LMS: Label Management Service for Intent-driven Cloud Management

Joon-Myung Kang
Hewlett Packard Labs
Palo Alto, CA, USA
joon-myung.kang@hpe.com

Jeongkeun Lee*
Barefoot Networks
Palo Alto, CA, USA
jeongkeun.lee@
barefootnetworks.com

Vasudevan Nagendra*
Computer Science Department
Stony Brook University, NY, USA
vnagendra@cs.stonybrook.edu

Sujata Banerjee*
VMware
Palo Alto, CA, USA
sujatab@vmware.com

***Abstract*—Today's cloud infrastructure is often overwhelmed by inputs from multiple users and administrators for enforcing the policies on which to run cloud services, and infrastructure administrators need to configure policies on different resource types such as compute, network, etc. Such complex policy enforcement decisions from multiple users could result in errors and conflicts. To mitigate such complexities in managing cloud infrastructure, there is a strong push towards decoupling high level intents ("what" should be done) from the underlying infrastructure implementations ("how" to do it). Unlike existing solutions which resolve conflicts at low level during run-time, intent-based systems aim to resolve potential conflicts at the intent specification level. To efficiently handle large scale cloud environments, we propose a Label Management Service (LMS) which provides meaningful abstractions and their relationships by analyzing target cloud infrastructure. It helps the cloud administrators to model their policy requirements efficiently by decoupling the intents from underlying specifics. LMS scales to large dynamic cloud environments and manages the life cycle of label-based intent and enforcement.**

I. Introduction

Today's cloud environments are unbelievably complex provisioning millions of compute nodes 'or'virtual machines (VMs) to users on a day-to-day basis [1]. For each of the compute nodes, hundreds of operational and policy related attributes are maintained in the cloud, leading to billions of such attributes maintained at complete data center level. Managing the cloud environments in the presence of such large number of operational attributes is a challenging task. Also, the dynamicity in the cloud environment due to the mobility of virtual machines (VM) and the resource scaling further complicates the management of cloud infrastructure.

Therefore, to effectively reduce the management complexity, the cloud management systems (CMSs) are effectively devising abstraction techniques to hide the low level details (i.e., IP, MAC, memory, CPU, security status, mobility of VMs and so on) from the cloud users or administrators. It is currently believed that the intent-based cloud management systems will be able to alleviate the pain of the users in administering the cloud environments by separating out "what" to do from "how" to do it [2]–[5]. This necessitates a need to provide an interface that can naturally express the human intents and systemically translates them down to the infrastructure rules. Also, it is expected that SDN (Software Defined Networks) and NFV (Network Functions Virtualization) based applications used in the cloud infrastructure highlights the need of such intent interface [2], [6], [7] for efficiently managing the cloud networks.

Therefore, ideally an intent interface must be simple and intuitive enough to accommodate the users intentions effectively, and portable by decoupling itself from low level infrastructure details [8]. Also, this decoupling is important to let the same intent specifications to be reused for deployment of same services/policies on multiple different cloud sites. Currently, these intents are designed as "subject-verb-object" after human language or as graph node indicating group of entities (VMs, network ports, etc.) over which the intents are targeted.

To decouple the intents from underlying specifics and to make our approach scalable, such a group of entities can be defined with logical *labels*, representing diverse attributes of entities, instead of listing all entity IDs (or specifics) in the group. That is, the labels are directly used in the context of intent specification to capture the properties of the infrastructure elements in the cloud. These labels are simple key-value pair designed to capture the basic properties (and their values) of the cloud entities e.g. compute (memory, cpu), network (interface, speed), security (patched, infected) and so on. In addition, values may dynamically change over the life cycle of an entity. For example, CPU utilization or memory consumption is dynamic. The existing label based frameworks fall short in satisfying the intent-based cloud management system as they lack the capability to capture the complex relation that the cloud infrastructure possesses. Hence, there is a strong need to enhance the capabilities of label management framework to support the scalable cloud infrastructure.

In this paper, we present a Label Management Service (LMS) that,

- Defines a label naming syntax and namespace,
- Automatically constructs the label namespace by extracting entity attributes from various cloud data sources,
- Captures relationship between labels for enabling proactive analysis and composition of label-based intents and
- Dynamically maps the entities with entity groups satisfying their membership predicates defined by Boolean operation

*Work done while at Hewlett Packard Labs

978-3-901882-89-0 @2017 IFIP

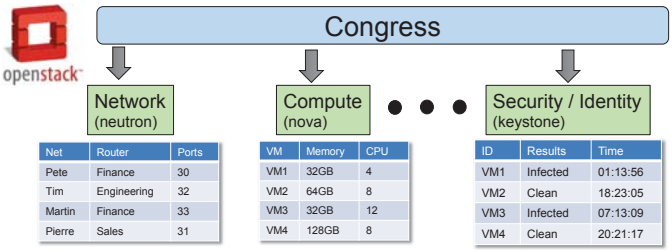


Fig. 1: Congress: Policy management service for OpenStack [12].

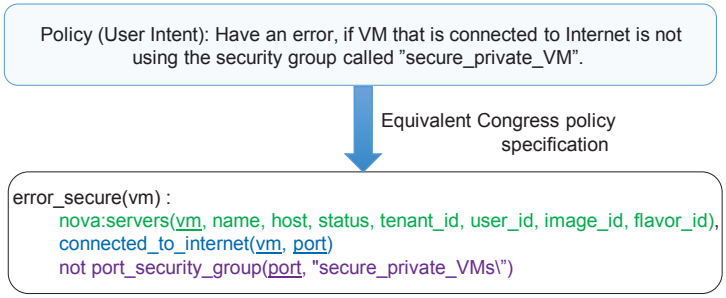


Fig. 2: User intent and equivalent Congress policy specification

of labels.

The LMS architecture is built to handle the complex cloud infrastructure and capture the relation between different entities inside the cloud using an hierarchical label-tree based architecture, while the existing Docker/Kubernetes based infrastructures use simple key-value pair to specify the reusable policies [9], [10]. To illustrate the benefits of our framework, we have implemented the LMS on top of OpenStack [11] and Congress [12] modules to build a label namespace and enforce high-level policy intents to VMs (entities). We estimate the time required to build the label name space and label-entity mapping and evaluate its memory consumption on the data set with thousands of entities and entity groups. We show that our LMS sends similar time for mapping entities over different sizes of label trees.

In the rest of the paper, we present the motivation behind the label-based cloud management system in §II and §III, and the details of LMS architecture and techniques used to build the label namespace using the cloud data sources in §IV. Then we present the preliminary performance results in §V, followed by discussion on the state-of-the-art work related to LMS in §VI.

II. Background

In this section, we describe the Cloud Management System (OpenStack) and the specific policy engine used (Congress). Later, we introduce the terms used in this paper, followed by the notions of label and label trees.

A. High level details of Cloud Management System

The Cloud Management System (e.g. OpenStack) provides a mechanism to manage the cloud services based on policies specified by the user/administrator. In OpenStack, Congress [12] is a policy enforcement framework that actively interfaces with diverse cloud services (e.g., application, network, compute and storage), for enforcing the policy intents specified by the user. Congress uses a variant of Datalog [13] as a policy language to specify the policies. Figure 1 shows examples of data sources available in OpenStack that interfaces with cloud services [14] (e.g. Neutron, Nova, etc). Congress continuously monitors to check if the system abides the policies implemented by users. Figure 2 represents an example policy of Datalog, the original policy states to report an error if the VM is connected to Internet without being a part of the existing security group `secure_private_VM`.

In this paper, we use Congress for managing OpenStack-based cloud and generate Datalog from simple label and label tree definition *automatically*. However, LMS is not limited to Congress and Datalog but applicable to any kind of CMSs.

B. Terminology

We use the following terms through out this paper:

- **Entity:** A smallest unit for policy enforcement (e.g., VM, container, network port, device).
- **Entity Group (EG):** A set of entities that satisfy a membership predicate (Boolean expression) defined over the labels. For example, an EG defined as "Tenant:Tom & Location:Zone-1" meant as a set of VMs owned by tenant 'Tom'and placed in location 'Zone-1'if an endpoint is a VM.
- **Label:** A membership predicate, a Boolean valued function representing an entity attribute (owner, placement location, security status, etc.). An attribute can either be static (e.g., VM ownership) or dynamic (e.g., security status).
- **Label Tree:** Captures the hierarchical relations between labels of the same kind (e.g. Location tree, Tenant tree, etc.)
- **Label Tree Definition:** Represents a metadata which define parent-child relationship between labels which is used for building a label tree.
- **Label Namespace:** Complete set of label trees that captures the relations between all the entities of same kind inside the cloud infrastructure.

C. Labels and Label Trees

A more detailed view of label and label tree is presented.
Label: Similar to label definitions used in Docker [15] and Kubernetes [16] infrastructures, we represent the label as a combination of key and value pair to represent any specific entity inside the cloud infrastructure i.e., in (`key:value`), a `key` defines the type of the entity attribute and `value` defines the actual value (dynamic or static) the attribute carries. For example, a key and value pair used to represent the location of the hosts in the cloud infrastructure is represented as "Location:AZ1". In this example, the *Location* attribute represents the key and *AZ1* represents the value that *Location* attribute carries.
Label Tree: In addition to existing capabilities, we introduce "label trees" for effectively capturing the hierarchical relations between labels (values) of the same kind. For example, the complex relation between different locations and the hosts/servers present in that location is captured using following "*Location*" based label tree, which in our case is automatically derived from the cloud data sources using LMS.

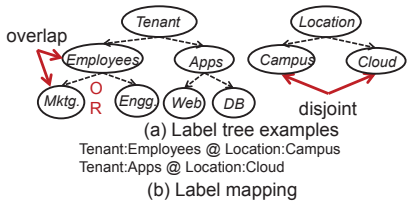


Fig. 3: Automatically generated label tree (Location) using LMS.

Figure 3 illustrates an example of label tree for *Location* which LMS automatically generates from a cloud cloud management service (Openstack) with two availability zones and four hosts. *Host1* and *Host2* are located in AZ1, *Host3* and *Host4* are located in AZ2. Following are the definitions of sub-components of a label tree.

- **Root node:** Root node of the tree is considered as the `key` attribute, which defines the name/identity of the label tree. A name of root node is unique in the label namespace. *Location* in Figure 3 represents the root node of the hosts location tree..
- **Non-root node:** are `value` attributes of the tree root. All nodes except a root node (*Location* in Figure 3) are non-root nodes. Non-root nodes are categorized into two sub types:
  - **Leaf node:** Leaf nodes are basic elements and membership predicates assigned to entities of the cloud infrastructure. *Host1*, *Host2*, *Host3*, and *Host4* are leaf nodes.
  - **Non-leaf node:** A non-leaf node is a composite predicate, i.e., "Boolean OR", of all of its descendant leaf labels. For example, *AZ1* is a composite predicate which encompasses entities of *Host1* and *Host2*.
- **Overlap:** A pair of labels with parent-child relationship can overlap. From Figure 3, *AZ1* and *Host1* has parent-child relationship and are overlapped because entities of *Host1* is a subset of entities of *AZ1*. If there is any policy intent for *AZ1*, it must be applied to *Host1* even.
- **Disjoint:** Any labels that do not have a parent-child relationship (i.e. sibling labels) are mutually exclusive, cannot be simultaneously "true" for any entity i.e., *AZ1* and *AZ2* in *Location* label tree are mutually exclusive and it means no child entity (e.g. *Host3* can exist in simultaneously in multiple parent entities (i.e., both in *AZ1* and *AZ2*).

III. Motivation

In this section, we discuss the limitations of existing infrastructure and the motivation for developing an LMS to handle the intent-based cloud management system. This paper focuses on the following three important aspects for effectively managing the cloud, by proposing label based management system to dynamically handle the cloud infrastructure at scale

Label Management: *Labels must be systemically defined and managed by extracting relevant information from various data sources*. The use of labels in portable policy specification is analogous to using variables/parameters to write portable computer programs, instead of using fixed constant values. As variables and variable namespace need to be clearly defined to correctly write and execute a program in any computer languages, the policy labels and label namespace should be managed. Otherwise, every user may create arbitrary labels without clear rules, and the randomly generated labels can easily overlap/conflict and create another chaos and management problems. In often cases, diverse entity attributes are defined/assigned/managed outside of the policy management system, e.g., each compute/storage/network resource controller or a separate security monitoring service. Manually creating each label for each and every entity attribute is not scalable.

Proactive Policy Specification and Analysis: *There must be an automated system that extracts such relations from existing data sources and constructs label tree and generates a mapping data structure, since doing so manually would be error-prone and a huge burden for human operators.* By using logical labels instead of specific entity identifiers, a policy can be proactively specified without actual entities in the system. However, this makes proactive policy analysis challenging. Analyzing/composing policies and ensuring that individual policy intents are satisfied prior to deployment, i.e., proactively, is highly desirable. Proactive composition can greatly reduce the number of conflicts/errors that a runtime system has to handle, and reduce the chance of system misbehavior compared to lazy runtime composition. In the worst case, proactive composition without actual entities in the system can lead to exponential state explosion because in the worst case, every combination of input policies should be considered. Thus, PGA [4] defines label trees and inter-tree label mappings, which capture [overlap vs. disjoint] relations between labels within a tree or across trees, enabling proactive and scalable composition policies.

Dynamicity and Scalability: *The management system must be scalable as the complexity of label tree construction and mapping can multiplicatively increase up to O(nk) where n is the number of entities and k is the number of labels in the system*. Entities can be assigned with labels dynamically at runtime, causing them capably move from one EG to another. Real-time tracking, management of the entity-to-EG memberships is critical for timely applying the policies to dynamically joining/leaving/changing entities or to detect policy violations.

IV. Solution: Label Management Service

In this section, we discuss how LMS constructs a label namespace automatically and maps entities and EGs. First, we present the LMS architecture and describe how LMS works with intent-based management system (IMS). Later, we present a method to automatically generate the label namespace and describe about how the label namespace can be used for intent description and composition. Finally, we show how LMS dynamically maintains mapping between entities and EGs in terms of scalability with three basic algorithms.

A. Overall Architecture

LMS is interacting with other external systems as shown in Figure 4. LMS provides northbound interfaces for managing

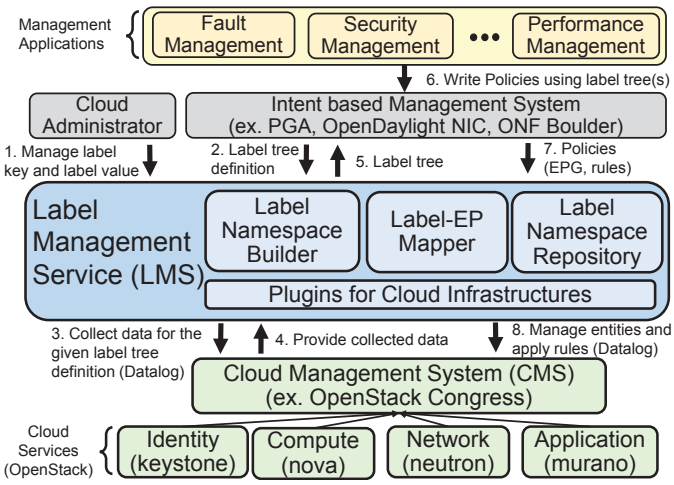


Fig. 4: LMS architecture and overall process

labels, label trees, and EGs to Intent-based cloud management system and southbound interfaces for obtaining cloud system data from a CMS. Similarly, LMS receives label tree definitions from its south-bound IMS and takes intents from users in the northbound applications (Fault Management, Security Management and so on). The Label Namespace Builder extracts system states from the underlying cloud services via CMS and constructs label namespace to present automatically generated labels and label trees which presents meaning abstraction allowing the user to express the policy intents. The Label-EP Mapper takes global intents which could be enforced on to the underlying infrastructure by dynamically tracking the low level entities satisfying the attributes (label predicates).

In the next section, we present how to automatically generate a label namespace and map EG and label in detail.

B. Automatic generation of label namespace

The Congress is considered as a suitable platform for enforcing the policies in the cloud. But, creating policy rules manually for cloud scale infrastructure using Datalog is a highly complex job considering both its scale and dynamicity, which could be easily prone to errors. Hence, this highlights the need for automatic generation of Datalog rules in accordance with the given user intent. As described in Section IV-A, LMS takes two different inputs from cloud administrator (or domain expert) and IMS. Initially, the cloud administrator creates label definitions which describe how to collect the related data from specific data sources. Then, IMS can use the label definition in order to define a label tree without any specific knowledge about the target cloud. For example, cloud administrator knows where to get available hosts from, whereas IMS just wants to create a label tree using the available hosts without knowing the specifics of data sources.

In the following sub-section, we introduce a label definition which the administrator can use to identify the entities, then we present a label tree definition based on the label definition for deriving the relation between the entities.

1) Label definition in EBNF: A label is defined with a key and value pair, which specifies how to capture data from the given data sources under the conditions. Here, we define the grammar for a label key definition in a Extended Backus-Naur Form (EBNF) as shown in Listing 1. We define multiple keys (e.g. Location, User, Tenant) and associate the keys to values.

$$\begin{aligned} \langle \text{label\_key} \rangle &\models \langle \text{keys} \rangle \mid \langle \text{key} \rangle \\ \langle \text{keys} \rangle &\models \langle \text{keys} \rangle, \langle \text{key} \rangle \\ \langle \text{key} \rangle &\models \text{key\_name} \end{aligned}$$

Listing 1: A grammar for label key definition

The grammar in Listing 2 defines a label value. The label value definition is composed of 6 key properties: *NAME, KEY, VALUE, DATA_SOURCE, CONDITION and REF*. (a) *NAME* is considered as identity value name of the label, (b) *KEY* is the name which is used for identifying the label tree, (c) *DATA_SOURCE* specifies the details of cloud data sources (e.g. keystone, nova and data tables provided by these sources) for building the label trees, (d) *VALUE* specifies one of attributes in the data sources, (e) *CONDITION* specifies the necessary conditions to extract the specific results using the given data source tables and attributes specified in *DATA_SOURCE* and (f) *REF* is used to refer to other data sources for deriving values from multiple data sources. For example, when we define the label *AVAILABILITY_ZONE*, we use *nova:hosts(zone)* as a reference data source because the availability zone name is used in that data source as well (as illustrated in Figure 5). Figure 5 shows an example of three different keys and four values using data sources and tables available from OpenStack Congress. In the example for *HOST* key, we can get the all hosts attributes (host name) from the hosts table in the nova data source.

$$\begin{aligned} \langle \text{label\_value} \rangle &\models \langle \text{values} \rangle \mid \langle \text{value} \rangle \\ \langle \text{values} \rangle &\models \langle \text{values} \rangle, \langle \text{value} \rangle \\ \langle \text{value} \rangle &\models \langle \text{name} \rangle \langle \text{key} \rangle \langle \text{ds} \rangle \langle \text{val} \rangle \langle \text{cond} \rangle \langle \text{ref} \rangle \\ \langle \text{name} \rangle &\models \text{NAME: value\_name} \\ \langle \text{key} \rangle &\models \text{KEY: key\_name} \\ \langle \text{ds} \rangle &\models \text{DATA\_SOURCES:} \langle \text{data\_sources} \rangle \\ \langle \text{val} \rangle &\models \text{VALUE:attribute} \\ \langle \text{cond} \rangle &\models \text{CONDITION:} \langle \text{conditions} \rangle \\ \langle \text{ref} \rangle &\models \text{REF:} \langle \text{data\_sources} \rangle \mid \lambda \\ \langle \text{data\_sources} \rangle &\models \langle \text{data\_sources} \rangle, \langle \text{data\_source} \rangle \\ \langle \text{data\_source} \rangle &\models \text{datasource\_name:table\_name}(\langle \text{attrs} \rangle) \\ \langle \text{attrs} \rangle &\models \langle \text{attrs} \rangle, \text{attribute} \mid \text{attribute} \\ \langle \text{conditions} \rangle &\models \langle \text{condition} \rangle \mid \lambda \\ \langle \text{conditions} \rangle &\models \langle \text{conditions} \rangle, \langle \text{condition} \rangle \\ \langle \text{condition} \rangle &\models \text{attribute} \langle \text{operator} \rangle \text{attribute} \\ \langle \text{operator} \rangle &\models == \mid != \mid > \mid < \mid <= \mid >= \end{aligned}$$

Listing 2: A grammar for label value definition

```
KEY: [USER, TENANT, LOCATION]
{ NAME: USER_NAME,
  KEY: USER,  DATA_SOURCE: keystone:users(name),
   VALUE: name, CONDITION: },
{ NAME: GROUP_NAME,
  KEY: USER,  DATA_SOURCE: keystone:groups(name),
   VALUE: name, CONDITION: },
{NAME: TENANT_NAME,
  KEY: TENANT, DATA_SOURCE: keystone:tenants(name),
  VALUE: name, CONDITION: },
{NAME: AVAILABILITY_ZONE,
  KEY: LOCATION, DATA_SOURCE:  nova:availability_zones(zoneName),
  VALUE: zoneName, CONDITION:, REF: nova:hosts(zone) },
{NAME: HOST,
  KEY: LOCATION, DATA_SOURCE: nova:hosts(host_name, service),
  VALUE: host_name, CONDITION: service == "compute“ } ]
```

Fig. 5: Label key-value examples

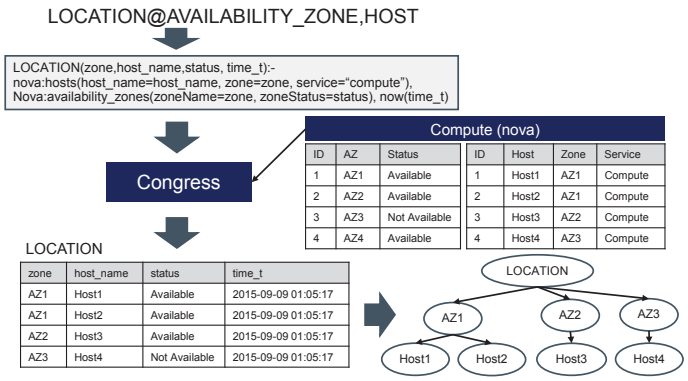


Fig. 6: Example of label namespace construction

2) Label tree definition: After cloud administrator creates label keys and values, we can use them for defining hierarchical order for each label tree. Instead of using low-level details, we can easily specify the label trees using tree definition, which LMS uses to extract the tree attributes by automatically converting the label tree definition in to Datalog. Listing 3 shows an example of label tree definitions for OpenStack.

```
USER:GROUP_NAME,USER_NAME
TENANT:TENANT_NAME
LOCATION:AVAILABILITY_ZONE,HOST
```

Listing 3: An example of label tree definitions

For example, we can define a label tree *LOCATION* using a value *AVAILABILITY_ZONE* and *HOST* and a label tree *USER* using a value *GROUP_NAME* and *USER_NAME* which have a parent-child relationship, respectively.

As shown in Figure 6, LMS generates Datalog using label definitions of *AVAILABILITY_ZONE* and *HOST*. However, our LMS is not limited to the given label tree definition including hierarchical relations. Because each value has its key name in the label definition, the hierarchical relationship (or other types of relationships) between all values which have a same key name can be automatically inferred, but this is not in the scope of this paper.

Figure 6 shows an example of how LMS constructs a label tree for *LOCATION* based on key, value, and label tree definition. As we defined the label tree, it is a hierarchy of availability zone and host. Based on the definition, LMS automatically generates a Datalog as shown in the top of Figure 6 in accordance with the given label relationship definition. In our label definition (key, value), we use all attributes (zone, zoneStatus, time_t) and data sources which Congress is using for Datalog definition. Here, we can directly convert them to Datalog without inference. This is stored to Congress, which keeps the tables up-to-date based on data source changes. Then, Congress starts to monitor the data from the data source and stores the data to *LOCATION* table. Our LMS builds a whole label tree using the currently supported tables maintained by Congress.

C. Label-based intent composition

Based on the generated label trees, we can write our high-level intents using the logical labels instead of using any low-level entity such as VM id, IP, or MAC address. In addition, intents can be expressed in the scope of each writer's interest. From the example label tree in Figure 6, there can be two policy managers: 1) host manager and 2) zone manager. First, a host manager wants to allow only *SSH* and *PING* from one compute node in AZ2 and the other compute node in AZ1. The host manager does not care about zone labels. Second, a zone manager wants to allow *HTTP* traffic from AZ2 to AZ1 regardless any of compute node. Basically, a zone manager wants to assign AZ1 to run web servers accessed by clients in AZ1. We can express these policies using any high-level policy management framework such as PGA [4] using logical labels. Because the label tree has a hierarchical relationship between labels, PGA can use it for resolving possible conflicts and composing the intents Using the label tree in Figure 6, when a zone manager writes a policy *AZ2* $\rightarrow$ *[HTTP]* $\rightarrow$ *AZ1* and wants to compose through LMS, it can be directly normalized to *Host3* $\rightarrow$ *[HTTP]* $\rightarrow$ *Host1* and *Host3* $\rightarrow$ *[HTTP]* $\rightarrow$ *Host2* during the composition. More details of composition process using the labels can be found in our previous work PGA [4], which can use the label trees automatically generated by LMS.

D. Automatic mapping label and cloud resources

In the cloud infrastructure, entities can be assigned by labels dynamically at runtime, causing them to move from one EG to another. For example, a server that was assigned the label *NORMAL* could subsequently be relabeled *QUARANTINED* when a network monitor detects the server issuing a DNS query for a known malicious Internet domain [4]. The runtime system needs to perform the operation of looking up and applying the correct rules for each entity depending on its current EG membership. Current approaches (Group Based Policy (GBP) [5] or other policy framework) have their own entity registry system, which is quite manual. The system typically maintains a set of labels assigned to each and every endpoint, with $O(n)$ complexity, where n is the number of entities in the system. This approach is difficult to implement in existing cloud infrastructures where entity attributes are defined and maintained over multiple distributed services. Another approach would be collecting a set of entities that

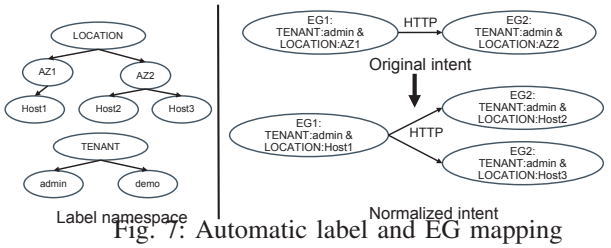


Fig. 7. Automatic label and EG mapping

Algorithm 1 Map entities to EGs (Congress)

```
1: function MAP(Entities, DatalogEGRules)
2:     mapping := ∅
3:     for i := 0 to DatalogEGRules.length do
4:         mapping[i] := ∅            ▷ Initialize a mapping array for all EGs
5:     for Entity ∈ Entities do
6:         for i := 0 to DatalogEGRules.length do
7:             if ISMATCHED(Entity, DatalogEGRules[i]) then   ▷ Congress can provide this function
8:                 mapping[i].add(Entity)
```

satisfies each label predicate (from-Label-to-entity mapping); this is easier to implement in Congress Datalog or other relational database systems but is not convenient to quickly determine a policy (EG) to apply to any given entity.

Our solution is to maintain a list of entities that satisfy given EG definitions, a membership predicate (Boolean expression) defined over the labels. Users can use the auto-generated labels to write their policies or policies defined written the labels in other sources can be imported. Using the overlap and disjoint relations captured in the label trees, individual policies can be automatically and proactively composed and analyzed as described in [4].

As shown in Figure 4, LMS receives EG definition and generates relevant infrastructure policy rules for capturing the membership. Figure 7 shows an example of EG written in a Disjunctive Normal Form (DNF) of Boolean expression of a set of labels *(key:value)*. In this example, there are two label trees: *LOCATION* and *TENANT*. We want to create an intent which allows HTTP traffic from $EG1$ to *EG2*, where *EG1* means to check if any entity is a member of a tenant *admin* and is located on the location *AZ1* and *EG2* means to check if any entity is a member of a tenant *admin* and located on the *AZ2*. Then, the EG definition is converted to a set of Datalog using the label definitions. Whenever any matched entity is appeared in the table for EGs, LMS detects and updates mapping entities and EGs, then intents for the EGs can be applied to the entities.

1) Mapping using Congress: In order to achieve this automatic mapping, we have started to use Congress itself as shown in Algorithm 1. The Mapping function in Algorithm 1 performs the task. It takes the given entities and Datalog rules for EGs. First, all entities are iterated for associating each entity to the matched EG by Datalog rules (IsMatched function) which evaluates all rules for each label in the EG. The time complexity of this algorithm is *O(mnk)*, where *n* is the number of entities, *m* is the number of EGs, *k* is the number of labels, respectively.

Algorithm 2 Map entities to EGs (hashing)

```
1: function GETEGNAME(Entity, Properties)
2:     EGName := Entity[Properties[0]]   ▷ Entity has at least one property entry which has a key and value
3:     for i := 1 to Properties.length do
4:         EGName := EGName + "-" + Entity[Properties[i]]
       return EGName

5: function MAP(Entities, EGs, Properties)
6:     table := ∅
7:     for EG ∈ EGs do
8:         table[EG.name] := ∅    ▷ Initialize a hash table for all EGs. The format of EG.name is label-···-label.
9:     for Entity ∈ Entities do
10:        EGName := GETEGNAME(Entity, Properties)
11:        table[EGName].add(Entity.id)          ▷ Update the hash table
```

Algorithm 3 Map entities to EGs (spatial hashing)

```
1: function MAP(Entities, EGs, Properties)
2:     table := ∅
3:     for Entity ∈ Entities do
4:         EGName := GETEGNAME(Entity, Properties)
5:         if EGName ∈ table.keys() then
6:             table[EGName].add(Entity.id)
7:         else
8:             table[EGName] := ∅
9:             table[EGName].add(Entity.id)          ▷ Update the hash table
```

2) Mapping using hashing: In order to improve the scalability, we have used a hash table as shown in Algorithm 2. Instead of evaluating all Datalog rules for each entity, we create a hash table in which the key is the name of EG, which can be generated from each entity's properties. The time complexity is *O(nk)* if $n \geq m$ *or O(mk) if* $n < m$, where *n* is the number of entities, *m* is the number of EGs, k is the number of labels.

3) Mapping using spatial hashing: However, Algorithm 2 has also an optimization challenge in the hash table for handling the scalability of the number of EGs. We do not have to create all entries in the hash table but create only entries which has an associated entity. Algorithm 3 shows the optimized function for the task. This function creates a spatial hash table for only the given entities. The time complexity is $O(nk)$, where n is the number of entities and k is the number of labels. In Section V-C2, we will show the comparison results of three algorithms in term of time complexity. We have used one of well-known optimized hashing techniques and our algorithm can be improved with better hashing algorithm.

V. PROTOTYPE AND EVALUATION

In this section, we describe the implementation details of our proposed LMS framework and evaluate it using OpenStack. We showcase the scalability and effectiveness of our framework with its resource consumption and latency aspects.

A. Prototype Development

We developed LMS prototype on the latest version of OpenStack Mitaka [17] using Python. Like other OpenStack services, we have built GUI for LMS as an OpenStack dashboard GUI module, implement python-based LMS client module

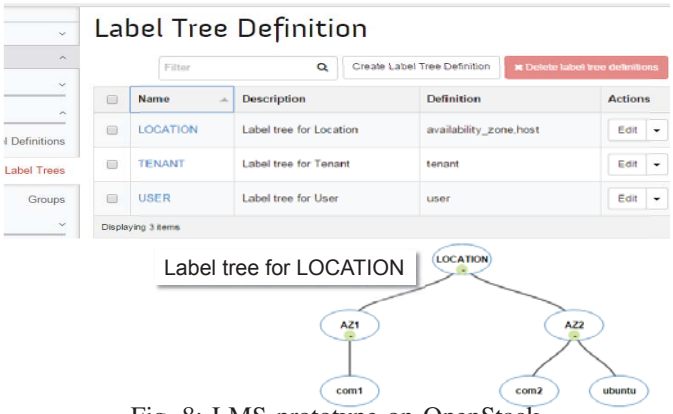


Fig. 8: LMS prototype on OpenStack

and LMS backend service for making our solution complete and portable to OpenStack. All LMS-related components are written in approximately 5000 lines of Python and 1000 lines of javascrtipt/HTML/CSS. Figure 8 shows a snapshot of the current GUI. The GUI will provide a easier and effective means to manage labels, label tree definitions and groups.

B. Testbed Setup

For our evaluations, an OpenStack controller with LMS is deployed on a server with 2x8 Intel Xeon 2.6 GHz cores with 128 GB memory running Linux kernel 3.19.0. We used two OpenStack services: `Keystone` for identity service and `Nova` for compute service. In this experiment, we used two data-source tables (user, project) from `Keystone` and three data-source tables (availability_zone, compute_node, instance) from `Nova`. Using the label tree definition in Section IV-B2, we created and maintained three label trees (*USER, TENANT, LOCATION*) by increasing a number of data entities such as the number of users and projects. For evaluating the cost involved in automatic mapping of entities to EGs, variable of number of VM instances and EGs are considered.

Test Data: For micro benchmark results shown in Figure 9, 10 and 11, we have used the real testbed data. For experimenting with medium scale testbed (of AWS type [18]) with thousands of VMs (Figure 12), we have synthetically created data set based on the OpenStack database schema (the synthetic testdata is kept in-sync with the micro benchmark real data).

C. Evaluation

In this section, we present our initial experimental results obtained with the prototype developed on top of OpenStack for evaluating the scalability of LMS on a dynamically changing cloud environment. We present the latency and resource over-head details of label tree construction and entity group mapping mechanisms.

1) Evaluation of label tree construction: To simulate a medium scale cloud environment, we experimented with up to 100k users, tenants and hosts. At each iteration, we repeated the experiments by constantly increasing the number of tenants, hosts and users (leaf-nodes) of the cloud infrastructure by a size of 5000. We conducted the same experimental procedure over hundred times with randomly generated data sets to approximate the latency and memory usage. Figure 9 shows the time and memory required to construct 1 to 3 label trees with increase in the number of users, tenants, and hosts (leaf nodes). The average time required to build 3 different types of label trees with leaf-node size of 100k is ≈20 seconds and consumes ≈170MB for building the necessary label trees. The LMS module adds very minimal overhead to the memory and CPU (with in ≈5%) utilization while building the label-trees on the server mentioned in Section V-B.

In our experiment at each iteration, we incrementally added ≈5000 leaf-nodes (users, tenants and hosts), for building the label trees, and the resultant label trees are built (updated) instantly with in ≈800 msec to ≈1 sec on top of the existing label trees. We can easily infer from current results, that even in the cloud environments with millions of entities, initial construction up of label-tree might consume considerable time (up to few minutes), later the incremental updates (i.e., updating the existing label trees with non-root node additions and deletions) should happen faster in few seconds.

2) Evaluation of entity and EG mapping: We evaluated the latency and memory over-head of the entity and EG mappings and compared them using three different algorithms described in Section IV-D. For our experiments, we used a VM instance as an entity and tested with 10, 50, 100, and 500 VM instances, respectively. For creating each VM, we randomly select a user and its associated project and host on which VM needs to be instantiated. Because EG can be defined with Boolean operation of a label (leaf node) per tree, the maximum number of EGs is n^m, where n is the number of leaf nodes per tree and m is the number of trees. In this experiment, we incrementally increased the number of leaf nodes by 10. Figure 10 shows time and memory spent in mapping different number of VMs and EGs for three of the algorithms (§IV-D).

Latency Improvements: As shown in Figure 10(a), this approach takes considerably higher time on increasing the number of VM instances (v) and the number of EGs (g) as the algorithm's time complexity is $O(vg)$. As shown in Figure 10(b), the Algorithm 2 remarkably reduces time when compared with Algorithm 1 used in Congress policy engine. However, it is not scalable on increasing the number of EGs. Finally, we used a partial hash table in which a hash entry is created only for existing VM instances. That is, we do not have to create all hash entries for all EGs. As shown in Figure 10(c), Algorithm 3 outperforms other two algorithms in terms of time. It is also scalable over the number of EGs and depends only on the number of VM instances. One of the interesting point in Figure 10(c), We found the error bar is longer than other two results because Algorithm 3 creates hash table entries which have any matched entity. That is, in case of mapping the same number of VMs, the latency depends on how many entries in the hash table are mapped with VMs.

Memory Benefits: Similarly, the memory benefits are illustrated only for 500 VM instances (Figure 11) for three different algorithms (described in Section IV-D). The Algorithm 3 outperforms others as this algorithm uses partial hash table in which the entry is created only for existing VM instances. This approach is highly scalable as this algorithm depends on

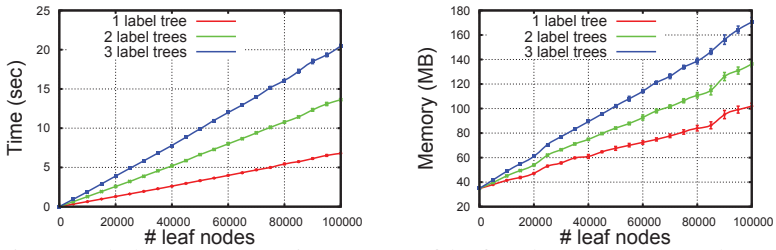
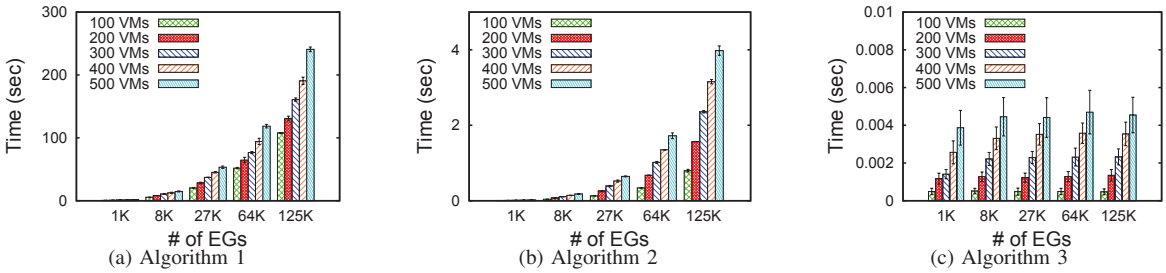


Fig. 9: Label tree construction over # of leaf nodes (user, tenant, host)



(a) Algorithm 1 (b) Algorithm 2 (c) Algorithm 3

Fig. 10: Mapping latency for entities and EGs

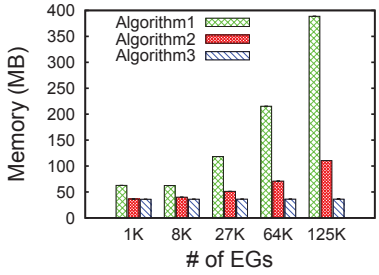


Fig. 11: Memory consumption for entities (500 VMs)

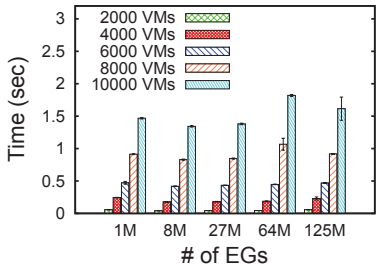


Fig. 12: Mapping latency for large scale cloud (Algorithm 3)

the number of VM instances rather than on the number of EGs, there by utilizing a constant amount of memory over varying the number of EGs as shown in Figure 11.

We also evaluate the scalability of our Algorithm 3 with simulating real cloud scale data using available size data from Amazon Web Service (AWS) Cloud [1], [18], [19]. Currently, AWS Cloud operates 35 AZs within 13 geographic regions around the world which contains more than 2 million servers by 1M customers. So, we need to test with bigger number of VMs and EGs. Figure 12 shows time spent in the case for mapping bigger number of VMs over bigger different number of EGs based on the AWS case. In the case of 10k VMs, it takes only ≈1.3∼1.8 seconds regardless the size of EGs.

VI. Related Work

Expressing high-level policies in terms of logical labels or tags is not new [20], [21]. It has been used in Security-Enhanced Linux [22] and more recently in Docker [9] and Kubernetes [10] and OpenDaylight policy/intent projects [23], [24]. Docker and Kubernetes uses simple key-value pairs for their labels to define the properties of container applications and pods. However, these models do not actually capture the relations between labels. ONF Boulder group is currently defining SDN-based North Bound Intent (NBI) which is a paradigm/methodology for interaction between SDN Applications and forwarding infrastructure [2], which is interested in separating high-level intents from low-level implementations by effectively translating NBI requests and executable system commands. We believe LMS is one of the best solutions for achieving the goal. OpenStack Neutron is also considering to add tags to neutron core resources [25]. LMS can help to generate system-generated tags instead of user-generated tags making groups and automatically capturing relations between groups and resources.

VII. Conclusion and Future Work

In this paper, we have proposed a label management service (LMS) to manage dynamic labels and map entities and EGs for intent-driven cloud management. First, we have defined a label syntax and EG using the logical *labels*. Second, we have shown how to automatically construct a label namespace by collecting data from cloud infrastructure, how to express intents using the logical labels and compose them using label trees, and shown how to dynamically map an entity to EG. Finally, we have showed performance results for constructing label trees and mapping entity to EGs.

As future work, we will test our LMS on large-scale cloud environments to evaluate the latency and resource consumption overhead. As an early stage work, we are integrating it with OpenDaylight Network Intent Composition (NIC) by storing label trees and entities to the mapping service [26].

REFERENCES

[1] "A rare peek into the massive scale of aws," http://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/.

[2] D. L. et al., *Open Networking Foundation – Intent NBI - Definition and Principles*, Std. ONF2015.327, 2016.

[3] "NeMo: An Application's Interface to Intent Based Networks," http://nemo-project.net.

[4] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 29–42.

[5] "GBP: Group-Based Policy," https://wiki.opendaylight.org/view/Group_Based_Policy_(GBP).

[6] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain, "An intent-based approach for network virtualization," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, May 2013, pp. 42–50.

[7] Minh Pham and Doan Hoang, "SDN applications - the intent-based Northbound interface realisation for extended applications," in *IEEE Workshop on SDN and IoT (SDN-IoT 2016)*, June 2016.

[8] Dave Lenrow, "Intent: What. Not How." https://www.opennetworking.org/?p=1633&option=com_wordpress&Itemid=155.

[9] V. Jain, "Proposal - Operational Intent for Docker Deployment," https://github.com/docker/docker/issues/11187.

[10] "Labels and Selectors," http://kubernetes.io/docs/user-guide/labels/.

[11] "OpenStack," http://www.openstack.org.

[12] "OpenStack Governance – Congress," https://wiki.openstack.org/wiki/Congress.

[13] J. D. Ullman, *Principles of Database and Knowledge-base Systems, Vol. I*. New York, NY, USA: Computer Science Press, Inc., 1988.

[14] Martin Casado, "Policy in OpenStack." Presented as the Keynote in OpenStack Silicon Valley, September 2014.

[15] "Docker," http://www.docker.com.

[16] "Kubernetes," http://kubernetes.io/.

[17] "OpenStack Mitaka Release," https://www.openstack.org/software/mitaka/.

[18] "Aws glopbal infrastructure," https://aws.amazon.com/about-aws/global-infrastructure/.

[19] "5 numbers that illustrate the mind-bending size of amazon's cloud," http://www.bloomberg.com/news/2014-11-14/5-numbers-that-illustrate-the-mind-bending-size-of-amazon-s-cloud.html.

[20] "Labeling," https://en.wikipedia.org/wiki/Labelling.

[21] "Tag," https://en.wikipedia.org/wiki/Tag_(metadata).

[22] "SELinux/Labels," https://wiki.gentoo.org/wiki/SELinux/Labels.

[23] "OpenDaylight Group Based Policy," https://wiki.opendaylight.org/view/Group_Based_Policy_(GBP).

[24] "OpenDaylight Network Intent Composition," https://wiki.opendaylight.org/view/Network_Intent_Composition:Main.

[25] "Add tags to core resources," http://specs.openstack.org/openstack/neutron-specs/specs/mitaka/add-tags-to-core-resources.html.

[26] Anu Mercian, Felipe Yrineu, Joon-Myung Kang, Raphael Amorim, Saket M Mahajani, Mario Sanchez and Sujata Banerjee, "Network Intent Composition (NIC) Be Feature Update and Demo: Intent Compilation, Lifecycle Management and Automated Mapping." Presented in OpenDaylight Summit 2016, September 2016.