled CRUD transactions, and the search operation.

A high-level overview of how different components of the PERSIST middleware interact for making data placement (i.e. data storage) decisions across a federated cloud storage setup are illustrated in Fig. 3. Upon arrival of data placement requests, this component first reads the tenant information, i.e., the tenant ID associated with each data storage request and calls the `TenantContext` component, the component of the *Multi-Tenancy* subsystem to set the current state of the tenant, i.e. the tenant ID (step 1. in Fig. 3). Then, the component interacts with the `Metadata Manager` component to check if the metadata of the incoming data element (i.e. an entity object) for a particular tenant already exists (step 2.1. in Fig. 3). PERSIST stores metadata separately from the application data. Metadata is stored within one of the core components of the middleware called the `Metadata Manager` component. This component supports both the memory-only storage and the persistent storage, and also offers metadata replication support and in our architecture, remains on private premises (i.e. on-premise).

The `Data Management Service` component only communicates with the `Metadata Manager` component to filter the annotations and read the metadata of the incoming data element (i.e. an entity object) for a particular tenant if it does not already exist in the `Metadata Manager` component or when cached values have been invalidated or have become stale (step 2.2. in Fig. 3). The latter component allows the `Data Management Service` component to avoid inspecting the same entity object multiple times for performance reasons, which involves filtering the annotations as well as reading the meta-data (addresses **C5**).

The `Data Management Service` component then calls the `Policy Engine` component of the *Data Storage Policies* subsystem to evaluate the data storage policies by passing the tenant information, i.e., the tenant ID and the metadata of the incoming data element (i.e. an entity object), returned by the `Metadata Manager` component (step 3. in Fig. 3). Afterwards, the `Data Management Service` component calls the `Configuration Selector` component of the *Configuration Management* subsystem by passing the tenant ID to get the configuration details of different back-end storage systems supported by different cloud storage providers, set by a particular tenant (step 4. in Fig. 3). After receiving the required information from both components (the `Policy Engine` component and the `Configuration Selector` component), the `Data Management Service` component calls the `Storage Selector` component to perform a lookup of the specific back-end nodes and make an appropriate data placement decision (step 5. in Fig. 3). The information is used by the `Storage Selector` component of the PERSIST middleware to determine the storage locations for the data placement decisions. Based on the returned information from the `Policy Engine` component and the `Configuration Selector` component, the `Storage Selector` component then decides where the data needs to be stored in a federated cloud storage architecture. The data placement decision across a federated cloud storage architecture is made by taking into account a number of different considerations, such as, the performance objective (e.g., write vs. read throughput), cross-provider data replication, whether the data has to be encrypted before storage, etc.

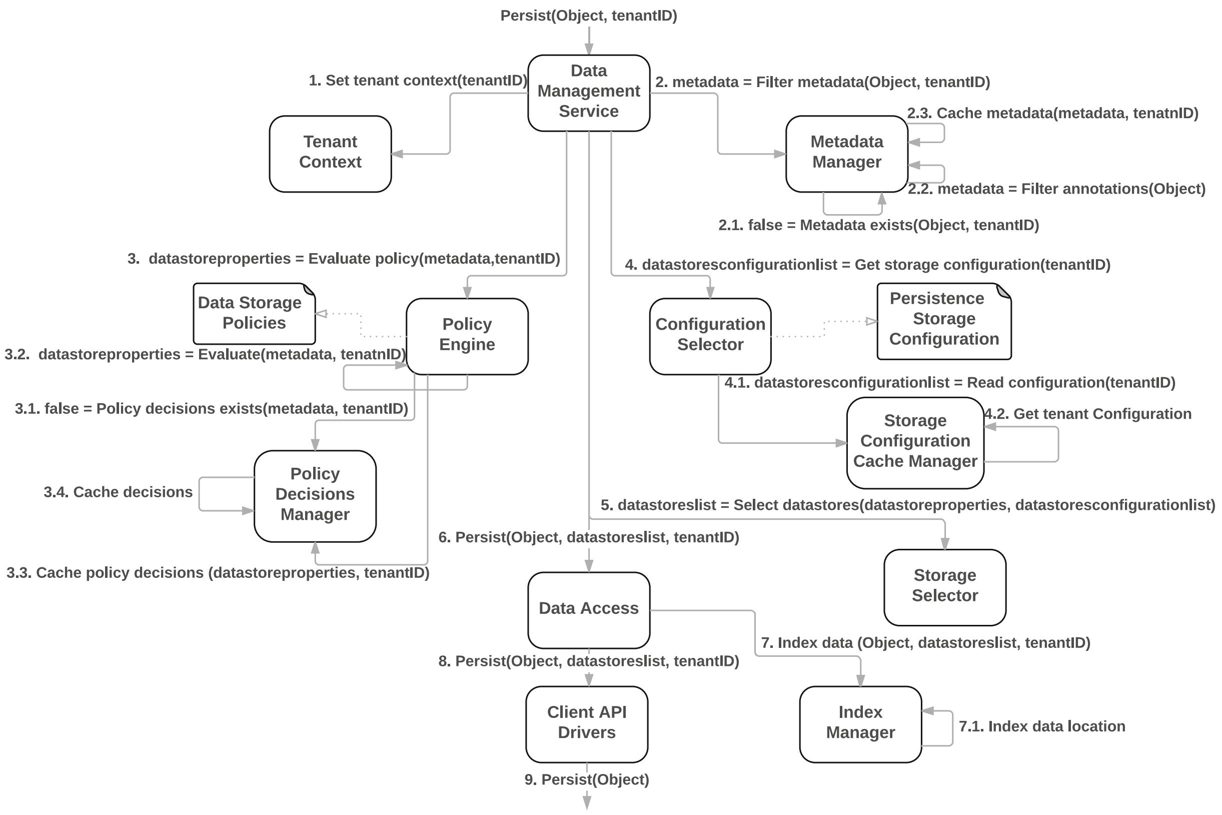

Persist(Object, tenantID)
Data Management Service
1. Set tenant context(tenantID)
Tenant Context
2. metadata = Filter metadata(Object, tenantID)
Metadata Manager
2.3. Cache metadata(metadata, tenatnID)
2.2. metadata = Filter annotations(Object)
2.1. false = Metadata exists(Object, tenantID)
3. datastoreproperties = Evaluate policy(metadata,tenantID)
4. datastoresconfigurationlist = Get storage configuration(tenantID)
Data Storage Policies
Policy Engine
3.2. datastoreproperties = Evaluate(metadata, tenatnID)
3.1. false = Policy decisions exists(metadata, tenantID)
Configuration Selector
Persistence Storage Configuration
4.1. datastoresconfigurationlist = Read configuration(tenantID)
Storage Configuration Cache Manager
4.2. Get tenant Configuration
3.4. Cache decisions
Policy Decisions Manager
5. datastoreslist = Select datastores(datastoreproperties, datastoresconfigurationlist)
6. Persist(Object, datastoreslist, tenantID)
3.3. Cache policy decisions (datastoreproperties, tenantID)
Storage Selector
Data Access
7. Index data (Object, datastoreslist, tenantID)
8. Persist(Object, datastoreslist, tenantID)
Client API Drivers
Index Manager
7.1. Index data location
9. Persist(Object)

Fig. 3 High-level overview of the requests flow in the PERSIST middleware to performance an insert transaction

The latter component realizes policy decisions, which matches the desired data store selection properties with the meta-data information (including both the deployment environment and the technical capabilities) of each data store for making data placement decisions. The component, based on the policy decisions, is responsible for selecting the data store(s) or cloud storage providers best suited for storing a part of the application or the tenant-specific data in a federated cloud storage setup and locating the data back when data from a single tenant can be distributed over cloud storage providers. This component returns the list of data store nodes, distributed across multiple cloud storage providers, each responsible for storing a part of the application data for a particular tenant.

The `Data Management Service` component passes the data element (i.e. an entity object) that needs to be stored; along with tenant information, i.e., the tenant ID; and the list of persistence configurations to the `Data Access` component (step 6. in Fig. 3). Before communicating with the `Client API Drivers` component to store the data across different data stores in a federated cloud setup, the `Data Access` component calls the `Index Manager` component and passes the information such as data element (i.e. an entity object), the tenant ID, and the list of persistence configurations (step 7. in Fig. 3). The `Index Manager` component uses the information to index the data storage locations for each tenant (step 7.1. in Fig. 3). The component helps tenants to read/search the data back from the federated cloud storage setup in an efficient manner such as, without the need to communicate with all the data stores distributed across various cloud storage providers or Database-as-a-Service providers. In addition, the component also helps tenants to update and delete data in an efficient way (i.e. by directly communicating with the storage node(s), which hold the data).

The `Data Access` component then iterates over the resulting list of data stores to get the connection details (e.g., private or public IP address, port, username, password, etc.), which are required to remotely connect these back-end data stores. Moreover, the component also checks whether the encryption is required before storing data. If encryption is not required, it interacts directly with the `Client API Drivers` component, which offers a uniform API and passes the persistence storage configuration details, tenant information (i.e. the tenant ID), and the data element (i.e. an entity object) which is to be persisted (step 8. in Fig. 3).

In case the encryption is required, the component communicates with the `Secure Data Manager` component of the *Secure Data Management* subsystem and passes the persistence storage configuration details, tenant information (i.e. the tenant ID), and the data element (i.e. an entity object) that needs to be encrypted before persisting in the back-end data store(s).

3.4.2 Secure Data Management

PERSIST facilitates tenants to have fine-grained control over data encryption by allowing them to specify confidentiality requirements at different levels of granularity (addresses **C2.B**) for different data elements. The responsible component to provide such a support in the PERSIST middleware is the `Secure Data Manager` component, which to accomplish this, uses the data mapping strategy motivated in [33] and [37].

After getting the request from the `Data Access` component, the `Secure Data Manager` component first determines the levels of granularity the data should be dealt with confidentially; for example, whether the entire data element (i.e. all properties of an entity object) is specified as confidential by the tenant or some properties of an entity object contain confidential information. The `Secure Data Manager` component determines the underlying data mapping strategy based on the levels of granularity at which the encryption is required. For example, if the entire data element is specified as confidential, the underlying data mapping strategy is different and thus confidential and non-confidential data is stored separately. Similarly, if some properties of an entity object are specified as confidential, the underlying data mapping strategy is again different and thus a part of the confidential data is stored together with the non-confidential data, while the other part of the confidential data is stored separately.

In both cases (whether the entire data element or some properties of the data element), data encryption is required. However, performing encryption at a fine-grained level (e.g. encrypt only some properties of an entity element) still enables inexpensive search operations on an unencrypted data.

In case the entire data element (e.g. an entity object) is considered to be confidential, search functionality can still be provided, however, has become highly expensive in terms of performance, as the middleware has to first scan all the data elements and then decrypt the entire data sets from the federated cloud storage setup (combination of both private and public cloud providers).

To enable encryption, the `Secure Data Manager` component requests the corresponding `Tenant Key` from the `Key Manager` component and passes the entity object, which requires encryption and the tenant-specific secure key to the `Encrypt/Decrypt` component. As explained in Section 3.1, the `Key Manager` provides an interface for tenants to set tenant-specific encryption keys. These are used by the components of the middleware to encrypt and decrypt tenant-specific confidential data. In the current implementation of the PERSIST architecture, advanced key management use cases (e.g. versioning or revoking tenant keys, or using combinations of keys for data encryption for a specific tenant) are not yet supported.

After getting the request from the `Secure Data Manager` component, the `Encrypt/Decrypt` component performs the encryption process to encrypt the entity object. Once its encrypted, the component returns the encrypted entity object to the `Secure Data Manager` component. The latter component interacts with the `Crypto Data Access` component to perform encryption-enabled CRUD transactions. The `Crypto Data Access` component, in turn, communicates with the `Client API Drivers` component and passes the persistence storage configuration details, the tenant ID, and the encrypted data element (i.e. encrypted entity).

3.5 Data Storage Policies

The *Data Storage Policies* subsystem in the middleware enables tenants and the SaaS provider to specify data storage policies for managing different types of data generated by the SaaS application. The system is responsible for decoupling storage logic from the application layer and therefore, allowing the application to be modified without (re)implementing and (re)compiling the application source code (addresses **C1.B** and **C3.B**). In addition, the system hides the complexity of managing tenant's data and addresses different data storage requirements of tenants. The system offers several interfaces to the SaaS provider, tenants, and other components of the middleware.

The `Policy Engine` is a key component of the `Data Storage Policies` system. It is responsible for the policy evaluation upon request of the `Data Management Service` component. After receiving the request from the `Data Management Service` component, the `Policy Engine` first extracts the tenant information, i.e., the tenant ID and the metadata of an entity object associated with each data management request. The `Policy Engine` then communicates with the `Policy Manager` component by passing the tenant ID to get data storage policies for a particular tenant.

The `Policy Manager` component is responsible for managing policies for the SaaS provider and its tenants. The component provides an interface for the SaaS provider to set the application-wide storage policy, which is global and applies to all tenants of the SaaS application. After receiving the request from the `Policy Engine` component, the `Policy Manager` component first checks if the policy for a particular tenant is available, which means the tenant has effectively defined a policy refinement. If so, the component returns both policies (i.e. the application-wide and the tenant-specific) to the `Policy Engine` component. These policies are then executed by two different sub-components of the `Policy Engine` component, the `Provider Policy Engine` component and the `Tenant Policy Engine` component. These sub-components execute the policies based on the metadata of an entity object. The object's metadata is then compared with all the rules, defined in the policy files and their corresponding action is executed. The action part of the policy rules contains data store selection properties (i.e. conditions and the requirements about the technical capabilities of data stores deployed in different cloud storage providers). After the policy execution, the `Policy Engine` component caches policy decisions by calling the `Policy Decisions Manager` component (step 3.4. in Fig. 3). However, before executing the policy, the `Policy Engine` component first requests the `Policy Decisions Manager` component to check if the SaaS provider's or a particular tenant's policy decisions are already cached (step 3.3. in Fig. 3). The latter is a performance measure to avoid frequent re-execution of these policies for the same tenant, by caching the actual policy decisions (addresses **C5**).

3.6 Configuration Management

The *Configuration Management* subsystem is responsible for managing the deployment configuration of the back-end storage architecture of the federated cloud setup (including both private and public cloud storage providers). In addition, it provides an access to different data stores, deployed across multiple cloud providers.

The `Configuration Selector` component allows the `Data Management Service` component to look up the configuration details of a specific back-end data storage node. After receiving the request from the `Data Management Service` component, the `Configuration Selector` component extracts the tenant information, i.e., the tenant ID and communicates with the `Storage Configuration` component to get the configuration details for a particular tenant. The latter component is responsible for storing these persistence configuration details (both application-wide and tenant-specific). The component also provides an interface to the SaaS provider to set application-wide persistence configurations (cf. Section 3.2) and enables tenants to refine these configurations by providing tenant-specific persistence configuration details (cf. Section 3.1). Thus, it enables the SaaS provider and tenants to flexibly change the storage configurations and easily port an existing application to different back-end storage setup (addresses **C4**). The SaaS provider and its tenants define these configuration details in the form of a configuration file (partially addresses **C3.A**). Along with the persistence configuration details about the back-end database technologies, this configuration file also defines meta-data information (i.e. additional properties) about the deployment environment and the technical capabilities of the database technology or its specific instances (cf. Section 3.2).

3.7 Client API Drivers

The `Client API Drivers` component (depicted at the bottom of Fig. 2) groups different storage drivers, at least one for each supported data store. In addition, it makes abstraction by providing a uniform API to the `Data Access` component and the `Crypto Data Access` component (addresses **C1.A**). This is highly similar to the existing design of object NoSQL database mappers (ONDMs) such as Impetus Kundera [27] or Hibernate OGM [23]. Each driver in the `Client API Drivers` (not depicted) acts as a client communicating with the specific storage system to issue native CRUD transactions and search operations. As such, the `Client API Drivers` component support portability across different back-end systems and technologies (addresses **C4**).

4 Prototype Implementation

We have validated the PERSIST middleware presented in the previous section in an extensive prototype, which is built upon existing open source tools and technologies[1]. More specifically, the prototype is an extension of Impetus Kundera [27], which is an open source data access middleware for NoSQL data stores (as such an implementation of the `Client API Drivers` component of our middleware architecture presented in Fig. 2). Our motivation for selecting Impetus Kundera as the foundation of our PERSIST implementation is that an earlier study [35] has shown that (i) of the existing data access middleware platforms, it introduces the least performance overhead, and (ii) the cost of porting an application in terms of lines of code (re)written to different back-end data stores is minimal.

As a consequence of this decision, the prototype supports the same range and variety of data store technologies as Kundera: in-memory data stores such as Redis, full-text search systems such as Elasticsearch, data processing systems such as Apache Spark, SQL-based databases such as MySQL, and NoSQL data stores: Oracle NoSQL, Apache Cassandra, MongoDB, Apache HBase, Neo4j, Apache CouchDB.

The prototype uses the JBoss Drools[2] policy engine for the evaluation of data storage policies. Drools is an open source, object-oriented rule engine written in Java that uses rule-based approach to implement

[1]The prototype implementation is available at: http://people.cs.kuleuven.be/∼ansar.rafique/PERSIST.zip.

[2]http://www.drools.org/

an expert system. The prototype provides a Java Persistence API (JPA) and a Java Persistence Query Language (JPQL) interface as an abstraction API to interact with various back-end storage systems. These are the de-facto standard APIs for Java [46], offer a number of standardized annotations to developers. As a consequence, different components of the PERSIST middleware implement the interfaces provided by the JPA and the JPQL. For example, the `Secure Data Manager` component (cf. Section 3.4.2) implements the `Entity Manager` [31] interface of the JPA. Thus, the component overrides all methods of the `Entity Manager` interface with a support to execute secure (i.e. encryption-enabled) CRUD transactions. In addition to annotations offered by the JPA and the JPQL standards, the additional meta-data is implemented by means of additional *application-specific*, *technology-specific*, and *middleware-specific* annotations.

In the multi-tenant LMaaS offering (cf. Section 2.1), the application defines annotations such as `@RawLog`, `@HistoricalLog`, `@IncidentReport`, `@Size`, etc., which are application-specific. Examples of technology-specific annotations are `@Writeconsistency`, `@Readconsistency`, `@Searchconsistency`, etc., which are supported for multiple data stores such as Apache Cassandra and MongoDB. Finally, the middleware-specific annotations are reusable annotations such as `@PersistEntity`, `@MetaInfo`, `@PersistConfidential(members={})`, `@FullTextSearch` etc. These annotations are supported at both the class-level and the field-level. As shown in Listing 4, the `@PersistEntity`, `@MetaInfo`, and `@PersistConfidential(members={})` annotations are used on a class-level, while the `@FullTextSearch` annotation is used on a field-level to specify additional information about the class and its properties. This additional information needs to be specified for the PERSIST middleware to take certain requirements into account for making data management decisions (e.g., CRUD transactions).

The prototype is deployed as a service on Tomcat 7[3] with an exposed configuration dashboard for both the SaaS provider/operator and its tenants. When

```
/**
 * Confidential entity of type raw log.
 */
package com.distrinet.lmaas.entities;
import com.distrinet.persit.PersistEntity;
import com.distrinet.persit.FullTextSearch;
...

@PersistEntity({
  @MetaInfo(key="type",value="Rawlog"),
  @MetaInfo(key="data",value="Confidential")
})
@PersistConfidential(members={"deviceName"})
public class Log implements Serializable {
    ...

    @FullTextSearch
    private String customerName;
    private String deviceName;

    ...
}
```

Listing 4 Annotations provided by the PERSIST middleware are supported on both the class-level and the field-level

the service starts up, it first reads the application-wide configuration files and the data storage policy files. The service allows tenants to customize these application-wide configuration files and the data storage policy files by uploading tenant-specific configuration files and/or the data storage policy files.

The meta-data, configuration files, and the policy execution decisions are cached and stored in Ehcache[4]. Ehcache is a pure Java cache, which provides in-memory support, but can also offload data into the permanent persistence storage (i.e. disk/database). Ehcache was adopted because of its features such as scalability and a support for permanent persistence storage. In PERSIST, Ehcache helps the `Policy Engine` component (cf. Section 3.5) to avoid executing the data storage policies multiple times for performance reasons by caching the policy decisions and store in the permanent disk/database storage. Similarly, Ehcache also supports the `Data Management Service` component (cf. Section 3.4.1) to avoid calling the `Metadata Filter` component for inspecting the same entity multiple times by caching the metadata, which requires memory-only storage (data is only stored in the memory). After placing the annotations provided by the PERSIST middleware platform, setting the data storage policy

[3]http://tomcat.apache.org/

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

files and the configuration files, the middleware is ready to serve tenant requests.

5 Evaluation

We have performed complementary evaluations of the PERSIST middleware in three different dimensions. Firstly, Section 5.1 compares the cost/effort required to implement a complex data storage policy (evaluation #1). Secondly, in a similar scenario-driven comparison, Section 5.2 assesses the cost/effort of changing the back-end storage architecture of a federated cloud setup (evaluation #2). Finally, Section 5.3 evaluates the performance impact, more specifically the performance overhead introduced by the PERSIST middleware (evaluation #3). Then, in Section 5.4, we discuss implications of our findings and the overall choices made for the evaluations.

These evaluations are conducted in the context of our implementation of the multi-tenant Log Management-as-a-Service (LMaaS) application, discussed in Section 2.1. We employ Impetus Kundera as the baseline for comparison since extensions of the PERSIST middleware were formed on top of the Kundera platform.

5.1 Cost/Effort to Implement Complex Storage Policy

This part of the evaluation compares the required cost/effort to implement a data storage policy, both with and without PERSIST (**C1.B** discussed in Section 2.4). In Section 5.1.1, we provide the details of the experimental application setup, while the results of this evaluation are presented and discussed in Section 5.1.2.

5.1.1 Application Setup

We developed two prototype implementations: one implementation is based on the Impetus Kundera middleware platform (prototype `Baseline`) and the other implementation leverages our proposed middleware platform (prototype `PERSIST`). More specifically, the same storage logic for the multi-tenant Log Management-as-a-Service (LMaaS) application is implemented in both prototype implementations. First, the storage logic is implemented for the prototype `Baseline` implementation in the application source code and then the same storage logic is defined for the prototype `PERSIST` by specifying the external data storage policy file.

The policy fragment of interest (storage logic) for this evaluation —as shown in Listing 2— focuses on the requirements of *raw log* entries, *historical logs*, and *log meta-data*: *raw log* entries require high write and read throughput as well as high availability, for *historical logs* good read throughput suffices, and finally *log meta-data* stand in needs for high write throughput as well as high availability. For all these data types, confidentiality requirements apply and sensitive data elements (i.e. confidential objects) must be encrypted before storage and decrypted after retrieval. In the context of the (simplified) federated storage architecture presented earlier in Listing 3, these requirements imply that (i) confidential and non-confidential *raw log* entries must be stored in Cassandra-Private and Cassandra-Public data stores respectively, but according to the data storage policy *raw log* entries will also be replicated within the data store (see lines 8 and 18 in Listing 2), (ii) confidential *historical logs* shall be stored in the MongoDB-Private data store, whereas non-confidential *historical logs* shall be stored in the MongoDB-Public data store, and finally (iii) confidential and non-confidential *log meta-data* must be stored in a replicated fashion (see lines 43 and 52 in Listing 2) in Cassandra-Private and Cassandra-Public data stores respectively.

We have calculated two metrics that are indicative of the required cost/effort: the number of lines of code (# LoC) that were implemented in the application and the number of lines of policy files (# LoP) that were introduced. In addition, as a consequence of these changes, the effect in terms of application (re)deployment is also considered.

5.1.2 Results

The results of this part of the evaluation are presented in Table 2. As shown, implementing the policy

Table 2 Overview of the # lines of code and # lines of policy added to implement the storage logic in both the prototype `Baseline` and the prototype `PERSIST`

Prototype	# LoC	% LoC	# LoP	(re)Deploy
Baseline	900	78%	–	✓
PERSIST	–	–	53	✗

fragment in the prototype `Baseline` involves 900 lines of application source code (which corresponds to 78% of the multi-tenant LMaaS implementation), and no policy files. The number of lines of code written in the application to implement the policy fragment are higher in the prototype `Baseline` implementation. This is mainly because to address the requirements in the prototype `Baseline` implementation, two different instances of the *EntityManager* from the *EntityManagerFactory* are instantiated to interact with different data stores (e.g., store *raw log* entries and *log meta-data* in the Cassandra data store, whereas *historical logs* in the MongoDB data store). In addition, the storage logic, which distributes a part of the application data across a federated storage architecture is implemented in the application code. Moreover, the confidentiality requirements for *raw log* entries, *historical logs*, and *log meta-data* are also taken into consideration, hence requires writing the encryption logic in the source code.

On the other hand, supporting the above policy fragment in the prototype `PERSIST` is done by defining an external storage policy file, which decouples the storage logic from the application source code. In the PERSIST middleware, we have externalized storage logic, and therefore, the implementation of the policy fragment was done without changing the application source code, but by only introducing 53 lines of policy (LoP), specifically in the application-wide data storage policy file (see introduced LoP in Listing 2). Furthermore, PERSIST has a built-in support for data encryption, which can be enacted at various levels of granularity. Therefore, the confidentiality of the sensitive data is ensured without writing an encryption logic in the application source code.

Additionally, changing the source code at the application layer also requires the prototype `Baseline` to be (re)compiled and (re)deployed to influence these changes, whereas in the prototype `PERSIST`, such policies can be introduced and changed at run time without requiring modifications in the application code.

5.2 Cost/Effort to Change the Back-end Cloud Storage Architecture

The back-end federated storage architecture may change over time for many reasons, for example, because of changes in data storage requirements, variations in performance of different data stores, specific requirements of new customer organizations (tenants) emerge, or even when external factors influence such as cloud storage providers (e.g., Database-as-a-Service providers) change their pricing policies. In general, addressing these evolutions involves changing the underlying data store configurations of the back-end federated storage architecture. In this part of the evaluation, we consider the scenario of effectuating change in the back-end storage architecture of the multi-tenant Log Management-as-a-Service (LMaaS) application (**C4** discussed in Section 2.4). Section 5.2.1 contains the description of the application setup, while results are then shown and discussed in Section 5.2.2.

5.2.1 Application Setup

Similar to the previous evaluation, we implemented the change scenario in the context of the multi-tenant LMaaS application for both prototype implementations: prototype `Baseline` and the prototype `PERSIST`. More specifically, the change scenario we implemented involves switching the back-end data stores for (i) *raw log* entries from the Cassandra data store to the MongoDB data store, (ii) *historical logs* from the MongoDB data store to Cassandra, and (iii) *log meta-data* from Cassandra to MongoDB. Again, we evaluate the required cost/effort by measuring and comparing the affected lines of code (# LoC) and lines of configuration (# LoCf) as well as their effect on the application.

In the prototype `Baseline` implementation, switching the back-end data stores can be done either by modifying the source code at the application layer or updating the configuration file. However, updating the configuration file in the prototype `Baseline` also requires modifying the source code at the application layer. This is mainly because there is a tight coupling between the application source code and the configuration model (i.e. configuration file). Hence, changes in the configuration model easily cause ripple effects to the application source code. However, for both scenarios either modifying the source code at the application layer or updating the configuration file require the application to be (re)compiled and (re)deployed. Therefore, for this part of the evaluation, we have considered both scenarios: applying changes in the application source code (scenario #1) as well as in the configuration file (scenario #2) for the prototype `Baseline` implementation.

5.2.2 Results

The results of this evaluation are presented in Tables 3 and 4 respectively.

As shown in Table 3, implementing the change scenario in the application source code (scenario #1) for the prototype `Baseline` involves modifying 24 lines of source code (which corresponds to 2% of the multi-tenant LMaaS implementation) and no configuration files. Similarly, applying changes in the configuration file (scenario #2) for the prototype `Baseline` involves modifying 20 lines of configuration files as well as 24 lines of source code in the application (which corresponds to 2% of the multi-tenant LMaaS implementation). In both scenarios for the prototype `Baseline` implementation, changes are made in the application source code and, therefore, the application needs to be (re)compiled and (re)deployed.

On the other hand, the PERSIST version of the multi-tenant LMaaS implementation requires no changes in the application source code and involves changing only the application-wide storage configuration file: 4 lines of configuration file. For example, as shown in Listing 5 (in bold and underlined), switching the back-end cloud storage provider for *raw log* entries from Cassandra-Private to MongoDB-Private, *historical logs* from the MongoDB-Private to Cassandra-Private, and *log meta-data* from Cassandra-Private to MongoDB-Private in PERSIST only requires modifying the `data` property for each data store configuration in the configuration model. This scenario requires only 2 lines (15 and 32) of the Listing 5 to be changed. In addition, this scenario can be executed in prototype `PERSIST` at run time, without changing the source code, (re)compiling, or (re)deploying the application.

5.3 Performance Impact

In the decision to use the PERSIST middleware to alleviate the complexity and maximize the benefits of

Table 3 Overview of the modified # lines of code and # lines of configuration to change the back-end storage architecture for both prototype implementations (scenario #1: changes are made in the source code for the prototype `Baseline`)

Prototype	# LoC	% LoC	# LoCf	(re)Deploy
Baseline	24	2%	–	✓
PERSIST	–	–	4	✗

Table 4 Overview of the modified # lines of code and # lines of configuration to change the back-end storage architecture for both prototype implementations (scenario #2: changes are made in the configuration file for the prototype `Baseline`)

Prototype	# LoC	% LoC	# LoCf	(re)Deploy
Baseline	24	2%	20	✓
PERSIST	–	–	4	✗

a federated cloud setup for multi-tenant SaaS applications, the impact on the application performance is a vital criterion. As shown in the previous two sections, the decision to externalize data storage logic from the application to external data storage policies and configurations has its advantages in terms of flexibility and run-time customizability. This however, comes at the cost of additional performance overhead. More specifically, we expect, especially the inclusion

```
1  ...
2  <storage name="Cassandra-Private">
3    <datastore>Cassandra</datastore>
4    <nodes>private-ip-address</nodes>
5    <port>port-no</port>
6    <keyspace>LMaaS</keyspace>
7    <username>username</username>
8    <password>password</password>
9     ...
10   <properties>
11     <location>private</location>
12     <trusted>true</trusted>
13     <writethroughput>high</writethroughput>
14     <readthroughput>high</readthroughput>
15     <data>Archievedlog&Historicallog </data>
16      ...
17   </properties>
18 </storage>
19 <storage name="MongoDB-Private">
20   <datastore>MongoDB</datastore>
21   <nodes>private-ip-address</nodes>
22   <port>port-no</port>
23   <keyspace>LMaaS</keyspace>
24   <username>username</username>
25   <password>password</password>
26    ...
27   <properties>
28     <location>private</location>
29     <trusted>true</trusted>
30     <writethroughput>low</writethroughput>
31     <readthroughput>high</readthroughput>
32     <data>Rawlog & Metadata</data>
33      ...
34   </properties>
35 </storage>
36 ...
```

Listing 5 Illustration of specific changes made (in bold & underlined) to switch the back-end for *raw log* entries, *historical logs*, and *log meta-data* in the prototype `PERSIST`

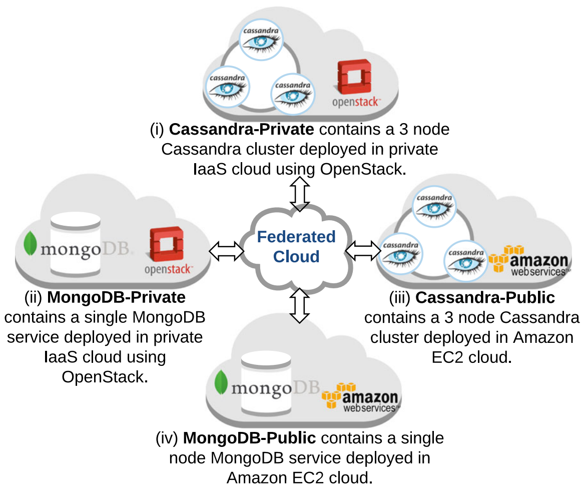

cassandra
openstack
(i) Cassandra-Private contains a 3 node Cassandra cluster deployed in private IaaS cloud using OpenStack.
mongoDB
(ii) MongoDB-Private contains a single MongoDB service deployed in private IaaS cloud using OpenStack.
Federated Cloud
amazon webservices
(iii) Cassandra-Public contains a 3 node Cassandra cluster deployed in Amazon EC2 cloud.
(iv) MongoDB-Public contains a single node MongoDB service deployed in Amazon EC2 cloud.

Fig. 4 An overview of various cloud providers (e.g., Database-as-a-Service providers) used to evaluate the performance overhead of the PERSIST middleware

of meta-data and the additional policy evaluation step to introduce a substantial increase in the performance overhead. Therefore, as mentioned in Section 4, we have implemented and integrated caching techniques in different components of the PERSIST middleware to address this issue (**C5** discussed in Section 2.4).

In this section, we present the results of a comparative performance benchmark of the multi-tenant LMaaS application, for create, read, update and delete (CRUD) transactions. We specifically focus on quantifying the extra overhead introduced by the PERSIST middleware over the Kundera platform (which is the baseline for comparison). We refer to our earlier study [35] for an indication of the performance overhead introduced by the Kundera platform itself.

In Section 5.3.1, we describe the overall application and experimental setup and also discuss the different deployment setups along with their hardware details in which we have executed the performance benchmarks. Then, Section 5.3.2 describes the workload characterization and outlines the design of conducted experiments, while the Section 5.3.3 subsequently presents the measurement methodology to characterize the performance overhead. Finally, Section 5.3.4 reports and discusses the performance overhead results.

5.3.1 Experimental Setup

We implemented two application setups doing the same CRUD transactions: one is based on the Kundera platform, which is the baseline for the performance comparison, while the other is based on the PERSIST middleware platform. The application-wide data storage policy (described in Listing 2) is considered to perform CRUD transactions for the application setup that uses the PERSIST middleware platform, whereas the same CRUD transactions logic is implemented in the source code for the application setup, which is based on the Kundera platform.

These application setups use a federated storage architecture as shown in Fig. 4 that combines the following cloud storage resources (i) the Cassandra-Private deployment setup contains a 3 node Apache Cassandra (version 2.1.18) cluster deployed and managed in a private IaaS cloud using OpenStack[5]; (ii) the MongoDB-Private deployment setup includes a single node MongoDB (version 3.4.9) service with all standard settings deployed and managed in a private IaaS cloud using OpenStack; (iii) the Cassandra-Public deployment setup comprises of a 3 node Apache Cassandra (version 2.1.18) cluster deployed in Amazon EC2[6] owned and managed by Instaclustr[7] (a Database-as-a-Service provider), and (iv) the MongoDB-Public deployment setup contains a single node MongoDB (version 3.4.9) service deployed in Amazon EC2 with all standard settings, which is owned and managed by mLab[8] (a Database-as-a-Service provider).

For the deployment setups of Apache Cassandra (i) and (iii) above, we defined a key space, with replication class set to `SimpleStrategy`. In addition, the key space `replication_factor` is set to 1 in order to address the high availability requirements of *raw log* entries and *log meta-data*, which are desired to be stored in a replicated manner (see lines 8, 18, 43 and 52 of Listing 2).

In our experiments, both the client (running an implementation of the multi-tenant LMaaS application) and the server (running data store instances) processes were running on separate machines. The client machine is equipped with Intel(R) Core(TM) i5 @ 2.60GHz (Dual) processor with 8 GB RAM and Windows 8 installed. In case of the private deployments ((i) and (ii) above, which are deployed in a

[5]https://www.openstack.org/
[6]https://aws.amazon.com/ec2/
[7]https://www.instaclustr.com/
[8]https://mlab.com/

private IaaS cloud using OpenStack), each node has Intel(R) 4 Core @ 2.60GHz processor, 8 GB RAM and is hosted on a compute node. The compute node consists of 40 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz processor with 120 GB RAM and runs the Linux/Ubuntu operating system. For the public cloud deployment setups ((iii) and (iv) above, which are deployed on Amazon EC2 and managed by different Database-as-a-Service providers), we utilized different instance types, so we specify their flavor for each deployment setup. In case of the public cloud deployment setup ((iii) above, which contains a 3 node Apache Cassandra cluster), each node was hosted on AWS EC2 t2.medium instance operating at eu-central-1 data center of the Amazon, equipped with 4 GB RAM and a dual-core vcpu @ 2.50GHz processor. Similarly, for the public deployment setup ((iv) above, which contains a single node MongoDB service), the service was hosted on multi-tenanted database server processes operating at eu-west-1 data center of the Amazon.

5.3.2 Workload Characterization

The benchmarks —to measure the performance in terms of execution time— were conducted on both application setups by executing the CRUD transactions under different workload conditions. We start our measurements with 100K log entries and gradually increase the workload upto 1000K (a million) log entries to determine how both setups scale when the data size increases. For example, we expect PERSIST introduces constant overhead regardless of the data scale. Therefore, the relative performance overhead of the PERSIST middleware should decrease progressively as a result of the increase in the workload (i.e. data scale).

We then conducted four experiments at different data scale which consider the storage requirements of *raw log* entries and *historical logs* of the multi-tenant LMaaS application. In the first experiment, both application setups use Cassandra-Private deployment setup for addressing the requirements of confidential *raw log* entries. In the second experiment, MongoDB-Private deployment setup is used to address the requirements of confidential *historical logs*. Then, in the third experiment, both application setups are configured to use Cassandra-Public for addressing the requirements of non-confidential *raw log* entries. Finally, in the fourth and the last experiment, MongoDB-Public deployment setup is used to address the requirements of non-confidential *historical logs*. As in all four experiments, confidential data is always stored in private deployment setups, data encryption was not required. Hence, the execution time doesn't take into account the time required to encrypt confidential data for both application setups.

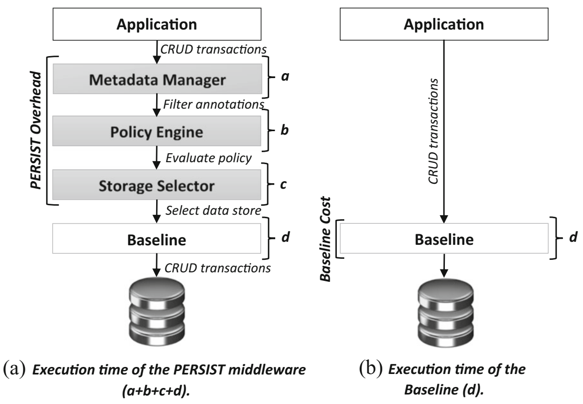


Fig. 5 Execution time of **a** the PERSIST middleware and **b** the baseline

5.3.3 Measurement Methodology

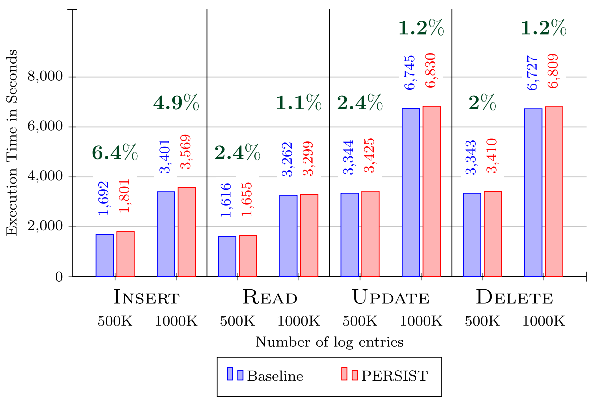
In order to evaluate the performance overhead of the PERSIST middleware, we first measure $t_{PERSIST}$, the total execution time PERSIST takes to perform CRUD transactions, which is the sum of time spent by different components of the PERSIST middleware (a+b+c+d) depicted on the left hand side (a) of the Fig. 5. Then, we measure $t_{baseline}$, the total execution time baseline takes to perform CRUD transactions (d) as shown on the right hand side (b) of the Fig. 5. Finally, by subtracting both measurements, the execution time of the baseline from the execution time of the PERSIST middleware, we can characterize the performance overhead introduced by the PERSIST middleware as $t_{overhead} = t_{PERSIST} - t_{baseline}$.

In order to avoid skewing the effect of any noisy measurements, we have repeated each experiment 3 times and present the averaged results. After completing all CRUD transactions for each run, we emptied the entire data store. Beyond these experiments, we also run some additional performance benchmarks where we have considered a combination of different deployment setups and other data types of the multi-tenant LMaaS application (e.g., *log meta-data*). However, these additional benchmarks do not demonstrate any significant impact on the performance overhead of the PERSIST middleware and thus we only focus on the above measurements.

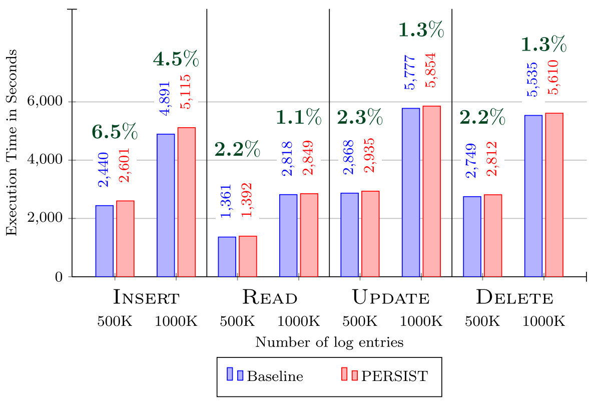
5.3.4 Performance Results

In this section, we report the results of our performance benchmarks. More specifically, the results of all four performance experiments for both application setups conducted at different deployment setup under various data scale are presented in Fig. 6.

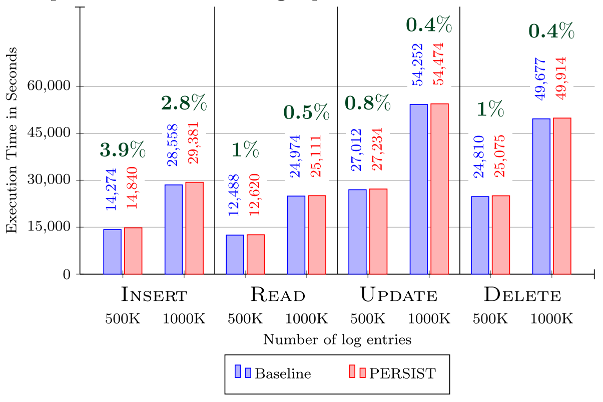
Figure 6a depicts the results of the first experiment, which measures the execution time in seconds to perform CRUD transactions for the Cassandra data store deployed in a private IaaS cloud. As shown for the 500K data scale, the PERSIST middleware platform ($t_{PERSIST}$) takes 1,801 seconds for the insert transaction, 1,655 seconds for the read transaction, 3,425 seconds for the update transaction, and 3,410 seconds for the delete transaction compared to the baseline ($t_{baseline}$), which takes 1,692 seconds, 1,616 seconds, 3,344 seconds, and 3,343 seconds for insert, read, update, and delete transactions respectively. Subsequently, the average relative performance overhead introduced by the PERSIST middleware ($t_{overhead}$) for



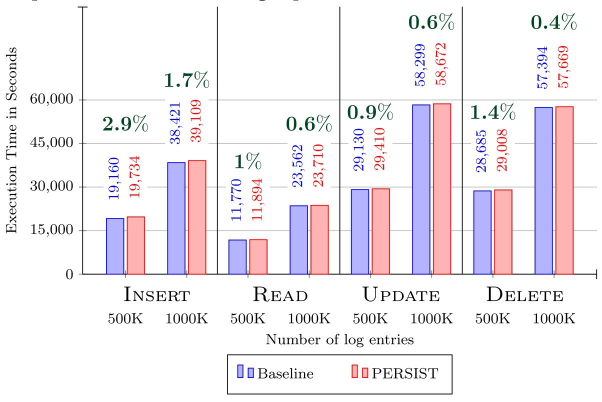
(a) Execution time in seconds to perform CRUD transactions for confidential *raw log* entries, which are stored in the Cassandra-Private cloud storage. The setup consists of a 3 node Cassandra cluster deployed and managed in a private IaaS cloud using OpenStack.



(b) Execution time in seconds to perform CRUD transactions for confidential *historical logs*, which are stored in the MongoDB-Private cloud storage. The setup consists of 1 node MongoDB service deployed and managed in a private IaaS cloud using OpenStack.



(c) Execution time in seconds to perform CRUD transactions for non-confidential *raw log* entries, which are stored in the Cassandra-Public cloud storage. The setup consists of a 3 node Cassandra cluster deployed in Amazon EC2 public cloud.



(d) Execution time in seconds to perform CRUD transactions for non-confidential *historical logs*, which are stored in the MongoDB-Public cloud storage. The setup consists of a 1 node MongoDB service deployed in Amazon EC2 public cloud.

Fig. 6 The relative performance overhead (presented above the bars) of the PERSIST middleware for 500K and 1000K data scale on four different deployment setups

500K data scale is 6.4% for the insert transaction, 2.4% for read and update transactions, and 2% for the delete transaction. Similarly, for the 1000K data scale, the execution time of the PERSIST middleware ($t_{PERSIST}$) is 3,569 seconds for the insert transaction, 3,299 seconds for the read transaction, 6,830 seconds for the update transaction, and 6,809 seconds for the delete transaction, whereas the baseline ($t_{baseline}$) takes 3,401 seconds, 3,262 seconds, 6,745 seconds, and 6,727 seconds for insert, read, update, and delete transactions respectively. The relative performance overhead of the PERSIST middleware ($t_{overhead}$) for 1000K data scale is 4.9% for the insert transaction, 1.1% for the read transaction, and 1.2% for update and delete transactions. In the rest of this section, we will omit the execution time in seconds (i.e. $t_{PERSIST}$ and $t_{baseline}$) and only present and discuss the relative performance overhead of the PERSIST middleware ($t_{overhead}$).

The results of the second experiment where the MongoDB data store is deployed in a private IaaS cloud and CRUD transactions were performed on different data scales are presented in Fig. 6b. As we can see in Fig. 6b (on top of bars), the relative performance overhead introduced by the PERSIST middleware ($t_{overhead}$) for 500K data scale is 6.5% for the insert transaction, 2.2% for the read transaction, 2.3% for the update transaction, and 2.2% for the delete transaction. The relative performance overhead of the PERSIST middleware ($t_{overhead}$) when the data size increases (e.g., for 1000K data scale) is 4.5% for the insert transaction, 1.1% for the read transaction, and 1.3% for update and delete transactions.

Figure 6c and present the execution time and the relative performance overhead for the third and the fourth experiment where Cassandra and MongoDB data stores are deployed in Amazon EC2 public cloud. For the third experiment where the data scale is 500K, the relative performance overhead of PERSIST ($t_{overhead}$) is 3.9% for the insert transaction, 1% for read and delete transactions, and 0.8% for the update transaction. For the 1000K data scale, the relative performance overhead of PERSIST ($t_{overhead}$) decreases to 2.8% for the insert transaction, 0.5% for the read transactions, and 0.4% for update and delete transactions.

Similarly, in the case of the fourth experiment, for the 500K data scale, PERSIST introduces the relative performance overhead ($t_{overhead}$) of 2.9% for the insert transaction, 1% for the read transaction, 0.9% for the update transaction, and 1.4% for the delete transaction. In case where the data size increases to 1000K data scale, the relative performance overhead of PERSIST ($t_{overhead}$) decreases to 1.7% for the insert transaction, 0.6% for the read and update transactions, and finally 0.4% for the delete transaction.

As we can see for all four experiments in Fig. 6a, b, c, and d as well respectively, the relative performance overhead of the PERSIST middleware ($t_{overhead}$) decreases substantially when the data size increases. For example, for the 500K data scale in the first experiment, PERSIST introduces 6.4% relative overhead for the insert transaction, whereas in the same experiment for the 1000K data scale, the relative overhead for the same operation decreases to 4.9%. Likewise, the relative performance overhead of PERSIST decreases to 4.5% from 6.5% for the insert transaction when the data size increases from 500K data scale to 1000K data scale.

This reduction is mainly caused by the baseline. For example, when the data size increases, the baseline (d) in Fig. 5 takes more time to perform CRUD transactions. Consequently, as we expected due to the constant overhead of the PERSIST middleware (a+b+c) depicted on the left hand side (a) of the Fig. 5, the relative performance overhead decreases substantially with the increase in data size. This correlation is very obvious in the third and the fourth experiment as shown in Fig. 6c and d respectively, where the Amazon EC2 public cloud deployment setup is used. In the case of public cloud, there is high latency between the client process (running an implementation of the multi-tenant LMaaS application deployed in a private IaaS cloud) and the server process (running data store instances deployed on Amazon EC2 public cloud). Hence, the baseline requires a substantially longer time to perform CRUD transactions (see Fig. 6c and d) as compared to the private IaaS cloud (see Fig. 6a and b).

We have also noticed that the relative performance overhead introduced by the PERSIST middleware ($t_{overhead}$) for the insert transaction is higher than read, update, and delete transactions. This is mainly caused by extra steps involved before executing the insert transaction that introduces an additional performance overhead (cf. Fig. 3 to see the flow of requests to perform the insert transaction). For example, PERSIST requires reading annotations to read the object's metadata; executing data storage policies for efficient data placement decisions; selecting the storage systems that satisfy all the desired requirements; and indexing data storage locations to further optimize the performance for read, update, and delete transactions. In addition, a cache hit[9] occurs almost all the time for the read, update, and delete transactions, whereas, there is atleast one cache miss for the write transaction, which requires validating the cache. Therefore, the performance overhead is higher for the insert transaction, which is maximum 6.4%, but minimal, which is 2.4% for read, update, and delete transactions.

5.4 Discussion on Evaluation

This section discusses the overall choices we have made in the middleware for the evaluation.

Firstly, we have examined the requirements of *raw log* entries, *historical logs*, and *log meta-data* where simple data storage and confidentiality requirements (e.g., encrypt the whole entity object) are considered. In addition, we have not taken the contrasting requirements of different tenants into consideration. We expect that addressing more complex data storage requirements (e.g., consider other data types of the multi-tenant LMaaS application) and the confidentiality requirements (e.g., support data encryption at various levels of granularity) as well as considering the contrasting requirements of multiple tenants —where each tenant has slightly different storage- and privacy-related requirements— will not impact the PERSIST middleware and no changes will be required in the application source code. In PERSIST, hard-coding specifics (i.e. storage logic) are externalized from the application layer to external data storage policies. These policies can be specified at run time by each tenant of the SaaS application to customize the PERSIST middleware according to its own specific requirements. On the other hand, the amount of implemented lines of code in the baseline implementation highly depends on the application requirements. As an example, addressing more requirements of the multi-tenant SaaS application or considering contrasting requirements of different tenants, we expect a significant implementation effort in terms of lines of code in the application will be required. More specifically, addressing different requirements for multiple tenants will require the storage logic to be implemented for each tenant of the SaaS application, which significantly influences the implementation effort and increases the overall lines of code to be added.

Secondly, to determine the cost/effort to change the back-end storage architecture of a federated cloud setup (evaluation #2), we have considered two data stores (Apache Cassandra and MongoDB), both deployed in a private IaaS cloud using OpenStack as well as Amazon EC2 public cloud. These data stores were used to manage the requirements of *raw log* entries, *historical logs*, and *log meta-data*. At the time we assess the cost/effort to change the storage architecture, these data stores were not used by other data types of the multi-tenant LMaaS application. Therefore, similar to the evaluation #1, again in the evaluation #2, we have only considered data types with simple data storage requirements where each data type is exactly stored in one back-end storage system. For instance, confidential *raw log* entries are stored in the Cassandra data store operating at a private IaaS cloud (i.e. Cassandra-Private), while non-confidential *raw log* entries are stored in Cassandra, missing in, operating at Amazon EC2 public cloud and managed by Instaclustr (a Database-as-a-Service provider for Cassandra). In addition, we have not considered the requirements of multiple tenants, where each tenant may impose slightly different requirements for a particular data type. For example, *raw log* entries might be confidential for only some tenants, but not for other tenants. Consequently, this may lead to a single data type being partitioned across multiple cloud storage providers in a federated cloud setup. In a more realistic scenario, application data is partitioned across multiple data stores and a single data store is usually responsible to manage different types of application data. To address such a scenario, additional changes will be required in the application source code for the baseline implementation.

On the other hand, the storage logic in the PERSIST middleware is externalized from the application source code to external data storage policies. In addition, there is an extra step of indirection between the data storage policies and the configuration model (i.e. configuration file). This extra step of indirection helps changing the configuration model without making any changes in the data storage policy file. Hence, considering more data types into account for the prototype

[9]A cache hit occurs when the requested data can be found in the cache.

PERSIST implementation only requires to modify a single line in the configuration model for each data type to change the back-end storage architecture. Furthermore, no additional changes are anticipated in the application when the requirements of multiple tenants are considered as these requirements can be addressed by specifying tenant-specific configuration and policy files.

Thirdly and finally, we have taken performance into consideration while designing the PERSIST middleware and therefore, implemented cache techniques in different components to improve the overall performance. For illustration purposes, we choose a simple example to demonstrate the improvement in performance when the cache techniques are implemented in different components (in **bold** and *italic*) of PERSIST (see Table 5). To accomplish this, we have used the static properties of an entity class (i.e. metadata), which remains unchanged at run time. Therefore, a cache miss occurs only for the first request, which causes the PERSIST middleware to filter the annotations and validate the cache. Afterwards, the cache hits occur for all the remaining requests. Similarly, the policy decisions are cached after evaluating and executing the policy for the first time only. For the latter requests, the policy decisions are fetched from the cache only. We expect both these techniques have a great impact on the overall performance improvement of the PERSIST middleware.

5.4.1 Threats to Validity

This section presents the threats to validity that can compromise the results of these evaluations.

Internal Validity The most serious threat to the validity of our findings is related to our conclusion on the performance overhead (evaluation #3) of the PERSIST middleware. To evaluate the performance overhead, we selected the best case: the meta-data properties of an entity object are not changed by tenants at run time and are static. However, in situations where the dynamic meta-data properties are enforced by tenants (i.e. the meta-data properties of an entity object are changed), the performance overhead of the PERSIST for the insert transaction may become slightly higher.

Table 5 Execution time of the PERSIST middleware components with and without cache

Component	Disabled	Enabled
PERSIST		
↳ ***Metadata Manager***	***72.48***	***2.31***
↳ Storage Selector	2	3.16
↳ ***Policy Engine***	***397.39***	***1.84***
↳ Baseline (Client API Drivers)	199.3	199.3
Total time in seconds	671.17	206.61

External Validity The major threat to the external validity is the fact that we have evaluated the PERSIST middleware for only one application case, the multi-tenant Log Management-as-a-Service (LMaaS) offering. Although indicative of the complexity of a typical SaaS application, we have only focused on the requirements of *raw log* entries, *historical logs*, and *log meta-data*. In addition, our evaluation efforts focus on a limited set of requirements (with respect to storage and privacy) and in our prototype implementation, we have not considered the full complexity of the multi-tenant SaaS application (simple requirements of only a single tenant were examined). We expect that the evaluation results —more specifically, for the evaluation #1 and the evaluation #2— might differ when taking into account the full complexity of the multi-tenant SaaS application.

6 Related Work

There has been growing interest in addressing the problem of heterogeneity and the lack of standardization that exists in different cloud providers. Consequently, a growing body of research, both from the industry [23, 27, 44] and the research community [2, 5, 18, 22, 38, 42] focuses on application portability and addresses the problem of heterogeneity by providing a uniform API across multiple NoSQL data stores. Similarly, some research works [3, 7, 14, 50] have proposed multi-cloud storage systems that use multiple cloud providers to either obtain better application performance, optimize data storage cost, enhance availability, and/or ensure data security. However, none of these multi-cloud storage systems supports data storage policies to facilitate data management across multiple clouds and offer multi-tenant customization support.

The current state of practice in data access middleware platforms or object NoSQL database mappers —systems like Kundera [27], Hibernate OGM [23], etc— only provide abstraction mechanisms to hide the complexity of different data models and APIs. They do not sufficiently: (i) support run-time cross-database and cross-provider polyglot persistence on a per-object level as it consistently requires creating multiple entities and writing storage logic in the application source code for new data storage requirements, (ii) support the development and run-time customization of multi-tenant SaaS applications, and (iii) provide a built-in support for security and privacy-related requirements of individual tenants. Existing academic systems, such as RACS [1], DepSky [8], HAIL [10], Scalia [32] combine multiple cloud providers to achieve high availability and address vendor lock-in problem.

Our work is similar to Tiera [39] in the sense that the authors proposed a middleware that also uses a policy-driven approach for making data storage decisions. Tiera utilizes multiple storage tiers (e.g., SSDs, local storage, etc.) for getting composite benefits. However, the implementation of Tiera only focuses on a single data center and as acknowledged by the authors, does not offer disaster recovery and thus also high availability. The authors considered spanning multiple data centers as an open issue and acknowledge that in many cases applications require multiple data centers to achieve better disaster recovery. In contrast, PERSIST achieves better disaster recovery and high availability in the sense that it facilitates cross-provider data replication and allows tenants to replicate data across multiple data centers in a federated cloud setup.

In another related work, the authors proposed Scalia [32], a cloud brokerage solution that makes data placement decisions based on data access patterns subject to storage cost optimization. Our research is similar to Scalia in aspects such as data placement strategy and the use of multiple cloud storage providers to achieve better availability. However, Scalia is a single-purpose solution, only focusing on the cost optimization. In contrast, PERSIST is a multi-purpose solution as it facilitates multi-tenant SaaS applications and thus also takes into account different requirements of tenants —in terms of performance, availability, scalability, etc— for different data elements.

The research conducted by Bermbach et al. [7] focuses on using a combination of cloud storage providers to manage consistency-latency trade-offs. The authors proposed MetaStorage, a federated architecture, which extends the Appscale platform with a unified access to diverse cloud storage services. Similar to the PERSIST middleware, MetaStorage also achieves high availability by replicating data across multiple cloud storage services, which further avoids vendor lock-in. However, in our research: (i) we use a policy-based approach for data storage decisions, (ii) we alleviate the complexity (i.e. hard-coding specifics) of a federated cloud storage architecture from the application layer to external storage policies and configurations, and (iii) we provide a support for fine-grained tenant customization, while they do not overcome these factors in their research.

In previous work [34], we have presented an initial architecture of the middleware that uses policies for data storage decisions. The architecture —which contains the detailed description and the implementation as well as the underlying concepts— of the PERSIST middleware presented in this paper as such extends that initial architecture by adding support for CRUD transactions, encryption-enabled CRUD transactions, and search operations across federated clouds. Furthermore, the data placement configurability over a federated cloud storage architecture is also supported. In addition, we implemented (i) run-time cross-database and cross-provider polyglot persistence support on a per-object level that facilitates specific data elements to be replicated across multiple cloud storage technologies or providers (e.g., Database-as-a-Service providers), and (ii) data encryption support to ensure confidentiality of sensitive data stored in external untrusted cloud providers using the data mapping strategy motivated in [37] that can be enacted at different levels of granularity (i.e. from entire data objects up to the level of individual properties of an entity). Finally, we significantly extended our evaluation, focusing on three different dimensions.

7 Discussion

We have validated and demonstrated the PERSIST middleware in the context of the Log Management system which is a high-throughput application. As explained in Sections 3.1 and 3.6; the supported meta-data, the annotations, and data storage policies are not hard-coded nor rigid, and PERSIST is not limited to only high-throughput computing applications. The platform can easily be extended to address the requirements of different applications and to other types of applications (e.g., high-performance applications). However, in a realistic deployment enviroment, the performance of these data stores will fluctuate in line with the workload. Therefore, the specification of meta-data statically may not be the most optimal solution and in some cases might even lead to inefficient data storage decisions. Integration of techniques related to dynamic data storage decision support and self-adaptiveness is part of our ongoing work [36].

Although we have primarily demonstrated and evaluated the PERSIST framework in terms of the CRUD operations, the presented middleware is also highly suited to address more challenging federated cloud data management issues. We discuss these below:

- *Data consistency* As discussed in Section 3.1, PERSIST provides support for tenants and SaaS providers to replicate data across multiple cloud providers. To accomplish this, different replication strategies and policies can be employed that are specific to the context of the federated cloud. For example, a *cross-provider replication* strategy involves replicating data across multiple cloud providers, whereas a *cross-technology replication* strategy focuses on replicating data across multiple database technologies to benefit from complementarity.

The selection of a multi-cloud data replication policy in general involves a key trade-off between data consistency (strong consistency vs. weak consistency) and performance. For example, in order to achieve strong data consistency, cross-provider data replication (to support high availability and disaster recovery) comes at the cost of increased performance overhead. This is mainly because write, update, and delete operations are considered as executed successfully only when the operations are performed across all and the successful response is received from all cloud providers. This, however, comes at an additional performance cost, but at the same time ensures that reading data from any cloud provider is always consistent as write, update, and delete operations always leave data consistent across cloud providers. Similarly, to provide cross-provider data replication support with optimal performance, write, update, and delete operations can be executed with weak consistency (e.g., an operation is considered to be successful when a response is received from at least one cloud provider), which may leave data inconsistent across cloud providers. This inconsistency, then can be resolved when the data is being read with the latching of timestamp.

In the context of the multi-tenant Log Management-as-a-Service case, strong data consistency is preferred for cross-provider data replication at the cost of additional performance. A simplified example of cross-provider data replication support in policies has been illustrated in Listing 1 (line 11). In essence, the core principles of PERSIST are to declaratively describe the data object properties (as annotations) and data store properties (e.g., location, provider, technology, etc.) and leverage these to create self-sufficient policies that rely exclusively on these properties (e.g., selecting multiple data stores that are hosted by different providers for data replication). The same principles easily hold for implementing different replication policies and tailoring these to the specific application requirements.

- *Data migration* As described in Section 3.6, PERSIST supports dynamic (re)configurability and thus allows tenants to change or (re)configure the underlying federated cloud storage architecture over time to accommodate for changing requirements. Dealing with this complexity again involves making an appropriate trade-off between performance, complexity, consistency, availability, etc. For example, the data can automatically be migrated when the tenant changes the storage configuration. This strategy has a negative impact on (i) performance (due to the migration of large data volumes), (ii) availability (due to the database downtime) and (iii) consistency (during the online data migration). In PERSIST, data migration is not explicitly supported, but we envision the integration of a lambda-style architecture comprising of a batch layer and a speed layer to accomplish this [47]. Another feasible strategy is to keep using the previous configuration for older data and using the updated version for new data objects. In such a scenario, requests for new data can be handled according to the updated storage configuration, while requests for old data can be handled according to the previous version of the configuration. However, such a solution comes at the cost of increased complexity of versioning as more than one cloud providers need to be managed.

In the multi-tenant Log Management-as-a-Service application case, we have implemented the latter strategy due to the availability and consistency requirements of this application and the introduced complexity of combining multiple cloud providers is managed by the PERSIST middleware.

– *Federated data search* As explained in Section 3.4.1, PERSIST distributes data of a single tenant across multiple cloud storage providers. In order to search tenant-specific data back efficiently without incurring additional communication costs, PERSIST maintains a cross-provider data index (the `Index Manager` component as depicted in Fig. 2). PERSIST implements a federated search protocol [21], i.e. search requests are transmitted to only those storage nodes that hold the relevant data, results are then aggregated and presented to the application. Again, to accomplish federated cloud data search, different search strategies are feasible and supported by the PERSIST middleware. A simplified example of activating search in policies with a preference on forwarding search requests to private storage resources was illustrated in Listing 1 (line 12). The other feasible search policies are (i) searching with a preference on external cloud storage provider's infrastructure (e.g., to limit the load on on-premise storage infrastructure or to search specific types of application data, i.e. non-confidential data) and (ii) broadcasting search queries over all involved nodes (e.g., to search different types of application data). In case of replication, the data (de)duplication is handled by the PERSIST middleware.

8 Conclusion and Future Directions

Cross-provider, hybrid/multi-cloud, and federated cloud storage architectures are increasingly compelling for service providers. However, in practice, managing such an architecture in the application is non-trivial and introduces substantial additional complexity. More specifically, in the context of multi-tenant SaaS applications —in which customer organizations (tenants) expect to make their customizations to the SaaS application and the federated storage architecture at run time up to the level of individual service request— this is rather desirable, but highly complex and challenging.

This paper has presented a policy-based middleware called PERSIST that (i) provides an abstraction to hide the complexity of the underneath storage architecture and exposes a uniform API to manage the application data across a federated cloud storage setup; (ii) offers a fine-grained tenant customization support to accommodate continuously changing requirements of tenants for specific data elements, as such (a) facilitates run-time cross-database and cross-provider polyglot persistence on a per-object level, and (b) ensures confidentiality of sensitive data at differing levels of granularity through data encryption; and (iii) externalizes the hard-coding specifics (i.e. complex storage logic) of a federated cloud storage architecture from the SaaS application layer, which helps to make the multi-tenant SaaS application more flexible, manageable, and agile toward the initial design and the future enhancements.

This work fits into our ongoing research on federated data storage architectures and multi-cloud portability of SaaS applications. Future enhancements of the middleware include a broader exploration and implementation of reusable data storage tactics that have become possible in such federated cloud storage setups, such as cross-provider replication, or data pseudonymization/de-identification. We also plan to further investigate to what extent our support of dynamic properties (i.e. metadata that changes at run time) can be enhanced and at what performance cost. Another important part of the future work is to extend such a static policy-driven setup —which is based on the static properties and meta-data of the operational environment (i.e. properties of different cloud providers) and may lead to sub-optimal data placement decisions across a federated cloud storage setup— with support for policies that are based on the dynamic conditions of a federated cloud architecture (i.e. continuously changing conditions of different cloud providers such as performance, availability). Finally, the scalability test of the PERSIST middleware for dynamic cloud storage resources is also considered for the future work.

Acknowledgements We would like to thank Bert Lagaisse and Vincent Reniers for their helpful comments and constructive feedback. This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS), the SBO DeCoMAdS project, and the imec SeClosed project.

References

1. Abu-Libdeh, H., Princehouse, L., Weatherspoon, H.: RACS: a case for cloud storage diversity. In: SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing. ACM (2010)
2. Alomari, E., Barnawi, A., Sakr, S.: CDPort: A framework of data portability in cloud platforms. In: iiWAS '14 Proceedings of the 16th International Conference on Information Integration and Web-based Applications &Services, pp. 126–133. ACM (2014)
3. Alzain, M.A., Soh, B., Pardede, E.: MCDB: Using multi-clouds to ensure security in cloud computing. In: Ninth International Conference on Dependable, Autonomic and Secure Computing, pp. 784–791 (2011)
4. Atzeni, P., Bugiotti, F., Rossi, L.: Sos (save our systems): A uniform programming interface for non-relational systems. In: Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, pp. 582–585. ACM (2012)
5. Atzeni, P., Bugiotti, F., Rossi, L.: Uniform access to non-relational database systems: the sos platform. In: CAiSE '12 Proceedings of the 24th international conference on Advanced Information Systems Engineering, pp. 160–174. ACM (2012)
6. Bȧżȧr, C. et al.: The transition from RDBMS to NoSQL. a comparative analysis of three popular non-relational solutions: Cassandra, mongodb and couchbase. Database Syst. J. **5**(2), 49–59 (2014)
7. Bermbach, D., Klems, M., Tai, S., Michael, M.: Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In: IEEE International Conference on Cloud Computing (CLOUD), 2011, pp. 452–459. IEEE (2011)
8. Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P.: DepSky: Dependable and secure storage in a cloud-of-clouds. Trans. Storage **9**(4), 12:1–12:33 (2013)
9. Blanke, T. et al.: Back to our data – experiments with NoSQL technologies in the humanities. In: IEEE International Conference on Big Data, pp. 17–20 (2013)
10. Bowers, K.D., 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)
11. Brewer, E.: Cap twelve years later: How the "rules" have changed. Computer **45**(2), 23–29 (2012)
12. Chohan, N., Bunch, C., Krintz, C., Canumalla, N.: Cloud platform datastore support. J. Grid Comput. **11**(1), 63–81 (2013)
13. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154 (2010)
14. Dobre, D., Viotti, P., Vukolic, M.: Hybris: Robust hybrid cloud storage. In: SOCC '14 Proceedings of the ACM Symposium on Cloud Computing, pp. 1–14. ACM (2014)
15. Ehcache. JAVA'S MOST WIDELY-USED CACHE. http://www.ehcache.org/. [Last visited on June 20, 2017]
16. Ferdman, M. et al.: Clearing the clouds: A study of emerging scale-out workloads on modern hardware. SIGPLAN Not. **47**(4), 37–48 (2012)
17. Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud computing and grid computing 360-degree compared. In: Grid Computing Environments Workshop, pp. 1–10 (2008)
18. Gessert, F., Bücklers, F., Orestes, N.R.: A scalable database-as-a-service architecture for low latency. In: IEEE 30th International Conference on Data Engineering Workshops (ICDEW), pp. 215–222 (2014)
19. Grolinger, K., Higashino, W.A., Tiwari, A., Capretz, M.A.M.: Data management in cloud environments NoSQL and newsql data stores. J. Cloud Comput. Adv. Syst. Appl. **2**(1), 1–24 (2013)
20. Grozev, N., Buyya, R.: Multi-cloud provisioning and load distribution for three-tier applications. ACM Trans. Auton. Adapt. Syst. **9**(3), 13:1–13:21 (2014)
21. Gupta, A.M., Gadepally, V., Stonebraker, M.: Cross-engine query execution in federated database systems. In: High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2016)
22. Haselmann, T., Thies, G., Vossen, G.: Looking into a rest-based universal api for database-as-a-service systems. In: IEEE 12th Conference on Commerce and Enterprise Computing (CEC), pp. 17–24 (2010)
23. Hibernate. Hibernate OGM - The power and simplicity of JPA for NoSQL datastores. http://hibernate.org/ogm/. [Last visited on June 20, 2017]
24. imec. D-BASE: Optimization of Business Process Outsourcing Services. https://distrinet.cs.kuleuven.be/research/projects/D-BASE. [Last visited on October 02, 2017]
25. imec. DMS2: Decentralized Data Management and Migration of SaaS. https://distrinet.cs.kuleuven.be/research/projects/(DMS)2 [Last visited on October 02, 2017]
26. imec. Sequoia: Middleware for scalable, attribute-based querying of multitenant, cloud-based databases. https://www.imec-int.com/nl/imec-icon/research-portfolio/sequoia [Last visited on October 02, 2017]
27. Impetus. A JPA 2.1 compliant Polyglot Object-Datastore Mapping Library for NoSQL Datastores. https://github.com/impetus-opensource/Kundera/. [Last visited on June 20, 2017]
28. Konstantinou, I., Angelou, E., Boumpouka, C., Tsoumakos, D., Koziris, N.: On the elasticity of NoSQL databases over cloud management platforms. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM '11, pp. 2385–2388. ACM, New York (2011)
29. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. **12**(4), 559–592 (2014)
30. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. [Last visited on Febuary 18, 2016]
31. Oracle. EntityManager (Java(TM) EE 7 Specification APIs. javax/persistence/EntityManager.htmljavax/persistence/EntityManager.html. [Last visited on June 22, 2017]
32. Papaioannou, T.G., Bonvin, N., Aberer, K.: Scalia: an adaptive scheme for efficient multi-cloud storage. In: SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. ACM (2012)
33. Rafique, A., Van Landuyt, D., Reniers, V., Joosen, W.: Leveraging NoSQL for scalable and dynamic data

encryption in multi-tenant saas. In: 2017 IEEE Trustcom/BigDataSE/ICESS, pp. 885–892 (2017)
34. Rafique, A., Van Landuyt, D., Lagaisse, B., Joosen, W.: Policy-driven data management middleware for multi-cloud storage in multi-tenant saas. In: IEEE/ACM 2nd International Symposium on Big Data Computing (BDC), pp .78–84 (2015)
35. Rafique, A., Van Landuyt, D., Lagaisse, B., Joosen, W.: On the performance impact of data access middleware for NoSQL data stores. IEEE Trans. Cloud Comput. (TCC) **PP**(99), 1–1 (2016)
36. 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, pp. 4:1–4:6 (2017)
37. Rafique, A., Van Landuyt, D., Reniers, V., Joosen, W.: Towards scalable and dynamic data encryption for multi-tenant saas. In: Proceedings of the Symposium on Applied Computing, SAC '17, pp. 411–416. ACM, New York (2017)
38. Rafique, A., Walraven, S., et al.: Towards portability and interoperability support in middleware for hybrid clouds. In: CrossCloud 2014: IEEE INFOCOM CrossCloud Workshop. IEEE (2014)
39. Raghavan, A., Chandra, A., Weissman, J.: Tiera: towards flexible multi-tiered cloud storage instances. In: Middleware '14 15th International Middleware Conference, pp. 1–12. ACM (2014)
40. Redhat. Drools. https://www.drools.org/. [Last visited on June 20, 2017]
41. IWT SBO. DeCoMaDs: Deployment and Configuration Middleware for Adaptive Software-as-a-Service. https://distrinet.cs.kuleuven.be/research/projects/DeCoMAdS [Last visited on October 02, 2017]
42. Sellami, R., Bhiri, S., Defude, B.: Odbapi: A unified rest api for relational and NoSQL data stores. In: 2014 IEEE International Congress on Big Data (BigData Congress), pp. 653–660 (2014)
43. Sharp, J., McMurtry, D., Oakley, A., Subramanian, M., Zhang, H.: Data Access for Highly-Scalable Solutions Using SQL, NoSQL, and Polyglot Persistence, 1st edn. Microsoft Patterns & Practices (2013)
44. Spring. Spring Data. http://projects.spring.io/spring-data/, 2015. [Last visited on June 20, 2017]
45. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era:(it's time for a complete rewrite). In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 1150–1160 (2007)
46. Storl, U., Hauf, T., Klettke, M., Scherzinger, S.: Schemaless NoSQL data stores âĂŞ object-NoSQL mappers to the rescue? In: 16th Conference on "Database Systems for Business, Technology, and Web" (BTW), pp. 579–600 (2015)
47. Vanhove, T., Van Seghbroeck, G., Wauters, T., De Turck, F.: Live datastore transformation for optimizing big data applications in cloud environments. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pp. 1–8 (2015)
48. Verginadis, Y., Michalas, A., Gouvas, P., Schiefer, G., Hübsch, G., Paraskakis, I.: Paasword: A holistic data privacy and security by design framework for cloud services. J. Grid Comput. **15**(2), 219–234 (2017)
49. Walraven, S., Truyen, E., Joosen, W.: A middleware layer for flexible and cost-efficient multi-tenant applications. In: Middleware '11: Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware, pp. 370–389 (2011)
50. Yang, K., Jia, X.: An efficient and secure dynamic auditing protocol for data storage in cloud computing. IEEE Trans. Parallel Distrib. Syst. **24**(9), 1717–1726 (2013)