Policy-Driven Data Management Middleware for Multi-Cloud Storage in Multi-Tenant SaaS

Ansar Rafique, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
E-mail: firstname.lastname@cs.kuleuven.be

***Abstract*—Multi-tenant Software-as-a-Service (SaaS) applications are increasingly built on combinations of cloud storage technologies and providers in a so-called multi-cloud setup. One advantage is that such a setup helps satisfying the different —sometimes even contrasting— storage requirements of different customer organizations (tenants). In such a multi-cloud environment, the application data is distributed and replicated over multiple cloud storage systems, each differing profoundly in supported data models, development APIs, performance, scalability, availability, and durability.**

Despite the clear benefits, managing such a multi-cloud storage architecture in practice is non-trivial. Addressing this complexity in the application layer is far from ideal, as it inherently limits the flexibility with which continuously changing application-wide and tenant-specific storage requirements can be met.

To alleviate this, we present a reusable data management middleware that (i) makes abstraction of multiple cloud storage technologies and thus also providers; (ii) follows a policy-driven approach for making data placement decisions; and (iii) provides tenant customization support, i.e. by allowing tenants to define storage configurations and data storage policies.

We validate and evaluate our prototype implementation in the context of a realistic multi-cloud SaaS application. Our performance benchmark results indicate that the benefits of the proposed middleware can be achieved with acceptable overhead.

***Index Terms*—Data management middleware, Multi-cloud storage, Policy-driven, Abstraction API for NoSQL, Multi-tenant SaaS**

I. Introduction

The cloud computing paradigm promises high availability, elastic scalability, and thus offers increased flexibility. Due to these characteristics of cloud computing, many organizations adapt their IT infrastructure to operate completely or partially in the cloud [4], [12]. Cloud storage providers offer online mass storage services, which support data management facilities for multiple applications in an efficient and scalable manner. In this context, *Not only SQL* (NoSQL) technology has come to the forefront and has become increasingly popular as cloud data management systems [12], currently owning the highest share of cloud storage offerings [3], [19].

In practice, selecting a single provider or a single storage technology is often too restrictive for three main reasons. Firstly, there is a wide variety and heterogeneity of storage systems in the NoSQL arena, spanning over multiple storage providers [11], each of them differing profoundly in terms of the data model, development application programming interface (API), performance, scalability, availability, and durability [11], [13], [19]. Each NoSQL system makes a slightly different trade-off in terms of the qualities listed above. Secondly, multi-tenant Software-as-a-Service (SaaS) applications serve multiple customer organizations (tenants) simultaneously, and therefore have to satisfy the different and sometimes contrasting storage requirements of these tenants. Thirdly, relying on a single cloud storage provider comes with the non-negligible risks of technology, provider or vendor lock-in, introducing concerns about provider reliability, availability, scalability, and performance guarantees.

For these reasons, cloud providers are actively searching for solutions that enable them to leverage the benefits of using a combination of different cloud storage technologies and providers, in so-called multi-cloud setups. However, configuring and operating a multi-cloud setup is inherently difficult for the following reasons:

- **Heterogeneity** — Accessing multiple systems supported by different cloud storage providers introduces complexity to the application as it has to deal with different application programming interfaces (APIs).
- **Complex storage logic** — In many cases, the application has to implement complex storage logic in the application source code to achieve the composite benefits of multi-datastore and the multi-cloud setup.
- **Tenant customization** — Moreover, the tenants of a SaaS application commonly have different data storage requirements, usually related to non-functional such as data confidentiality, reliability, and security of data stored in the cloud. Especially in a multi-tenant context, the economic feasibility of the cloud offering relies extensively on providing the tenants some degree of on-demand self-service capabilities [22], i.e. allowing tenants to configure the SaaS applications autonomously, without requiring intervention from the SaaS provider.

To address these problems, we present a reusable, policy-driven data management middleware that (i) makes abstraction of different cloud storage technologies and thus also providers; (ii) follows a policy-driven approach for making data placement decisions and getting the benefits of multi-datastore and the multi-cloud setup; and (iii) allows the specification of storage configurations and advanced data storage policies, both by the service provider and the tenants. Our prototype implementation, on which we built a realistic multi-

978-0-7695-5696-3/15 $31.00 © 2015 IEEE
DOI 10.1109/BDC.2015.39

tenant and multi-cloud SaaS application validates the proposed middleware. Additionally, our evaluation efforts focus on the performance overhead.

The rest of the paper is structured as follows: Section II motivates this work from the context of a realistic SaaS application case, derives the problem statement, and identifies key goals of our middleware. Section III introduces the middleware architecture in support of policy-driven storage and multi-tenant customization, while Section IV discusses the validation of the proposed middleware in a prototype. Section V subsequently evaluates the performance overhead introduced by this middleware, and Section VI discusses related work. Finally, Section VII concludes the paper and indicates potential tracks for future research.

II. Motivation and Problem Statement

The motivation for this paper is based on our experiences with a number of multi-tenant SaaS applications, which have been studied in the context of several applied research projects in collaboration with industry [6], [7], [9]. A tenant is an organizational customer of a SaaS application and the customers of that organization are the end users of the application. A multi-tenant SaaS application serves multiple tenants and their end users at once [8], [18], [23]. In Section II-A, we introduce a specific SaaS application, on which we rely to derive the problem statement in Section II-B. Finally, we present the key goals of the middleware in Section II-C.

A. Application Case

The Log Management as a Service (LMaaS) application is a multi-tenant, business-to-business (B2B) cloud offering that takes over log management, analysis, and storage from the tenant organization. The tenants of the LMaaS application are customer organizations of all sizes (such as banks, supermarkets, hospitals). Different tenants of the LMaaS application have different, sometimes even contrasting requirements with respect to how the data should be dealt within such a multi-cloud storage environment. For example, for some tenants (e.g., banks), even log data is considered highly sensitive, and therefore they prefer to keep using their own on-premise storage infrastructures, whereas other tenants have no objections against having their log data stored in a public cloud.

Furthermore, the application deals with large amounts of heterogeneous data, coming from different sources, and many different data types with different storage requirements (raw log entries, log meta-data, archived logs, historical logs, and incident reports). Different requirements apply to these different data types, for example, raw log entries require high availability as well as high write and read throughput, whereas for historical logs high write throughput counts.

Examples of more fine-grained storage logic involve taking into account more specific data properties such as the object size. For example, incident reports of different data sizes have to be dealt with differently in the LMaaS application: if they are less than 20MB in size, they should be stored in an on-premise infrastructure, otherwise they must be placed in a public cloud. At the same time the requirements regarding data confidentiality and low latency search must be taken into account.

B. Problem Statement

A multi-cloud setup introduces substantial complexity in the application layer. Clearly, the LMaaS application needs to deal with this complexity to satisfy different application-wide as well as tenant-specific storage requirements. Specifically, the LMaaS application has to deal with heterogeneity in terms of different APIs. In addition, the application has to implement the complex storage logic (which usually involves (re)writing the source code) to achieve the benefits of both the multi-cloud and the multi-datastore setup. Finally, the application needs to provide configuration support to tenants, to allow them the self-service that is crucial for the economic feasiblity of a cloud offering.

C. Key Goals

We define the following key goals for our middleware:

1) **Uniform API** — The middleware needs to provide a uniform API for the widely different data storage systems, to allow application development to be technology and vendor independent.
2) **Multi-datastore support** — The middleware needs to support multiple datastores at once in order to achieve the benefits of a multi-cloud storage setup.
3) **Multi-tenant customization** — The middleware needs to be highly configurable and customizable to support the contrasting requirements of different tenants of the SaaS application.

III. Middleware Architecture

This section discusses the architecture of our proposed middleware platform that makes abstraction of multiple cloud storage providers and thus also of storage systems. In addition, the architecture supports flexible and dynamic data management using policies and offers multi-datastore as well as multi-tenant customization support. Figure 1 presents the architecture of our proposed middleware. The architecture is divided into four different layers: (i) the Multi-tenancy layer (optional for multi-tenant SaaS applications); (ii) the SaaS Application layer; (iii) the Data Management Middleware layer; and (iv) the Decentralized, Distributed Storage layer. The core of the middleware and the focus of this paper is the Data Management Middleware layer. The next section focuses on the Data Management Middleware layer with respect to different components offered by this layer.

A. Data Management Middleware Layer

As shown in Figure 1, the Data Management Middleware layer consists of three coarse-grained components: (a) the *Data Access Middleware* component, (b) the *Configuration Management* component, and (c) the *Storage Drivers* component. The *Storage Drivers* component addresses heterogeneity and hides the complexity of back-end systems, distributed across a

Listing 1: Annotations are supported on both the class-level as well as the field-level.

```
...
@Confidential
@IncidentReport
public class Report implements Serializable{
    ...
    @Size
    private double size;
    ...
}
```

multi-cloud setup by providing a uniform API. However, we omit this component for space reasons and rather focus on *Data Access Middleware* component and the *Configuration Management* component.

1) Data Access Middleware Component: The SaaS applications are developed on top of the *Data Access Middleware* component, independent of the underlying technologies supported by multiple cloud storage providers. To accomplish this, standardization is key: our middleware offers a Java Persistence API (JPA) and Java Persistence Query Language (JPQL) because they are a de-facto standard for developing Java applications [21].

The *Data Access Middleware* component provides an additional set of annotations to specify meta-information about the data. The proposed middleware supports an additional application-specific (e.g., `@RawLog`, `@IncidentReport`, `@Size`), technology-specific (e.g., `@Writeconsistency`, `@Readconsistency`), and the middleware-specific (e.g., `@Confidential`, `@Nonconfidential`) annotations. These annotations are supported on both the class-level as well as the field-level as shown in Listing 1. Developing and deploying a SaaS application on top of this middleware involves a set of configuration tasks. These tasks are accomplished in the form of placing additional annotations provided by the middleware platform and setting data storage policies as show in Figure 1. This component comprises of two main service components: (a) the `Data Management Service` component, and (b) the `Policy Management Service` component.

The `Data Management Service` component facilitates insert/create, read, update, and delete (CRUD) operations and provides an interface for SaaS applications to interact with the proposed middleware. This component reads meta-information about the incoming data and then calls the `Policy Management Service` component. The latter one is responsible to interact with the policy engine in order to evaluate the data storage policies based on the meta-information. Then, it returns the information to the `Data Management Service` component about the storage systems, best suited for storing the data. Based on the returned information, the `Data Management Service` component, then decides where the data needs to be stored and whether the data needs to be encrypted. The component calls `Encrypt/Decrypt` component, if the data needs to be encrypted or decrypted. The `Data Management Service` component then communicates with the `Configuration Management Service` component to get

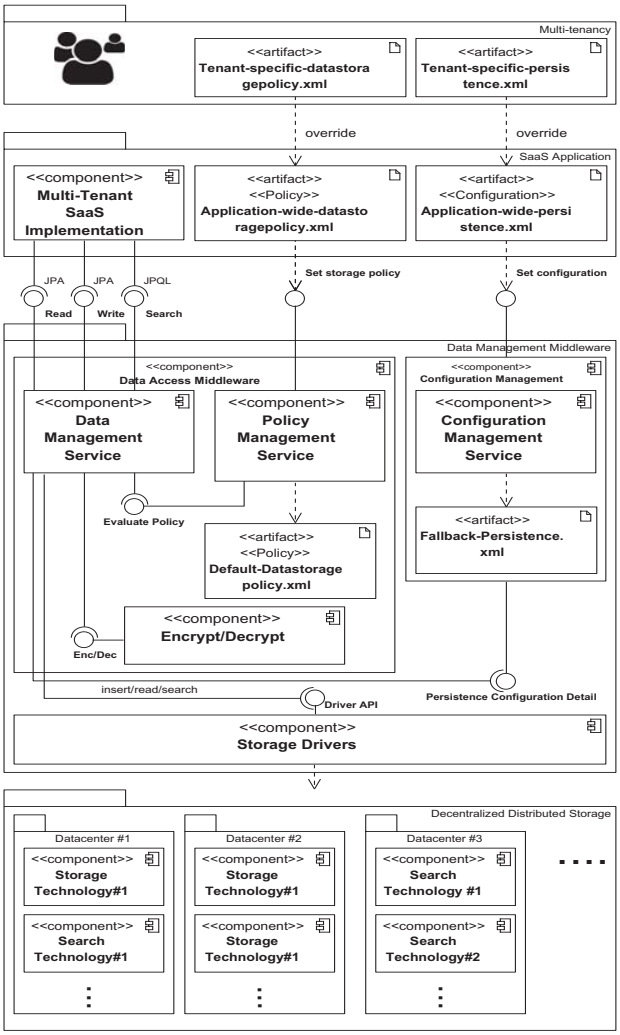


Fig. 1: Architecture of the proposed middleware platform for mulit-cloud storage in multi-tenant SaaS.

the storage configuration details about the back-end storage systems, best suited for storing data across multiple cloud storage providers. After getting the information, the `Data Management Service` component interacts with `Storage Drivers` component and overrides the configuration detail about the storage system, which is best suited for storing data.

The `Policy Management Service` component is responsible for the data storage policies and provides an interface for SaaS applications to set an application-wide policies. However, in order to cope with the different and varying tenant requirements, the component allows tenants to override an application-wide policies and set tenant-specific data storage policies. Listing 2 shows as an example of tenant-specific data storage policy. Moreover, in order to get greater fault tolerance, high availability, and reduced client-perceived latency, multiple storage systems, distributed across multiple cloud storage providers can be specified using (&&) separator.

Listing 2: Example of a tenant-specific data storage policy that allows tenants to use on-premise storage infrastructure.

```
1 ...
2 rule "dealing with confidential historical log data"
3 when
4 dsSelector:Entity(data == "confidential" and
      documentType == "historicallog")
5 then
6 dsSelector.setDatastore("cassandra_private");
7 end
8 ...
```

The Policy Management Service component also facilities tenants to configure and use on-premise storage infrastructure than relying on the SaaS provider storage facilities. The component uses the default data storage policies if none of the policies are specified as shown in Figure 1.

2) Configuration Management Component: The *configuration management* component is responsible for storing persistence configuration details of tenants and SaaS providers about the back-end storage systems. This component is composed of Configuration Management Service component, which provides an interface for tenants and SaaS providers, to set tenant-specific or application-wide persistence configuration files. The SaaS providers and tenants define the configuration details in storage-configuration.xml file.

IV. Prototype Implementation

We implemented a prototype on top of Kundera by extending it with an advanced middleware features support using the Java programming language. Kundera [14] is an open source project which provides drivers for different storage systems and addresses the heterogeneity among them. The advanced middleware features include the introduction of (a) policy engine, (b) multi-datastore support, and (c) multi-tenant customization support. In addition to this, the middleware provides a number of middleware-specific, application-specific, and the technology-specific annotations along with JPA annotations to specify additional information about the data. The prototype uses the Drools[1] for the policy evaluation, a rule engine that uses rule-based approach to implement an expert system. The prototype supports a number of different back-end systems, including in-memory storage systems (e.g., Ehcache, Redis), full-text search systems (e.g., Elasticsearch), SQL-based systems (e.g., MySQL) and NoSQL storage systems (e.g., Oracle NoSQL, Cassandra, MongoDB, HBase, Neo4J).

The middleware is deployed as a server on Tomcat 7[2] with an exposed interface to the client. On start-up, the middleware reads an application-wide storage configuration files and data storage policy files. In order to set the tenant-specific storage configurations and data storage policies, the tenant must override these files. All the meta-data and the policy decisions are cached and stored in Ehcache. Once the application-wide configuration files and the data storage policy files are configured, the middleware is ready to serve tenants requests.

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

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

V. Experimental Evaluation

In the decision to use the proposed middleware platform for alleviating complexity in the application layer, the impact on the application performance is a vital criterion. Therefore, two questions drive our experiments:

Q1. What is the impact on the performance in terms of overhead using the proposed middleware platform?

Q2. How the performance of the proposed middleware platform can be optimized?

The experiments are conducted using the LMaaS application discussed in Section II-A. All the experiments are designed for a storage scenario by taking into account the tenant-specific data storage policy (see Listing 3) that uses a combination of cloud storage providers.

Section V-A presents the benchmark and their setup. Then, Section V-B presents the results, which are then summarized in Section V-C.

A. Experimental Setup

The middleware platform is evaluated by storing tenant-specific data, which have different storage requirements, across five different deployment setups: (i) *Cassandra-private* contains a Cassandra instance deployed on a single node, (ii) *Cassandra-public* contains a three-node Cassandra cluster deployed in a private IaaS cloud lab, (iii) *MongoDB-private* contains a MongoDB instance deployed on a single node, (iv) *MongoDB-public* contains a MongoDB instance deployed on Morpheus[3] public cloud, and (v) *Elasticsearch-private* is deployed on a single node in a private environment.

TABLE I: Hardware Setup

Client Node	
Processor	Intel(R) Core(TM) i5 @ 2.60GHz (Dual)
Memory	8GB
Operating System	Windows 8
Server: 1 to 3 Nodes	
Processor	2 X Intel(R) Core(TM) 6400 @ 2.13 Ghz
Memory	8GB
Operating System	Linux/Ubuntu

We conducted four experiments in which the LMaaS application uses a combination of five deployment setups discussed above. In the first experiment, the application is configured to use *Cassandra-private* deployment for storing 1 280 000 confidential Raw Log entries as shown in Listing 3 from lines 2 - 7. In the second experiment, the application is configured to use *Cassandra-public* for storing 1 280 000 non-confidential Raw Log entries as shown in Listing 3 from lines 9 - 14. In the third experiment, the application is configured to store 1 280 000 confidential Incident Reports

[3]http://gomorpheus.com/

having size < 20 MB in *MongoDB-private* and size > 20 MB in *MongoDB-public* as shown in Listing 3 from lines 16 - 30. However, the Incident Reports are confidential and have low latency search requirements. Therefore, the Incident Reports need to be encrypted first before storing in *MongoDB-public* and must also be indexed in *Elasticsearch-private* to meet low latency search requirements. In the last experiment, the application is configured to use both the *Cassandra-private* and the *MongoDB-private* for storing 1 280 000 confidential Raw Log entries, which also have requirements with respect to confidentiality and high availability, as shown in Listing 3 from lines 32 - 37.

We compared two application setups for all the performance experiments, one with the proposed middleware layer enabled, and one without (where the application directly accesses each data storage system using Kundera platform which is the baseline for the performance comparison). The overhead is evaluated by comparing the performance of the middleware layer with the baseline. Table I lists the hardware used for the evaluation on both client and server sides.

B. Performance Impact Results

This section presents the results of our performance benchmarks and as such shows the performance in terms of overhead of the proposed middleware platform.

1) Performance Overhead: The performance overhead results of four experiments discussed in Section V-A are presented in Figure 2a, Figure 2b, Figure 2c, and Figure 2d respectively (addresses **Q1**). The proposed middleware with *cache-disabled* introduces, 236%, 17%, 60%, and 140% relative performance overhead compared to the baseline for all four experiments respectively.

The performance overhead results show that the overhead of the proposed middleware is significant. However, we were interested to inspect the total time each component of the Data Management Middleware layer takes to perform an operation. Therefore, we have measured the total time spent on different components of the middleware with *cache-disabled* by manually injecting profiling statements for all the four experiments. Table II shows the results of this experiment for Figure 2a.

TABLE II: Total time spent on each component of the middleware with *cache-disabled* for Figure 2a.

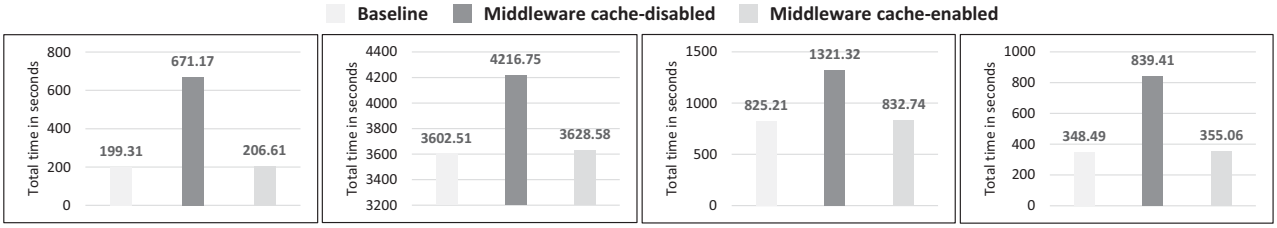
Component	Seconds
Middleware	
↳ ***JPA/Reflection (Data Management Service)***	***72.48***
↳ Data Management Decisions	2
↳ ***Policy Evaluation (Policy Management Service)***	***397.39***
↳ Baseline (Storage Drivers)	199.3
Total	671.17

We have learned that the significant overhead introduced by our middleware mainly corresponds to reading meta-data as well as evaluating and executing the policies at run-time.

Listing 3: Tenant-specific data storage policy contains rules for data distribution across multiple deployment setups.

```
1  ...
2  rule "dealing with confidential raw log data"
3  when
4  dsSelector:Entity(data == "confidential", type == "
       rawlog")
5  then
6  dsSelector.setDatastore("cassandra_private");
7  end
8
9  rule "dealing with non-confidential raw log data"
10 when
11 dsSelector:Entity(data == "nonconfidential", type ==
       "rawlog")
12 then
13 dsSelector.setDatastore("cassandra_public");
14 end
15
16 rule "dealing with confidential incident reports
       where the size of each incident report is less
       than 20 MB"
17 when
18 dsSelector:Entity(data == "confidential" and
       documentType == "incident-report" and size < 20)
19 then
20 dsSelector.setDatastore("mongodb_private");
21 end
22
23 rule "dealing with confidential incident reports
       where the size of each incident report is
       greater than 20 MB"
24 when
25 dsSelector:Entity(data == "confidential" and
       documentType == "incident-report" and size > 20)
26 then
27 dsSelector.encryptFirst(true);
28 dsSelector.setDatastore("mongodb_public");
29 dsSelector.setIndexstore("elasticsearch_private");
30 end
31
32 rule "dealing with confidential raw log data which
       also requires high availability"
33 when
34 dsSelector:Entity(data == "confidential", type == "
       rawlog", availability == "high")
35 then
36 dsSelector.setDatastore("cassandra_private &&
       mongodb_private");
37 end
38 ...
```

As shown in Table II, the Data Management Service component, which provides a JPA interface for the applications, takes 72.48 seconds to process 1 280 000 confidential Raw Log entries. The component requires inspecting the entity at run-time to read application-specific, technology-specific, and the middleware-specific annotations using the Reflection. The other component, which takes most of the time is the Policy Management Service component. This component uses the Drools policy evaluation engine and takes 397.39 seconds to process 1 280 000 Raw Log entries. The Policy Management Service component evaluates the data storage policy at run-time for each data storage request to make data placement decisions. The next section optimizes the performance of the proposed middleware. More specifically, the performance impact of the Data Management Service



(a) Insertion of confidential `Raw Log` entries in *Cassandra-private*.

(b Insertion of non-confidential `Raw Log` entries in *Cassandra-public*.

(c) Insertion of confidential `Incident Reports` of < 20 MB in *MongoDB-private*, > 20 MB in *MongoDB-public*.

(d) Insertion of confidential `Raw Log` entries in both *Cassandra-private* and *MongoDB-private for* high availability.

Fig. 2: Total time in seconds to insert 1 280 000 entries of different types using the proposed middleware platform *with and without performance optimization*.

component and the `Policy Management Service` component (printed in bold and italic in Table II) is taken into account.

2) Performance Optimization: In this section, we discuss how the performance of the proposed middleware platform can be optimized (addresses **Q2**). As discussed in the previous section, the `Data Management Service` component reads meta-data for each data storage request at run-time using the `Reflection`. For example, storing 1 280 000 `Raw Log` entries require inspecting the entity 1 280 000 times. We have realized that inspecting entity for each data storage request to get the meta-data is unnecessarily costly operation. The performance can be optimized by inspecting the entity only once at run-time and keeping meta-data in memory. The entity only needs inspection the next time when the meta-data changes. Similarly, the `Policy Management Service` component fires an event for each data access object to get the details about the data storage locations. This impact can also be reduced drastically by an efficient policy implementation mechanism where the policy decisions can be cached.

To illustrate this, we re-run the same experiments discussed in Section V-A, however, by inspecting the entity as well as evaluating and executing the policies only once at run-time and keeping both the meta-data and policy decisions in cache (with *cache-enabled*). The results of these experiments are presented in Figure 2a, Figure 2b, Figure 2c, and Figure 2d respectively. As shown, the overhead of using the proposed middleware with *cache-enabled* decreases drastically for all the experiments. For Figure 2a, we have looked at the total time spent on different components of the middleware that uses the cache. The results of this experiment are presented in Table III. The results indicate that the `Data Management Service` component now takes 2.31 seconds to process 1 280 000 confidential `Raw Log` entries, whereas for the same number of entries the `Policy Management Service` component takes 1.84 seconds. The performance improvement in both components also reflects the overall performance impact of the middleware, the overhead decreases significantly compared to previous experiments. The proposed middleware with *cache-enabled* introduces 3.66%, 0.72%, 0.91%, and 1.88% relative performance overhead compared to the baseline for all four experiments respectively.

TABLE III: Total Time spent on each component of the middleware with *cache-enabled* for Figure 2a.

Component	Seconds
Middleware	
↳ ***JPA/Reflection (Data Management Service)***	***2.31***
↳ Data Management Decisions	3.16
↳ ***Policy Evaluation (Policy Management Service)***	***1.84***
↳ Baseline (Storage Drivers)	199.3
Total	206.61

C. Discussion

Our proposed middleware supports CRUD operations across multiple data storage systems, however, this paper only focuses on the insert/create operation.

Overall, the proposed middleware platform provides an abstraction by hiding the complexity of multi-cloud storage architecture in the application layer and offers multi-tenant customization support. In addition, it helps to get rid of (re)writing an application code to: (i) support multi-cloud setup and thus also multiple storage systems, and (ii) address different tenant requirements in the application layer. The complexity of (re)writing an application code is taken away by defining policies that facilitate smart storage of an application data across multi-cloud storage architecture. The results indicate that the overhead of the proposed middleware platform is always less than 4%. The performance of the proposed middleware platform can be optimized and run-time overhead can further be reduced by inspecting the entities as well as evaluating and executing the policies at compile time only and keeping the decisions in memory.

VI. Related Work

In this section, we discuss some cloud storage works that are most relevant to our research. The problem of heterogeneity and the lack of standardization across different cloud storage providers in terms of storage systems has been attracting more attention lately from both the industry [14], [20] and the research community [2], [16], [19]. However, so far these studies have all concentrated primarily on heterogeneity and portability by building an abstraction layer. The abstraction layer supports application portability across multiple cloud providers with minimal migration efforts, compared to when the native API is directly used. Similarly, there are a number of studies [3], [4], [10] that provide a federated cloud storage system to integrate with diverse public cloud storage providers. However, to the best of our knowledge, none of these studies support policies that facilitate the smart storage of application data within and across multiple cloud storage providers and offer multi-tenant customization support.

Our work is similar to Tiera [17], a middleware that is flexible and enables a rich array of policies. While they focus on hiding the complexity of different interfaces, we leverage the ability to store a part of the application data that enables smart and efficient data storage while also optimizing the performance. In addition, the implementation of Tiera, only supports a single data center, whereas we enable the implementation of multi-cloud setup.

Another related solution is provided by Papaioannou et al. [15]. The authors presented Scalia [15], a cloud storage brokerage solution which is similar to HAIL [5] and inspired by RACS [1]. Scalia continuously adapts the placement of data based on data access patterns. Our work is similar to Scalia in aspects such as data distribution and the use of multi-cloud setup. However, we use policy-driven approach for the data placement decisions, whereas they focuses on data access patterns. In addition, we provide multi-tenant customization support, which they have not considered in their research.

VII. Conclusion

The benefits of a multi-cloud setup are compelling for both SaaS providers and tenants. However, managing a multi-cloud storage architecture in practice is not trivial as it introduces additional complexity in the application layer.

This paper presents a policy-driven data management middleware platform that (i) makes abstraction of multiple cloud storage providers, (ii) supports a rich array of continuously changing application-wide and tenant-specific storage policies for the data management, and (iii) supports tenant customization and refinement. We have validated the core concept by building a multi-tenant SaaS application, a log management prototype implementation on top of the proposed middleware that uses a combination of multiple cloud storage providers. The results of our experimental evaluation demonstrate that the benefits of the proposed middleware are achieved with minimal performance overhead.

There are a number of interesting variants to explore in the follow-up work. Firstly, we intend to investigate the problems related to implementing search operations in a multi-cloud storage setup and envision supporting different search strategies. Secondly, we plan to address the limitation of our middleware that stores metadata and policy decisions in the cache server that is deployed to a single geographical location. We aim at replicating metadata and pre-compiling policy decision services, thereby accommodating geographically distributed clients to realize low latency requirements.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS) and the iMinds DMS2 project, which is co-funded by iMinds (Interdisciplinary institute for Technology), a research institute founded by the Flemish Government. Companies and organizations involved in the project are Agfa Healthcare, Luciad, UP-nxt, and Verizon Terremark, with project support of IWT (government agency for Innovation by Science and Technology).

References

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: a case for cloud storage diversity. In *SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010.

[2] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: the sos platform. In *CAiSE '12 Proceedings of the 24th international conference on Advanced Information Systems Engineering*, pages 160–174. ACM, 2012.

[3] David Bermbach, Markus Klems, Stefan Tai, and Menzel Michael. Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *IEEE International Conference on Cloud Computing (CLOUD), 2011*, pages 452–459. IEEE, 2011.

[4] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernand Andre, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4), November 2013.

[5] Kevin D. Bowers, Ari Juels, and Alina Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009.

[6] CUSTOMSS. CUSTOMization of Software Services in the cloud. http://www.iminds.be/en/research/overview-projects/p/detail/customss, 2011. [Last visited on June 27, 2014].

[7] D-Base. Decentralized support for Business processes in Application Services . http://www.iminds.be/en/research/overview-projects/p/detail/d-base, 2014. [Last visited on June 27, 2014].

[8] Tom Desair, Wouter Joosen, Bert Lagaisse, Ansar Rafique, and Stefan Walraven. Policy-driven middleware for heterogeneous, hybrid cloud platforms. In *ARM '13 Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware*. ACM, 2013.

[9] DMS2. Decentralized Data Management and Migration for SaaS (iMinds ICON project). http://www.iminds.be/en/research/overview-projects/p/detail/dms2, 2014.

[10] Dan Dobre, Paolo Viotti, and Marko Vukolic. Hybris: Robust hybrid cloud storage. In *SOCC '14*, pages 1–14. ACM, 2014.

[11] Brian F. Cooper, Adam Silberstein, Erwi Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10*, pages 143–154. ACM, 2010.

[12] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):2–22, December 2013.

[13] Till Haselmann, Gunnar Thies, and Gottfried Vossen. Looking into a rest-based universal api for database-as-a-service systems. In *12th IEEE International Conference on Commerce and Enterprise Computing*, pages 7–24. IEEE, 2010.

[14] Impetus. impetus-opensource/Kundera . https://github.com/impetus-opensource/Kundera, 2015.

[15] Thanasis G. Papaioannou, Nicolas Bonvin, and Karl Aberer. 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.

[16] Ansar Rafique, Stefan Walraven, Bert Lagaisse, Tom Desair, and Wouter Joosen. Towards portability and interoperability support in middleware for hybrid clouds. In *CrossCloud 2014 : IEEE INFOCOM CrossCloud Workshop*. IEEE, 2014.

[17] Ajaykrishna Raghavan, Abhishek Chandra, and Jon Weissman. Tiera: towards flexible multi-tiered cloud storage instances. In *Middleware '14 15th International Middleware Conference*, pages 1–12. ACM, 2014.

[18] Anna Schwanengel and Uwe Hohenstein. Challenges with tenant-specific cost determination in multi-tenant applications. In *Proceedings of CLOUD COMPUTING 2013*, 2013.

[19] Rami Sellami, Sami Bhiri, and Bruno Defude. Odbapi: A unified rest api for relational and nosql data stores. In *IEEE International Congress on Big Data (BigData Congress)*, pages 653–660. IEEE, 2014.

[20] Spring. Spring Data. http://projects.spring.io/spring-data/, 2015.

[21] Uta Storl, Thomas Hauf, Meike Klettke, and Stefanie Scherzinger. Schemaless nosql data stores – object-nosql mappers to the rescue? In *BTW*, 2015.

[22] Wei Sun, Xin Zhang, Chang Jie Guo, Pei Sun, and Hui Su. Software as a service: Configuration and customization perspectives. In *Congress on Services Part II, 2008. SERVICES-2. IEEE*, pages 18–25, Sept 2008.

[23] Stefan Walraven, Eddy Truyen, and Wouter Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. In *ACM/IFIP/USENIX 12th International Middleware Conference*. ACM, 2011.