SSICLOPS

Design of Inter-Cloud Security Policies, Architecture, and Annotations for Data Storage

SSICLOPS Deliverable D2.2

Felix Eberhardt[2] Jens Hiller[1] Oliver Hohlfeld[1] Stefan Klauck[3] Max Plauth[2] Andreas Polze[2] Matthias Uflacker[3] and Klaus Wehrle[1]

[1]*Communication and Distributed Systems, RWTH Aachen University, Germany*
[2]*Operating Systems and Middleware Group, Hasso Plattner Institute, Germany*
[3]*Enterprise Platform and Integration Concepts Group, Hasso Plattner Institute, Germany*

January 29, 2016

This paper has received funding from the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 ("SSICLOPS"). It reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

Contents

Executive Summary

Today, cloud providers define their rights with respect to handling of user provided cloud data themselves within the boundaries given by law. They specify these rights in privacy policy statements that users have to accept when using a cloud service. Users of cloud services themselves are unable to specify their expectations on data handling for their very own data that they provide to the cloud. Moreover, they often underestimate the possibilities that cloud providers have with respect to data handling in the limits of the law, do not read privacy policy statements or misunderstand them. Consequently, users lose control over data that they provide to cloud services. In this deliverable we describe the design of a privacy policy language that enables cloud users to specify their own expectations on handling of data that they provide to the cloud. Thereby, the policy language takes into account the special requirements raised by the application in the cloud scenario, which raises special needs regarding efficiency, scalability, expressiveness, and flexibility.

To evaluate the applicability of our privacy policy language design, we present an exemplary application scenario that involves the provisioning of Hyrise in-memory database instances in an *OpenStack*-based cloud environment. For both Hyrise and *OpenStack*, we point out implementation approaches for integrating certain policy attributes into the existing components. However, local testing is merely adequate in order to derive reliable statements about the correct behavior of policy enforcement mechanisms in distributed, cloud-based setups. Using nested virtualization, we address this deficiency by providing a testbed that resembles the basic properties of a production-grade environment. The testbed is comprised of multiple *OpenStack* instances, that form a federation of private cloud infrastructures. This facility enables us to employ the means of fault injection to study and improve the behavior of policy enforcement mechanisms even in degraded system states. To achieve these goals, we are going to build up on top of our preceeding efforts in the project, where we introduced replication mechanisms to Hyrise [12] and presented a virtualized single-site *OpenStack* testbed that enables us to perform fault injection experiments [2].

1. Introduction

Today, cloud providers treat data of customers according to their own rules, their privacy policy. They present these to their customers in privacy statements. Lacking a channel in the opposite direction, the users are unable to specify their own requirements regarding the handling of their data that they provide to cloud services [6, 7]. Consequently, users only have the choice to either accept the data handling by the cloud provider as specified in the privacy statement or to refrain from the usage of the cloud service. Additionally, cloud providers typically provide only a single privacy statement which specifies the handling procedures for all data. This neglects the different levels of privacy required for different data, e.g., banking information or social security numbers require significant efforts with respect to privacy handling while other data such as the employer of a person can be less privacy sensitive. However, with a single privacy statement, all data is handled the same way, either ignoring the increased demands of very privacy sensitive data or waisting resources for protection mechanisms for data that does not need it. These problems are not limited to private users, but also affect business users, e.g., 55% of IT professionals rank the lack of control over data in public clouds as one of their top three concerns [9]. Furthermore, 78% need to comply with regulatory mandates and 57% of those refrain from storing data that is affected by these regulations in private or public clouds [9]. The ability to control where data is stored and processed as well as the ability to compare security levels of different cloud providers would improve their confidence to make use of the cloud [9]. Thus, the ability of users to specify their own expectations on the data handling by cloud services, as well as the handling of data based on the individual requirements of this data is inevitable for future cloud environments that handle privacy sensitive data [6, 7].

Our approach to enable an automatic handling of user expectations employs a privacy policy language that enables users to specify their expectations on the data handling individually for each data item. In Deliverable 2.1 [1] of the SSICLOPS project, we provided a detailed requirements analysis for such a privacy policy language. We derived that a policy for a data item should travel with this data as a small data annotation [6, 7] that allows evaluation of the policy whenever required, e.g., when selecting a storage place for data, decrypting data for processing, or even determining allowed routing paths between data centers. Furthermore, we analyzed existing (privacy) policy languages with respect to their applicability to the cloud scenario. This analysis showed shortcomings for all of these analyzed policy languages in this setting. In this deliverable, we present our design of a privacy policy language that fulfills the derived requirements for the application in the cloud scenario.

In addition to providing the privacy policy language design, this document also discusses strategies for implementing and integrating policy concepts into established software components. We present an exemplary use case that involves the provisioning of Hyrise in-memory database instances in an *OpenStack*-based cloud environment. For both Hyrise and *OpenStack*, we point out implementation approaches for integrating a certain subset of policy attributes into the existing components. In our preceding efforts, we created a reproducible, virtualized testbed that resembles the basic properties of a production-grade *OpenStack* installation [2]. Furthermore, the design of the testbed enables us to apply fault-injection mechanisms to study the behavior of *OpenStack* components in the presence of system faults. To support the development and evaluation process of policy enforcement mechanisms, we build up on top of our previous efforts and extend our virtualized testbed to accommodate multiple *OpenStack* instances that form a federation of private cloud setups within a single machine. This facility enables us to employ the means of fault injection to study and improve the behavior of policy enforcement mechanisms in a degraded system state.

The structure of this deliverable is as follows: Section 2 outlines the cloud scenario already presented in Deliverable 2.1 and recaps the corresponding derived requirements. We present our design of a privacy policy language for this scenario in Section 3. In Section 4, we present implementation strategies for integrating the policy language concepts by example of a use case that involves the in-memory database Hyrise and its scale-out implementation *Hyrise-R*, as well as a federated *OpenStack* environment. We discuss how both components can be extended to support selected policy descriptions. Furthermore, we point out how a virtualized *OpenStack* Cloud federation testbed can be leveraged to evaluate the correct functionality of policy enforcement mechanisms using the means of fault injection. We conclude the deliverable in Section 5 and give an outlook on upcoming tasks.

2. Cloud Scenario

When users employ cloud services, they hand over their data to the cloud providers that handle this data based on their privacy policies. These privacy policies are rather static statements dictated by the cloud provider. Most notably, users do not have the possibility to negotiate the applied privacy rules with the cloud provider. Thus, users have to accept the privacy policy of the cloud provider or refrain from usage of the service. If they decide to use the service and hand over their data to the cloud provider, they lose control of their data in the range specified by the privacy policy [6, 7]. Notably, users are not only unable to specify their own requirements for the handling of their data. Additionally, privacy statements employed by cloud providers to communicate their privacy policies to users are legal documents that are often hard to read and understand. Most users do not read these legal documents, but assume that their rights are the same as if they would store the data on their local devices [10]. Hence, users hand over their data without any knowledge of what the cloud provider will and will not do with it.

However, there are plenty of different expectations that users would like to express to stay in control of the handling of their data that is transferred to the cloud [6, 7]. As an example, Figure 1 shows a user that selects the storage location of the database that handles the cloud data. Other expectations are the selection of encryption methods, kind of storage, guaranteed data deletion, access logging or offered redundancy [1, 10]. In this deliverable, we describe the design of a privacy policy language that allows users to express such expectations on data handling to cloud providers. The formulation of these enables cloud providers to consider the various different privacy expectations of different users and even allows them to apply rules that are specific for a single data item. As expectations based on policy languages also provide the ability for automatic

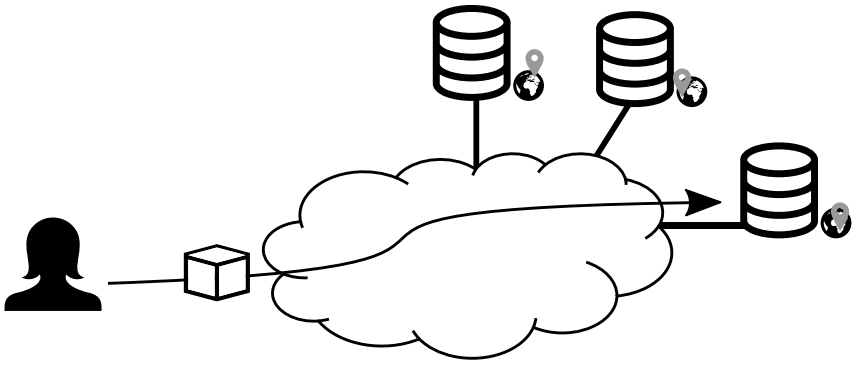

Figure 1: Cloud scenario with databases at different locations.

processing and complying with user expectations, they are appropriate for the usage in the cloud scenario [6, 7], which requires a highly scalable solution.

In the following, we give a short overview on the most important requirements that such a privacy policy language must fulfill to be eligible for the cloud scenario, before we present our design of a privacy policy language for the cloud in Section 3.

2.1. Requirements

Supporting possibly different expectations for each data item requires this expectation to be available whenever an action is performed with the data, e.g., determining the storage location of data or retrieving it for processing. In order to ensure the availability of the expectation for these checks, the user expectations can be attached to the corresponding data item as data annotation [6, 7]. The primary requirements for a privacy policy language that supports the data annotation mechanism and is suitable for the cloud scenario are (i) a small storage footprint, (ii) human readability, (iii) detection of conflicts at time of specification, (iv) sufficient evaluation performance, (v) adequate expressiveness for the cloud scenario, (vi) extensibility for adaptation to new policy statements that come up with emerging cloud services, and (vii) a matching mechanism that checks if the offer of a cloud provider fulfills the expectations formulated in the policy [1].

In the following we shortly describe and argue for each of these requirements. We refer to Deliverable D2.1 [1] for an extended analysis. Furthermore, Deliverable D2.1 contains an in depth analysis of existing policy languages concluding that no existing policy language fulfills all requirements for the cloud scenario.

(R1) Storage footprint When user expectations travel with the respective data items as data annotations, they add transmission and storage overhead. This overhead is determined by the storage footprint of the data annotation that, therefore, must use a concise representation for the policy.

(R2) Readability An instance of the policy language represents the user's expectations regarding handling of data by cloud services. Hence, users, who typically are not experts in the domain of policy languages and formal logics, nevertheless need to be able to understand their specified expectations. Notably, they may not be required to specify the policy themselves as they can retrieve prepared policies from trustworthy institutions, e.g., the office of fair trading, experts in the field of cloud privacy, or use easily understandable policy generators. Nevertheless, they should be able to verify if a formula of the policy language matches their expectations. Notably, this requirement is contrary to the requirement of a limited storage footprint (R1) as highly compressed data is typically readable for machines, but not for humans.

In addition to readability for humans, policies must also be readable by machines in order to support automated evaluation of user expectations.

(R3) Conflict detection Formulation of logic rules bears the risk to generate expectations that are not satisfiable, especially when created by non-experts like cloud users. However, as the cloud handles data in the background, users may not be available at the time when a conflict is detected. Hence, the policy language should support the detection of conflicts at time of specification. In this case, the user is available for feedback and can resolve the conflict before it is actually used for data in the cloud.

(R4) Performance Policies must be evaluated at numerous circumstances, e.g., relocation or replication of data and processing of data. Hence, the overhead for checking if expectations of a policy match with the properties of a data center or server must be limited. This performance requirement is especially important as cloud providers may refrain from supporting the policy language mechanism if it significantly increases the load of their cloud instances.

(R5) Expressiveness Cloud services provide diverse services to users. Hence, there is a large spectrum of expectations for handling of data that users may want to negotiate, e.g., (i) restriction of storage location to a certain continent or country, (ii) deletion of a data item at a specified point in time, (iii) logging or notification when data is accessed by a third party, or (iv) replication rate of data to ensure availability [1]. A privacy policy language for the cloud must provide users the ability to express expectations for these various kinds of data handling. This especially requires the support of *environmental context* (e.g., (i)), *time-based triggers* (e.g., (ii)), and *event-based triggers* (e.g., (iii)).

(R6) Extensibility Cloud services are subject to continuous change especially due to emerging new services. With such new cloud services, new requirements regarding the specification of expectations by users can emerge. Thus, a privacy policy language for the cloud needs to be extensible in order to support the formulation of these new expectations.

(R7) Matching Today, cloud providers specify their handling of data in static privacy policy statements. Therefore, it is not only required that users can specify their expectations on data handling to the cloud service, but also cloud providers must be able to specify what their data centers can offer and what they are willing to offer the users. The specification of cloud provider interests is important as, e.g., cloud providers must adhere to local jurisdiction, i.e., they may not be able to fulfill all expectations of a user even if it would be technically possible. Hence, the policy engine must not only be able to check if a data center can address the expectations technically, but also if the cloud provider is willing to fulfill the expectations, i.e., if user expectations and provider privacy policy are compatible.

3. Policy Language Design

Existing policy languages do not support the requirements listed in Section 2 [1]. In this section, we describe the design of our privacy policy language that tackles the challenges emerging in the cloud scenario.

Figure 2 gives an overview on the design of our privacy policy language. A user has an expectation on the handling of her data and providers provide an offering, which may be derived from their current privacy policy. Together with the actual properties of a data center, the *provider offering* specifies which expectations a data center can fulfill. When the user provides data to the cloud, a *policy decision point* (PDP) in the vicinity of the user determines suitable routing paths and data centers. This requires knowledge of the user expectations for the data item and the offerings of the available data centers. To this end, the expectations for a data item are attached as *data annotation* to make it available for the decision process. In order to decrease the networking overhead, expectations are compressed before they are attached to the data item as annotation.

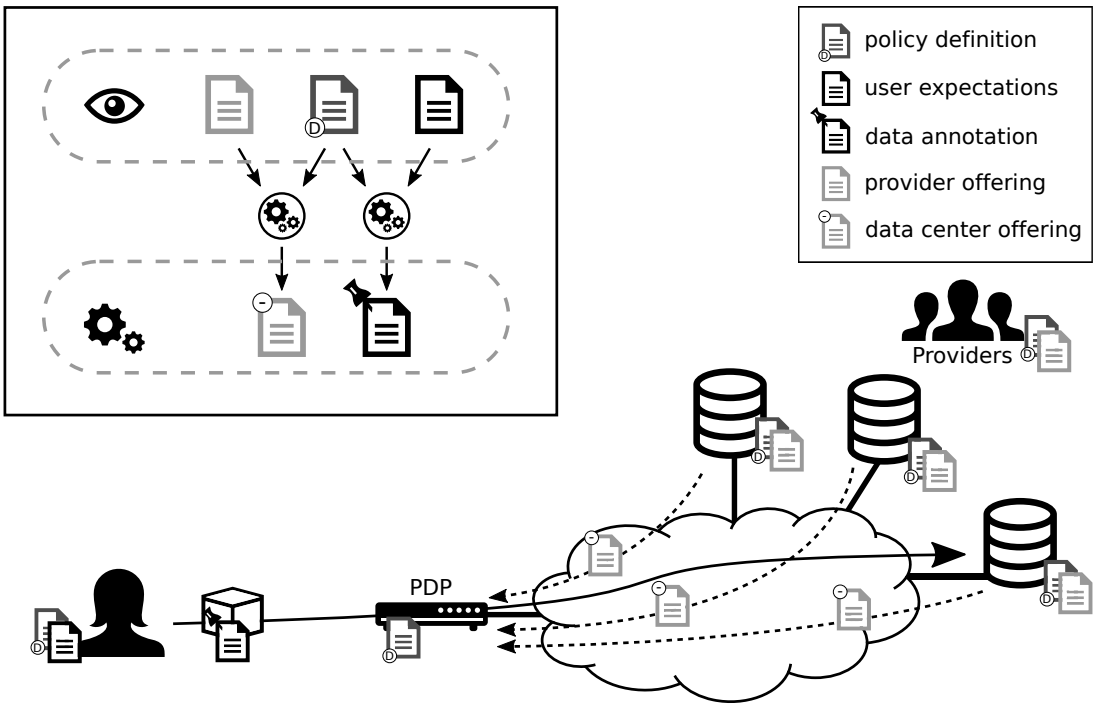

Figure 2: Overview on policy language design.

When a PDP performs matching of expectations and data centers, the PDP also requires access to the *data center offerings*. To this end, data centers periodically provide the additionally required data center offerings to the PDP. Similarly to the data annotations, these data center offerings are compressed before transmission. For the compression to data annotations and data center offerings, every entity has access to a *policy definition*. The policy definition specifies a cloud domain specific compression for the textual representation of user expectations and provider offerings. Especially the (compressed) data annotations are no longer readable for humans, but optimized on the level of bits and therefore enable efficient utilization of network resources. Still machines, especially the PDP, can efficiently check if the expectations of a user and the offerings of a database match and select an appropriate data center for the supplied user data.

In the rest of this section, we describe the structure of human readable expectations and provider offerings. We then cover the domain specific compression of expectations to data annotations and describe the matching process that checks if a data center fulfills the expectations for an annotated data item.

3.1. User Expectations and Policy Definition

In our policy language, users express their expectations in logic formulas. These formulas need to be easy to read (R2). Therefore, our policy language expresses user expectations as text based boolean formulas. The limitation to simple boolean formulas enables even users that do not have strong background knowledge in logics to determine the meaning of a given user expectation. Still, we show that our boolean formulas provide sufficient expressiveness for the cloud scenario (R5).

Listing 3.1: User expectation example.

```
1  storage.provider != "CompanyA"
2  & storage.log_access = true
3  & storage.deleteAfter(1735693210)
4  & storage.notify("delete", "email", "example@example.eu")
5  & storage.backupHistory("1M")
6  & ( storage.location = "DE"
7      | ( storage.location = "EU"
8          & storage.fde.activated = true
9          & storage.fde.algorithm = "aes"
10         & storage.fde.keySize = 256
11       )
12   )
13 & storage.replication >= 2
14 & storage.availability >= 0.99
```

Listing 3.1 shows an example of a user expectation. The expectation lists the requirement to not store the data at cloud storage of company A (l. 1), which is useful when using federated

clouds that are built up from clouds of different companies. Furthermore, the user expects the cloud to log data access (l. 2), e.g., when cloud services access the data for processing tasks. The expectation also expresses that the cloud needs to delete the data at time stamp 1735693210 (01 January 2025 1:00) and the requirement to report the deletion of the data via email (ll. 3-4). Additionally, the user expects the cloud to save a backup of the current data every month (l. 5), such that a history of data changes is available. Moreover, the data must be stored on a data center located in Germany (l. 6) or, alternatively, at any data center located in the EU that uses full disk encryption based on the cryptographic algorithm AES with 256 bit keys (l. 12). Finally, the user expects her data to be replicated at least two times (l. 13) and a data availability rate of 99% (l. 14).

To provide this information in the formula, the policy language specifies different variable types. The example shows the usage of boolean variables (true), numeric variables (256) and strings ("CompanyA"). Furthermore, the formula contains keys such as *storage.provider* which depend on the actual data center that may be eligible to store the data item. A list of available keys is given by the policy definition. Users form variable values and keys to relations, e.g., $\textit{storage.provider} \mathrel{!}= \textit{"CompanyA"}$, each of them resolving to a truth value when inserting the actual data of a server for the key. Users connect these relations to boolean formulas that a server must fulfill to be eligible to handle the corresponding data item. Additionally, our policy language supports functions such as *storage.notify*, which provide an increased expressiveness for user expectations.

The keys that are available for a formula depend on the underlying policy definition. An example for a policy definition that lists the keys for the expectation example above is given in Listing 3.2. The policy definition is formulated in standard JSON syntax. For each key, the definition specifies the name and the variable type (cf. l. 7), e.g., *boolean*, *string*, *int32*, *float32*, or *function*. In the special case of a finite value set, e.g., location values (cf. ll. 18-19), a values field lists the values that this key can take. This approach enables enumeration of the values, which is beneficial for the compression as detailed in Section 3.2.3. Additionally, if a key has the type function, the policy definition gives a tuple for the types of the function parameters (cf. ll. 9-11, 12-15). In order to increase the readability, keys can be summarized in groups such as *storage* (l. 5) or *fde* (l. 22). Other natural groups for the cloud scenario would be *processing* and *routing*. These groups could also be used to skip keys if they are irrelevant for the planned action, e.g., keys that specify the handling of data during processing are not necessarily relevant when deciding where data should be stored.

In the rest of this subsection, we describe the grammar of user expectations in detail. We first cover the usage of variable types to form relations and functions. Afterwards we describe the connection of relations and function into a boolean formula, the user expectation.

Listing 3.2: Policy definition example.

```
{
    "identifier": 0, /* 16 bit identifier of the policy */
    "variables": [
        {
            "name": "storage",
            "variables": [
                { "name": "provider", "type": "string" },
                { "name": "log_access", "type": "boolean" },
                { "name": "deleteAfter", "type": "function",
                    "parameters": [ "int32" ]
                },
                { "name": "notify", "type": "function",
                    "parameters": [ "string", "string", "string" ]
                    /* action, notification_type, address */
                },
                { "name": "backupHistory", "type": "function",
                    "parameters": [ "string" ]
                },
                { "name": "location", "type": "string",
                    "values": ["DE", "FR", "US", "GB", "NL", "EU"]
                },
                { "name": "fde",
                    "variables": [
                        { "name": "activated", "type": "boolean" },
                        { "name": "algo", "type": "string" },
                        { "name": "keySize", "type": "int32"}
                    ]
                },
                { "name": "replication", "type": "int32" },
                { "name": "availability", "type": "float32" }
            ]
        }
    ]
}
```

3.1.1. Expectation Atoms

As described in the example, user expectations consist of relations and functions that are interconnected as boolean formulas. We first describe the available variable types, i.e., *truth values*, *numeric values*, *character strings*, and *enumerations*. The differentiation into these different classes is important to enable efficient compression (R1). At the same time, the described atoms provide sufficient expressiveness for the cloud scenario (R5). Afterwards, we describe the grammar that specifies the usage of variable types in order to form relations and functions.

3.1.1.1. Atomic Variable Types

Variable types are the basic atoms in our privacy policy language. Each key, variable value and parameter of a function has a specific variable type. For the users, truth values, numeric values and character strings are readable and comprehensible.

Boolean Variables Users employ truth values in their expectation formulas to express the enforcement or prohibition of predefined data handling actions. Truth values can, e.g., be used as type for keys that specify the protection of data on storage with full disk encryption, dictate the storage on volatile storage, or forbid the transfer to third parties.

Numeric Variables Boolean variables enable a concise representation of variables that can be true or false. However, they do not scale when variables have to express numbers, e.g., the replication rate. Therefore, our privacy policy language supports the use of numeric values.

Strings Even more flexibility than with numeric values can be achieved with the usage of character strings. They enable the usage of a large variety of identifiers, e.g., specification of access rights for a specific person. However, the encoding of character strings requires significantly more space than numeric or truth values. Therefore, the other constructs should be preferred whenever possible or, if the set of possible values for a string variable is limited, an enumeration should be used.

3.1.1.2. Enumerations

Especially for character strings, the flexibility of supported values results in a hardly compressible encoding. However, strings still allow for a very efficient compression if the actual values of a string are limited to a finite set. An enumeration is a special variable type that leverages the possibilities of string variables with a finite value set to improve compression results. An example is the key that specifies the storage location for a data item. When using the two character country code, this provides a limited list of possible values for this key. This enables the enumeration of the possible values as given by the occurrence in this list. Another example is the specification of

applicable encryption algorithms. Notably, it would be possible to encode such keys as numeric variable directly, however, this would negatively affect the readability of the user expectation (R2).

In order to enable a suitable compression and readability for this type of expectations, the policy definition contains the finite list of possible values for the key as already shown in Listing 3.2 (ll. 19-21). Overall, enumerations enable users to specify enumeration based values in readable form, but also support an efficient compression.

3.1.1.3. Relations

Users employ the variable types described in Section 3.1.1.1 and Section 3.1.1.2 to compare the properties of a data center with desired values for the handling of their data. Therefore, they use the keys that are specified in the policy definition to create relations. Each relation is a tuple of key, comparison method, and a variable value of the type specified by the key. However, for keys of the type boolean variable, the comparison method and the constant can be omitted for conciseness. Thereby, the comparison method defines the *relation type*. The possible relation types depend on the variable type of the key as follows: Keys that are resolved to a boolean variable can employ negation (!), numeric variables support equality ($=$), non-equality ($\neq$), greater ($>$), greater equal ($\geq$), less ($<$), and lesser equal ($\leq$). Finally, strings as well as enumerations support equality ($=$) and non-equality ($\neq$). Notably, values of enumerations can only take the values supported as specified in the policy definition.

Overall, we obtain the following grammar for relations (R):

$$
\begin{array}{lcl}
R & \rightarrow & key_{bool} \\
R & \rightarrow & ! \; key_{bool} \\
R & \rightarrow & key_{num} = const_{num} \\
R & \rightarrow & key_{num} \neq const_{num} \\
R & \rightarrow & key_{num} < const_{num} \\
R & \rightarrow & key_{num} \leq const_{num} \\
R & \rightarrow & key_{num} > const_{num} \\
R & \rightarrow & key_{num} \geq const_{num} \\
R & \rightarrow & key_{string} = const_{string} \\
R & \rightarrow & key_{string} \neq const_{string} \\
R & \rightarrow & key_{enum} = const_{enum} \\
R & \rightarrow & key_{enum} \neq const_{enum}
\end{array}
$$

3.1.1.4. Functions

So far, relations provide a suitable expressiveness for some expectations of users. However, more expectations with very flexible input such as specification of notification in certain circumstances such as data access, specification of backups within specific time intervals, or requiring data deletion at a specific point in time can not be represented by keys in a scalable fashion. Furthermore, these expectations require a more detailed analysis if they are supported by the data center, e.g., a data center may impose minimal and maximal durations for the lifetime of a data item, which would affect the supported values for expected data deletion. In order to determine if a data center provides the desired handling of data, users specify these types of expectations as functions, e.g., *storage.notify("delete", "email", "example@example.eu")*. When a data center sends a data center offering to the PDP, it also includes a small script that checks if the function is supported with respect to the input parameters. Similar to relations, the function evaluates to *true* if the database supports the expectation given by the function and its parameters, or *false* otherwise. Similar to keys, the available functions are listed in the policy definition, together with the variable types of the parameters for the function.

In the context of our grammar, functions behave like variables, i.e., they are used to forming relations. Similar to boolean variables, the comparison method of a relation that contains a function and the constant can be omitted, and negation is supported:

$$\begin{aligned} R &\rightarrow function(param1, param2, ..., paramN) \\ R &\rightarrow \; ! \, function(param1, param2, ..., paramN) \end{aligned}$$

3.1.2. Expectation Formulas

The final expectation of a user is formed by relations and functions, each representing a single expectation. Users employ the logical and (&), logical or (|), and negation (!) to interconnect the relations and functions. Furthermore, the support of brackets enables the creation of concise formulas and increases readability (R2). The concluding part of the grammar for a user expectation (E) using relations (R) is as follows:

$$\begin{aligned} E &\rightarrow R \\ E &\rightarrow \; ! \, E \\ E &\rightarrow (E) \\ E &\rightarrow E \; \& \; E \\ E &\rightarrow E \mid E \end{aligned}$$

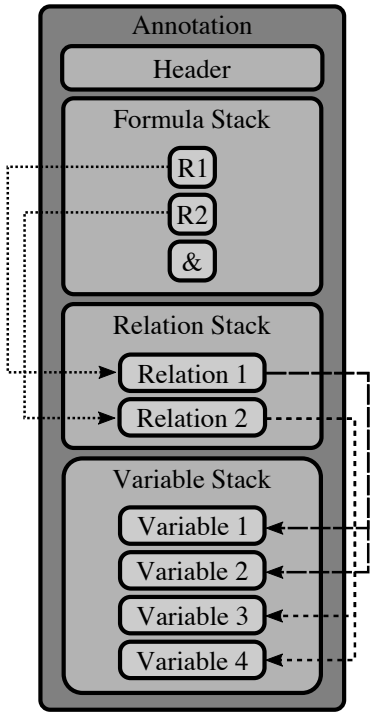

Figure 3: Compressed user expectation.

3.2. Compression to Annotation

User expectations as discussed so far are comprehensible and readable for users, but encompass overhead. More specifically, as expectations travel with their data item, an expectation increases the required bandwidth for each transmission of a data item, e.g., when it is relocated for processing. Furthermore, the required storage space that providers must supply and for that users possibly are charged increases. To reduce this overhead, expectations are compressed to data annotations before they are attached to data items.

The compression of user expectations to data annotations mainly depends on the policy definition. First, the policy definition defines the available keys and functions. In the expectations, these keys and functions are human readable identifiers that can be compressed to numerical values in the data annotation. The numerical value of a key or function is assigned based on order of occurrence in the policy definition. Notably, this compression is reversible if the data annotation contains a reference to the used policy definition. Additionally, the policy definition enables compression of enumerations. The list of predefined possible values is used to translate each value into a number, which enables space efficient encoding of enumerations.

The actual encoding of an annotation is illustrated in Figure 3. An annotation is split into four parts (i) the header stores an identifier for the policy definition used for the following encoding, (ii) the *formula stack* stores the logic operations connecting the relations of a formula, (iii) the *relation stack* encodes relation information, and (iv) the *variable stack* stores the actual variable

values and keys. This separation allows leveraging redundancies for compression, i.e., if a relation is used repeatedly in an expectation, the relation stack contains it only once and the formula stack references it repeatedly. For example, if *Relation 2* would be a complete copy of *Relation 1* (cf. Figure 3) *R1* would reference *Relation 1* such that *Relation 2*, *Variable 3*, and *Variable 4* could be omitted. Similarly, redundantly used variables are encoded only once and referenced multiple times by the relation stack, e.g., if *Variable 4* would equal *Variable 1*, *Variable 4* would be omitted and *Relation 2* would reference to *Variable 1*. The following subsections describe the encoding of the stacks and the referencing between them in more detail.

3.2.1. Formula Stack

The formula stack encodes the interconnection of relations, i.e., it must encode negation, logical and, logical or, evaluation order as given by the brackets, and references to relations. In order to save the space for an explicit encoding of the evaluation order, the formula stack uses *reverse polish notation*. Hence, we only require encodings for negation, logical and, logical or, and references to relations in the relation stack. These four possibilities can be encoded with two bits. However, each relation reference need to point to a relation in the relation stack. Intuitively, the bit sequence that identifies a relation reference can be appended with an address specifying a relation in the relation stack. However, if the relations in the relation stack are ordered based on the occurrence of their reference in the formula stack, the formula stack can omit the address. Instead, the policy engine can jump to the next relation on the relation stack.

Nevertheless, in order to leverage redundant relations for compression, the second reference to an already referenced relation needs to make use of such an address. Therefore, we introduce a *next relation* bit sequence, which references to the next relation on the relation stack, and a new bit sequence for a *redundant relation*, which is appended with an address to the relation stack. With this approach, we require bit sequences for negation, logical and, logical or, next relation, and redundant relation, which requires three bits. However, employing De Morgan's law it is possible to defer the handling of negations to the relation stack such that the formula stack only needs four different bit sequences. Consequently, a two bit sequence suffices.

Finally, to determine the end of the formula stack, i.e., the beginning of the relation stack, we could employ a length field. However, a more space efficient approach is to include an additional bit sequence to indicate the end of the formula stack. Furthermore, a length field would limit the maximal length of the formula stack while an additional bit sequence supports formula stacks of arbitrary length. In order to not increase the size of the relation bit sequences, we employ the first bit followed by a redundant relation bit sequence. This bit indicates either the end of the formula stack, or the presence of a redundant relation. In the latter case, the bit is followed by the address of the redundant relation.

Summing up, the formula stack contains the formula in reverse polish notation and encodes logical and, logical or, next relation, and redundant relation symbols within two bit sequences. Thereby only redundant relation symbols are followed by an additional signaling bit, which either

indicates the end of the relation stack or a following address that references a relation on the relation stack.

Note that the number of bits reserved for the address of a redundant relation is fixed and specified in the policy definition. Otherwise, each address would require a length field which would add additional overhead for each redundant relation. However, the fixed address length limits the maximal number of relations that the redundancy mechanism can reference, i.e., only the first relations up to the number representable with the address can be referenced. Thus, an increased address length can, depending on the existence of redundancy, make better use of the redundant relation mechanisms, but increases the overhead introduced for each redundant relation.

3.2.2. Relation Stack

The relation stack uses an encoding that is similar to the formula stack. It encodes the relation types $= True$, $= False$ (both for boolean variables only), $=$, $\neq$, $>$, $\geq$, $<$, and $\leq$ in a three bit sequence. Each of these relation bit sequences is followed by one (boolean) or two *next variable* and/or *redundant variable* symbols encoded as a single bit. Similar to a redundant relation, the redundant variable bit is followed by an address, which has a predefined maximum size, that references a variable on the variable stack.

However, there is one optimization that leverages the fact that most relations are equality checks with two next variable symbols. In order to further compress this standard case, a single bit in front of each relation bit sequence signals if the relation is an equality check on the two next variables on the variable stack. If this is the case, the relation and variable bit sequences are omitted. If the bit indicates another kind of relation, it is encoded as before. Thus, this optimization trades off a smaller standard case for a larger special case.

The end of the relation stack and the beginning of the variable stack is determined by the number of relations, which can be derived from the formula stack. Hence, an explicit encoding of the relation stack length can be omitted.

3.2.3. Variable Stack

The variable stack uses a bit sequence to encode the variable type followed by a type-dependent representation of the variable value. The possible types are defined in the policy definition. A typical minimal set for variable types would be {boolean values, 64 bit integers, 64 bit floats, strings, enumerations, functions, keys}. The compression could benefit from the extension of integers and floats with different bit-lengths, as the annotation size decreases in this case if many numbers are representable with considerably less number of bits compared to the largest appearing number in an expectation formula.

With the standard set presented above, 3 bits suffice to distinguish between the 7 variable types. The standard values are encoded as follows (encoding for new variable types can be easily

deduced): Boolean values are encoded as a single bit. Similarly, integers and floats are encoded with their respective bit size. Strings are encoded as null-terminated ASCII values. The actual compression at the variable stack takes place for keys and enumerations. First, keys are encoded as numbers as given by the order of appearance in the policy definition. The number of bits required for this identifier is determined by the number of keys in the policy definition. Similarly, the value of an enumeration is given by the number in the sorted list of possible values for this enumeration. The enumeration type can be derived from the other variable of the current relation, which needs to be a key. Finally, functions are encoded by a number that references the actual function in the policy definition. As the policy definition specifies the parameters of the function, the parameters can be encoded directly, omitting an encoding of the variable type.

3.3. Data Center Offering

Listing 3.3: Data center offering example.

```
{
    "variables": [
        { "storage.provider": "CompanyA" },
        { "storage.log_access": true },
        { "storage.deleteAfter": true },
        { "storage.notify": true },
        { "storage.backupHistory": true },
        { "storage.location": "DE" },
        { "storage.fde.activated": true },
        { "storage.fde.algo": "aes" },
        { "storage.fde.keySize": 256},
        { "storage.replication": 5 },
        { "storage.availability": 0.99 }
    ]
}
```

The data center offering specifies the actual value of a data center for each key given in the underlying policy definition such that the PDP can, given the policy definition, data annotation, and data center offering, decide if the data center fulfills the expectations of a user. Listing 3.3 shows a data center offering example for the policy definition example in Listing 3.2. Similar to the policy definition, data center offerings are encoded in JSON syntax. For each key in the policy definition, it provides the values that match the data center properties and the provider offerings, e.g., it contains the provider name and data center location, lists supported storage encryption methods, level of data replication and availability factor. Furthermore, the offering specifies for each function defined in the policy definition if it is supported. If a function is supported, the data center provides a small script that checks if the function is supported given the parameters specified by the expectation. These scripts are provided in addition to the data

center offering and enable the PDP to evaluate the function with the given parameters in the data annotation.

The offerings can be compressed with standard (JSON) compression methods, and using the enumeration compression described in Section 3.2. However, in principle the data center offerings only need to be exchanged between data centers and the PDPs in the case of changes. Therefore, compression of data center offerings is less important than compression of user expectations as data annotations.

3.4. Matching

For the matching, i.e., the decision if a data center supports the expectations annotated to a data item, the PDP replaces the keys in the data annotation with the values given by the data center offering. Furthermore, the parameters for functions are extracted from the data annotation and fed into the provided function implementation. Finally, the PDP checks if the boolean formula is satisfied.

In the matching process, the PDP can apply the logical operations in the order given by the formula stack as the reverse polish notation eliminated bracket based evaluation order. Furthermore, the boolean result of each relation is stored for future referencing. If the relation occurs again, i.e., the formula stack indicates a redundant relation, the relation does not need to be evaluated again. Instead, the PDP reuses the already computed result.

3.5. Extensibility

The variable types and relation types supported by our privacy policy language are limited, but sufficient for application in the cloud scenario today. Nevertheless, we designed the privacy policy language such that its grammar is easily extensible in order to extend the expressiveness, provide more concise representations, or improving the compressibility. More specifically, it is easy to add new variable types and define new relation types, e.g., relations based on a similarity score instead of exact equality. However, these additions also affect the presented compression to data annotations, as additional variable or relation types may require more bits to distinguish between them when encoding the relation and variable stack. Hence, the application of those modifications requires a detailed evaluation of its cost-benefit ratio, which is a part of future work.

4. Implementation concepts

In the following sections, we discuss the technical implications of employing the *SSICLOPS policy language* discussed in Chapter 3 by example of the use case scenario illustrated in Figure 4. The scenario includes numerous users, where each user requests an instance of the *Hyrise-R* in-memory database in a *Platform as a Service* (PaaS) like manner. However, users impose certain requirements regarding attributes ranging from the coarse-grained properties such as data center location to fine-grained requirements like database configuration parameters. The *policy decision point* (PDP) acts as the main entry point for users requests. While Figure 7 depicts the *PDP* as a centralized component, its actual implementation strategy might vary. Based on the policies specified by a user, the *PDP* routes requests through a series of *policy enforcement points* (PEP) in order to comply with the respective policies. With the *SSICLOPS policy language* at hand, users can impose requirements on service providers by annotating their requests accordingly. On the coarse level, requirements such as geolocation or *Quality-of-Service* (QoS) might be expressed, whereas the fine-grained level can be used to specify application specific demands like availability properties or user rights.

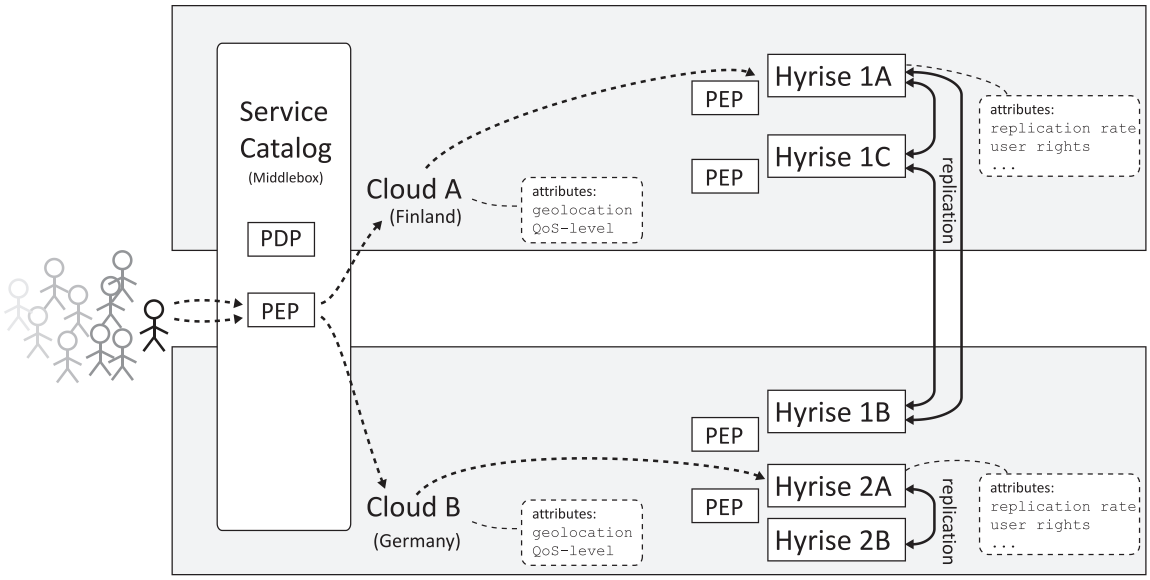

Figure 4: Use case scenario: Users request instances of the *Hyrise-R* in-memory database and annotate their requests with certain policy demands. The *policy decision point* (PDP) acts as the initial entry point and routes requests through a series of *policy enforcement points* (PEP) to process the requests accordingly.

The scenario demonstrates that multiple components have to cooperate in order to consider policy requirements on all levels. In Section 4.1, we discuss the technical facilities within *Hyrise-R* to achieve different levels of availability. To illustrate this, we provide different *QoS*-levels of the database service and leverage internal mechanisms of the database (k-safety, replication) to satisfy the associated availability policies.

In Section 4.2, we consider requirements for certain availability policies as well as restrictions on the geographic locality of the services provided in the context of *OpenStack*. Furthermore, we propose a framework that allows us to evaluate the dependability of policy enforcement mechanisms. Using nested virtualization, a federated *OpenStack* setup comprising multiple regions provides the means for performing fault-injection experiments and performance evaluations in a repeatable environment.

4.1. Policy Language Implementation in Hyrise-R

This section discusses the *SSICLOPS policy language* implementation in the scope of the database cluster *Hyrise-R*. It starts with an introduction of the in-memory database *Hyrise* and describes its main-delta architecture and flexible table layout. We have implemented a scale-out extension of *Hyrise*, called *Hyrise-R*. After giving an overview of the architecture and key concepts, we discuss policy features that are supported by the design of Hyrise-R.

4.1.1. In-Memory Database Hyrise

Hyrise is an in-memory research database, implementing a main delta architecture like SAP HANA [16]. Tuples in the main partition are stored dictionary compressed with a sorted dictionary to support efficient vector operations and optimized range queries. New tuples are inserted in the write-optimized delta partition with an unsorted dictionary as trade-off for better write and reasonable read performance. The periodic merge process moves tuples from the delta to the main partition [11]. *Hyrise* supports a flexible hybrid table layout to store attributes corresponding to their access patterns [4, 5]. A columnar arrangement is well-suited for attributes which are often accessed sequentially, e.g., via scans, joins, or aggregations. Attributes accessed in OLTP-style queries, e.g., projections of few tuples, can be stored in a row-wise manner. *Hyrise* exploits an insert only approach and multi-version concurrency control with snapshot isolation as default isolation level [18].

4.1.2. Database Cluster Extension Hyrise-R

We extended *Hyrise* with the capability to form a database cluster. The extension is called *Hyrise-R* [19] and the implemented distribution approach can be classified as lazy master replication [3]. A query dispatcher is the user's access point for submitting database requests and

propagates the queries to appropriate cluster nodes. A single database instance, called master node, is responsible for transaction handling. The master node sends its log messages, describing the physical data changes, to the other cluster instances, called replicas. The replica nodes update the data with the log information to keep in sync with the master node. To detect node failures, the master node sends heartbeat requests to the replicas.

4.1.3. Policy Features in Hyrise-R

Our goal is to employ the *SSICLOPS policy language* for *Hyrise-R*. The user will not only be able to store and query data in *Hyrise-R* but also to describe policy properties. However, replication, i.e., storing the same data on each node of the database cluster, does not support all policy features. Given a *Hyrise-R* cluster spread over multiple instances, the user can only store the data on all or none of them. This subsection covers selected policy properties, supported by the design of *Hyrise-R*.

We implemented *Hyrise-R* for read scalability and availability. The number of database cluster nodes can be increased to scale the read throughput. The dispatcher will distribute incoming reads among all cluster nodes. Besides scalability, a higher number of *Hyrise* instances in the cluster increases availability. We will implement K-safety for *Hyrise*. K reflects the number of replicas in the cluster. These replicas can take over the role of the master node in case of a node failure. This requires a failover mechanism and an approach replicating data changes before transactions commit.

The geolocation of cluster instance, i.e., the identification of the real-world geographic location of the computer running the database, is a further policy property a user or *PDP* may want to control in database cloud scenarios. However, the desired policy properties may conflict. On the one hand legal requirements may forbid distributing data and storing it in specific countries. On the other hand companies may want to store their data in different clouds or geolocations to increase availability or decrease latencies. The dispatcher can propagate database requests to instances located close to the user to reduce response latency.

Policy languages can describe access control and rights management. We distinguish database users and their privileges. Privileges can be classified into object and system priviledges. Object priviledges specify end users' rights on database objects, i.e., tables, indices and procedures. They describe for example which SQL operations, e.g., select, insert, update, delete, create, alter, drop, the user is allowed to execute and grant to others and whether he can debug database operations. System priviledges concern the administration of the database, e.g., logging, backups, user management.

4.2. Policy-aware OpenStack cloud federations

As cloud computing becomes more and more popular, there is an increasing number of implementations to offer various cloud service models like infrastructure as a service (IaaS), platform as a service (PaaS) or software as a service (SaaS). While many companies offer commercial solutions like the Amazon Elastic Compute Cloud (EC2) or HP Helion, there are also open source alternatives that can be freely installed and configured to meet the needs of one's projects with respect to the underlying hardware available.

The *OpenStack* project offers a cloud software stack which allows for offering infrastructure as a service, almost independent of the underlying hardware setup. The project itself can be seen as a collection of services that can be configured according to the specifications of the planned use cases. Central components that *OpenStack* is comprised of are networking, virtualization and storage services. Furthermore, it is possible to add further components to an *OpenStack* installation, e.g., services that handle billing or allow for object storage in the cloud.

When offering a service in cloud computing, it is crucial that customers can rely on service properties such as security mechanisms and dependability. A service should be highly available, meaning that the system should be continuously operational without failing. Furthermore, all policies that the service provider and the consumer agreed upon have to be adhered to. Therefore, our main goal is to integrate a basic set of policy attributes into the *OpenStack* ecosystem. To do this in a controlled environment, we chose to manually set up a clean single-node *OpenStack* environment (i.e., none provided by a third party like HP Helion) on which we would be able to run the specific analyses. In our recent efforts [2], we turned this manual installation into an automated process in order to simplify and speed up the process of setting up a working *OpenStack* test environment and making the resulting analyses repeatable. Since no *OpenStack* installation is exactly the same, the repeatability of the results of such analyses is not an easy feat. We tackle this issue by making the test environment for the experiments completely virtual. Thus, we circumvent tedious hardware setup and hardware errors that disturb the experiments.

Within a single *OpenStack* instance, mechanisms such as service replication can be used to ensure that certain availability requirements are met. In setups where multiple instances of *OpenStack* are interconnected in order to form a federated cloud, improved availability properties can be implemented. In single region setups, the coarse grained choice of the region is sufficient in order to adhere to geolocation policies. For federated setups that span across multiple countries however, complying with geolocation attributes requires more fine-grained mechanisms that enable individual requests to be processed in the proper location.

At its current state, our virtualized testbed [2] can automatically create a setup consisting of a single *OpenStack* instance. However, our goal is to study federated *OpenStack* environments, that are comprised of multiple *OpenStack* instances. An overview of the available methods for creating such federated setups based on *OpenStack* are presented in Section 4.2.1. In this Section, we also discuss potential approaches for integrating a subset of the *SSICLOPS policy language* into the *OpenStack* ecosystem. Finally, Section 4.2.2 documents our ongoing efforts in

extending our single-instance setup to a federated environment. In this Section, we also propose the application of fault injection mechanisms in order to validate the correct behavior of the policy enforcement mechanisms we intend to provide for the *OpenStack* ecosystem. In addition to test cases that investigate the adherence to policy attributes, the virtualized testbed provides the means for evaluating non functional parameters such as performance metrics.

4.2.1. Review of Federation Mechanisms

At the time of writing, the *OpenStack Architecture Design Guide* [15] differentiates between two approaches for interconnecting multiple *OpenStack* instances in order to form a federated setup. In this section, the architecture of each approach is presented and opportunities for integrating policy enforcement mechanisms are discussed.

4.2.1.1. Cloud Management Platforms

The approach based on *Cloud Management Platforms* (CMPs) assumes mostly unaltered *OpenStack* instances, which are coordinated by a broker-like entity, the so-called *Cloud Management Platform* (see Figure 5). The main advantage of *CMPs* is that they require very few to no alterations of existing *OpenStack* instances. This property can be traced back to one of the main goals of *CMPs*, which is to abstract from the underlying cloud platform in order to support hybrid cloud setups and cloud bursting scenarios. While this abstraction may be beneficial for cloud-bursting scenarios in hybrid setups, it might oppose further intertwining among *OpenStack* instances in federated setups. At the same time, replicating all management facilities in each instance introduces additional overhead. Finally, many *CMPs* are proprietary products of public cloud providers that can not be used for self-hosted use cases. However, with Scalr [17] and ManageIQ [13], there are open source based projects that can be customized.

Regarding opportunities for integrating the *SSICLOPS policy language* concepts into federated cloud setups, *CMPs* are an oncoming target, since *CMPs* have a similar role compared to *PDPs*. Furthermore, only the *CMP* itself has to be altered. However, the main drawback is that the *CMP* represents a single point of failure. As soon as the *CMP* enters a degraded operational state, the proper enforcement of policies is at stake.

4.2.1.2. Multi-Site OpenStack Instances

Providing an alternative approach, multi-site *OpenStack* setups consist of multiple *OpenStack* instances, that share a certain set of common services. The *OpenStack* reference design providing location-local services through multi-site installations is illustrated in Figure 6. However, it should be noted that federated setups with different goals can use a very similar setup that only shares the *Keystone* authentication service and does not require a load balancer. When multiple

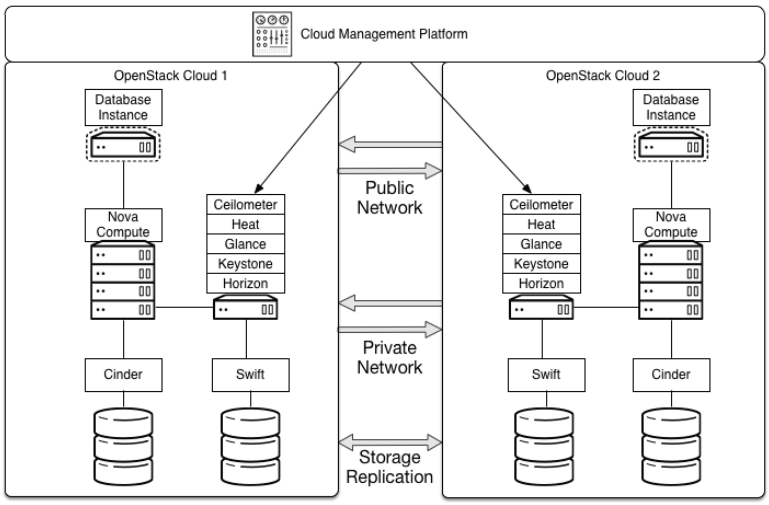

Figure 5: Very few to no alterations to *OpenStack* are required when a *Cloud Management Platform* (CMP) is used to form a federation of multiple instances. Source: [15]

sites are interconnected, *OpenStack* differentiates between four roles that an instance can embody: *Cells*, *Regions*, *Availability Zones* and *Host Aggregates*.

Cells provide the most basic mechanism for organizing a cloud installation in a distributed mode of operation without the need for additional technologies such as *CMPs* or without having to alter existing nova services. *Cells* provide a hierarchical, tree-based structure for partitioning multiple hosts in a cloud setup.

While *Regions* provide similar semantics and are intended for organizing hosts in partitions, the main difference is that *Cells* only expose the API for provisioning compute resources at the top-level *Cell*. In contrast to this, each *Region* exposes its own compute resource API, which provides users with a more explicit mechanism for deciding which region should be used for allocating compute resources.

Both *Cells* and *Regions* provide interesting means for implementing geolocation policy attributes. Using a single entry point at the *Cell* level would be compelling, as the evaluation of policy attributes could be woven into the *OpenStack* nova-scheduler component. However, performing invasive alterations on a quick-moving target such as *OpenStack* comes with a high risk of failure. As a result, using the mechanism of *Regions* seems to be feasible, since the explicit control over regions should make the implementation of a decoupled *PEP* feasible.

Availability Zones can be used to organize *OpenStack* resources in groups that offer a certain degree of physical isolation and/or redundancy from other *Availability Zones*. Providing some examples, the feature can be used to distinguish resources that are connected to a different *Power Distribution Unit* (PDU) on the fine-grained level, or machines that are located in the nearby failover data center. In contrast to this, *Host Aggregates* provide an additional mean for specifying

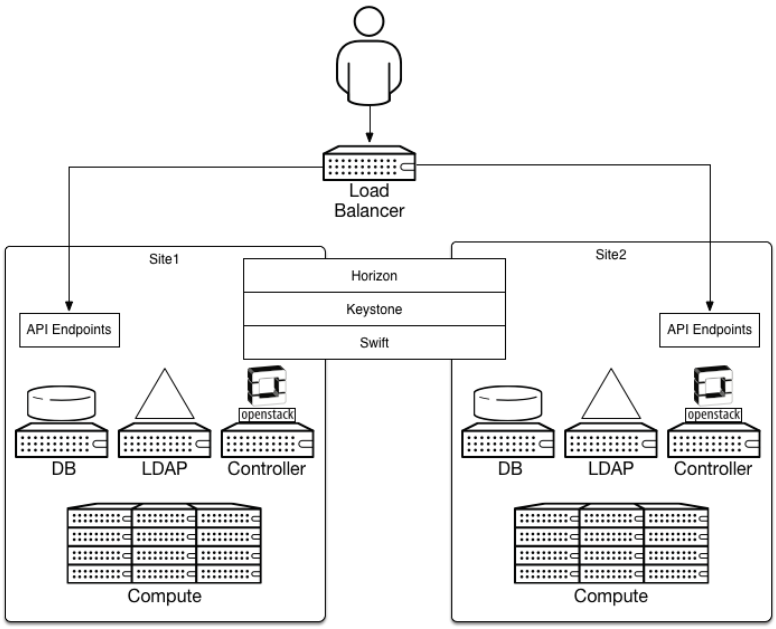

Figure 6: A location-local multi-site setup is illustrated. However, there are many ways to configure a multi-site setup. Many *OpenStack*-components can be shared across sites, however at least the *Keystone* authentication service has to share the same database. Source: [15]

resource domains for load balancing. A popular use case is to use *Host Aggregates* in order to distinguish between different classes of hardware (e.g. processor speed, network link speed, special hardware such as GPUs or FPGAs).

Using the scopes of *Availability Zones* and *Host Aggregates* in multi-site *OpenStack* installations can be leveraged in the implementation of *PEPs* that consider policy attributes such as *QoS* levels or availability parameters.

4.2.2. Single-node experimental platform for federated OpenStack setups

In this Section, we describe the automated installation process of our OpenStack-based testbed in our virtual environment. We give an introduction to the usage and a conceptual overview. Furthermore, we describe mechanisms for implementing test cases that evaluate the correct behavior of policy enforcement mechanisms using fault injection.

The automated scripts for setting up the testbed are developed and tested using *Ubuntu 14.04.3 LTS (Trusty Tahr)* server version. All required dependencies (e.g., *Ansible*, *libvirt*, etc.) are installed automatically, thus only an Internet connection is required. The complete installation can be started by running a simple script, which creates a federated *OpenStack* setup according to the architecture depicted in Figure 7. On the physical machine (layer Φ), virtual machines are created that represent an *OpenStack Region* or a datacenter location (layer Δ). Within the Δ-layer,

further virtual machines are created that host basic *OpenStack* services (layer Ω). It should be noted, that each service runs on a separate virtual machine, which emulates the behavior of a data center, where the services run on individual machines as well. This is also a unique characteristic compared to other single-node *OpenStack* installations such as DevStack [14] or HP Helion [8]. The ι-layer represents virtual machines that are provided to users by *OpenStack*. Finally, applications such as *Hyrise-R* can be run on these end-user VMs on the α-layer. All virtual machines up to the Ω-layer are automatically created in the course of the automated testbed initialization.

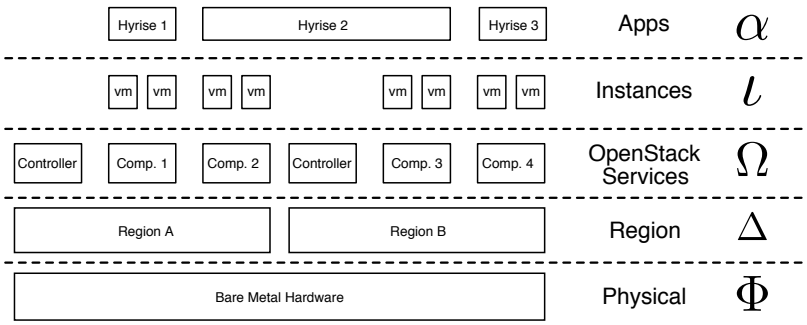

Figure 7: The virtualized *OpenStack*-testbed uses nested virtualization to resemble multiple *Regions* (Δ), individual servers (Ω) within a *Region* that host *OpenStack*-components, and VMs (ι) and applications (α) provisioned in the federated setup. Except for the α-layer, box sizes represent available memory and compute resources.

The implementation of test cases follows a fixed structure, so that it is easy to add further test cases if necessary. The general idea is that the injection of faults can provide insights on how they affect the functionality of policy enforcement mechanisms and how the implementation of such mechanisms can deal with faults. This serves partly to show weak points of the setup and partly to document details about its behavior. Experiments consist of multiple stages: The *setup* stage creates all elements necessary to run the experiment. The *break* stage then injects a fault. An optional *heal* stage can be implemented to remove the fault. After each of these stages, a *check* step is executed, which observes the state of the system and reports its findings to the user. Generally, after the setup stage all checks should be successful. In cases where automated testing is not feasible and user interaction is required, the user is prompted accordingly.

5. Conclusions

Today, users lack a mechanism to express their own expectations on the handling of cloud data to cloud providers. Policy languages can be used to express such expectations [6, 7]. However, existing policy language are not designed for the cloud scenario and consequently do not address all requirements for a policy language in this setting [1]. We showed the design of the *SSICLOPS policy language*, which is a promising policy language that is specifically tailored for the requirements of the cloud setting. It provides a representation that enables users to read and understand the represented expectations. However, this representation is large and would impose considerable network overhead for cloud providers, which may limit deployability. To tackle this problem, a domain specific compression nevertheless allows users to attach the compressed expectations as annotations to the data such that they are available during the full life-cycle of data in the cloud.

In the future, we will implement the presented design of the *SSICLOPS policy language* and evaluate its application in the federated cloud scenario. Hyrise-R will be one exemplary application to exploit the presented language to configure policy characteristics, e.g., the replication rate of the stored data. At a lower level of the cloud application stack, we are going to investigate approaches for integrating policy attributes regarding geographical location and *Quality-of-Service* levels into the *OpenStack* ecosystem. To support this process, we will provide a fully virtualized testbed based on OpenStack, which ensure repeatability for both test cases and performance measurements. Furthermore, applying fault injection in test cases will allow us to harden policy enforcement mechanisms even if a degraded system state has been reached. To achieve these goals, we are going to build up on top of our preceeding efforts in the project, where we introduced replication mechanisms to Hyrise [12] and presented a virtualized single-site *OpenStack* testbed that enables us to perform fault injection experiments [2].

Bibliography

[1] F. Eberhardt, M. Plauth, A. Polze, S. Klauck, M. Uflacker, J. Hiller, O. Hohlfeld, and K. Wehrle. *D2.1: Report on Body of Knowledge in Secure Cloud Data Storage*. Tech. rep. June 2015.

[2] J. Eschrig, S. Knebel, and N. Kunzmann. "Dependable Cloud Computing with OpenStack". *Proc. Third HPI Cloud Symposium "Operating the Cloud" 2015*. 2015.

[3] J. Gray, P. Helland, P. O'Neil, and D. Shasha. "The Dangers of Replication and a Solution". *Proc. 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. 1996.

[4] M. Grund, P. Cudre-Mauroux, J. Krüger, S. Madden, and H. Plattner. "An overview of HYRISE - a Main Memory Hybrid Storage Engine". *IEEE Data Engineering Bulletin* (2012).

[5] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. "HYRISE: A Main Memory Hybrid Storage Engine". *Proc. VLDB Endow.* (2010).

[6] M. Henze, M. Grossfengels, M. Koprowski, and K. Wehrle. "Towards Data Handling Requirements-Aware Cloud Computing". *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*. Vol. 2. Dec. 2013, pp. 266–269.

[7] M. Henze, R. Hummen, and K. Wehrle. "The Cloud Needs Cross-Layer Data Handling Annotations". *Security and Privacy Workshops (SPW), 2013 IEEE*. May 2013, pp. 18–22.

[8] Hewlett Packard Enterprise. *HPE Helion OpenStack*. http://www8.hp.com/us/en/cloud/hphelion-openstack.html. Accessed: 2016-01-14.

[9] Intel IT Center. *Peer Research: What's Holding Back the Cloud?* Tech. rep. 2012.

[10] I. Ion, N. Sachdeva, P. Kumaraguru, and S. Čapkun. "Home is Safer Than the Cloud!: Privacy Concerns for Consumer Cloud Storage". *Proc. Seventh Symposium on Usable Privacy and Security*. SOUPS '11. ACM, 2011, 13:1–13:20. DOI: 10.1145/2078827.2078845.

[11] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. "Fast Updates on Read-optimized Databases Using Multi-core CPUs". *Proc. VLDB Endow.* (2011).

[12] J. Lindemann, S. Klauck, and D. Schwalb. "A Scalable Query Dispatcher for Hyrise-R". (to appear). 2015.

[13] *ManageIQ*. http://manageiq.org. Accessed: 2016-01-14.

[14] OpenStack Project. *DevStack - an OpenStack Community Production*. http://docs.openstack.org/developer/devstack/. Accessed: 2016-01-14.

[15] OpenStack Project. *OpenStack Architecture Design Guide*. http://docs.openstack.org/arch-design/. Accessed: 2016-01-12.

[16] H. Plattner. "A Common Database Approach for OLTP and OLAP Using an In-memory Column Database". *SIGMOD* (2009).

[17] *Scalr*. http://www.scalr.com. Accessed: 2016-01-14.

[18] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner. "Efficient Transaction Processing for Hyrise in Mixed Workload Environments". *IMDM in conjunction with VLDB*. 2014.

[19] D. Schwalb, J. Kossmann, M. Faust, S. Klauck, M. Uflacker, and H. Plattner. "Hyrise-R: Scale-out and Hot-Standby Through Lazy Master Replication for Enterprise Applications". *Proc. 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. 2015.