2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

CloudSight: A tenant-oriented transparency framework for cross-layer cloud troubleshooting

Hyunwook Baek*, Abhinav Srivastava† and Jacobus Van der Merwe*
*University of Utah
†AT&T Labs - Research

***Abstract*—Troubleshooting in an infrastructure-as-a-Service (IaaS) cloud platform is an inherently difficult task because it is a multi-player as well as multi-layer environment where tenant and provider effectively share administrative duties. To address these concerns, we present our work on CloudSight in which cloud providers allow tenants greater system-wide visibility through a transparency-as-a-service abstraction. We present the design, implementation, and evaluation of CloudSight in the OpenStack cloud platform. We also develop two example applications that make use of the CloudSight abstraction and use the applications to explore real cloud problems.**

I. Introduction

Today's IaaS cloud software systems are under active development. To survive in the cloud marketplace, different software vendors competitively introduce new features. Developing new features over short development cycles for a cloud platform, which is essentially a sophisticated distributed system, has the potential to introduce bugs, which might impact the reliable operation of the system. The problem is, in a cloud environment, it is neither the cloud developer nor the cloud provider, but rather the cloud tenant who most often encounter new bugs.

This problem is exacerbated because cloud tenants have almost no means to deal with an infrastructure level bug. Various cloud monitoring tools are being offered to cloud tenants, but no single tool can clearly indicate whether the cause of a problem is from the provider's or the tenant's side. For example, cloud tenants have a difficult time getting answers to questions such as:

• "I have recently rebooted my VM, but cannot access it after a while. Is it because I have misconfigured my VM or because the provider's server is down?"

• "I have created a firewall, but the forbidden packets are still bypassing it. Is the firewall not actually created? Did I make a mistake? Or is it a system bug?"

• "I created a virtual interface that has silently disappeared. When was it deleted? Did somebody else delete my interface or did I?"

The fundamental reason why tenants cannot get an answer for these questions is the way today's IaaS cloud platforms present an abstract view of the states of resources to tenants. IaaS cloud platforms simplify the work of cloud tenants by providing clean virtual resource abstractions. Specifically, IaaS cloud platforms maintains the states of resources in a central database, and forward the information to tenants when tenants query the platform. The problem is that the states recorded in the database are not necessarily consistent with the real-world states of resources. We call this notion '*functional (in)consistency*' of a resource[1]:

Def. *If a virtual resource **X** is **functionally consistent** at time **t**, the data object representing the state of resource **X** in the system must correspond to the real-world state of resource **X** at time **t**.*

A resource can be functionally inconsistent either because the actual process of creating/deleting/modifying the resource is not yet complete, or because of a bug or misconfiguration of the system. Regardless, for cloud tenants, there is no way to identify such inconsistency, and this leads to uncertainty with respect to the cause when problems occur. For a clear answer, tenants must depend on support from the provider, which is costly for both tenant and provider.

Explicitly telling the tenants about inconsistencies can help the tenants help themselves. In other words, if a tenant can distinguish the state of a resource written in the central database from its state directly monitored in the cloud infrastructure and/or its state according to a unit tester, the tenant can answer the questions listed above by themselves.

In this paper, we introduce *CloudSight*, a transparency-as-a-service framework that maintains the history of states of cloud resources from various vantage points, and two applications on top of CloudSight: *functional consistency verifier* and *time-traveling cloud debugger*. CloudSight dynamically inserts monitoring functions into the target IaaS system to monitor state changes of resources in different vantage points, stitches together state information of the same resource monitored in different places, and maintains the state change history in a graph database. CloudSight applications can trace resource state histories of interest using the Gremlin graph traversal language.

The aforementioned fact that IaaS software systems are under active development introduces a significant challenge in the system design of CloudSight: compatibility. For example,

[1]The term *functional consistency* originates from the real-time system research of Audsley et al. in 1993 [1].

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.97

OpenStack, one of the market leading IaaS software systems, releases a new major version update approximately once every six months. To keep invasive monitoring tools such as CloudSight compatible, it is required to update the expert level domain knowledge about the target system and to patch the tools accordingly. Likewise, to increase the coverage of monitoring, similar effort is required. Thus, if a tool is made by simply understanding a specific version of the target system and modifying the target system, the amount of time and effort to maintain compatibility becomes significant. We combine two key techniques to resolve this challenge. First, CloudSight uses *key clustering* to automate updating domain knowledge of the target system. Key clustering is adopted from the research on entity resolution [2], which is about extracting, matching and resolving entities appearing in heterogeneous records. Second, *dynamic code injection* directly permits CloudSight to inject monitoring functions into the target system without modifying the target system's source code. This enables CloudSight to be applied to a cloud platform regardless of the software version of the cloud system. In addition, this technique also allows cloud providers to dynamically adjust the monitoring level by turning on/off desired monitoring functions.

In summary, we make the following contributions:

• We show how the visibility problem of cloud tenants can be reduced to the visibility of the functional consistency of resources, and show how the CloudSight transparency-as-a-service framework effectively realizes this visibility.

• We adopt an entity resolution algorithm from the distributed systems domain, which efficiently minimizes the effort to maintain domain knowledge of the target cloud system in CloudSight.

• We present a working prototype implementation of the CloudSight framework in OpenStack, as well as two applications, a *functional consistency verifier* and a *time-traveling cloud debugger*.

• We demonstrate the utility of CloudSight by diagnosing real problems, one of which is a critical security bug first identified by our work [3].

II. CloudSight Architecture

Overview. CloudSight inserts loggers into existing cloud components to obtain comprehensive monitoring of the infrastructure. The logs obtained by the loggers are stored in log storage and forms the basis for CloudSight's transparency-as-a-service abstraction. Specifically the logs are processed to generate a resource graph, which essentially represents changes of resources' states in both the cloud database and the cloud infrastructure. The information contained in the resource graph can be queried via the CloudSight API by tenants and tenant applications.

Figure 1 depicts the operational phases of CloudSight. A variety of cloud data sources are collected from the cloud

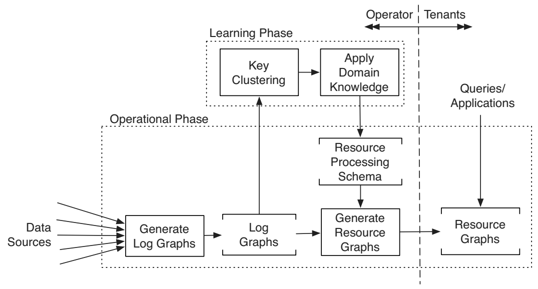


Figure 1: CloudSight Operation

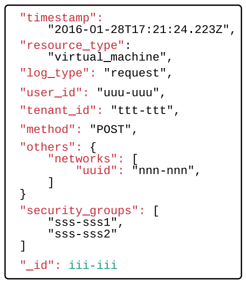
platform and transformed into graph format to generate a log graph. The log graph forms the main input for the offline *learning phase*. Specifically we apply semantics-based clustering, termed *key clustering,* to associate related data from different data sources. The cloud operator applies domain knowledge to the output of this step to generate a resource processing schema. During the *operational phase* this schema serves as input to generate a resource graph which represents the means whereby CloudSight exposes information to cloud tenants.

A. Instrumentation and Log-preprocessing

CloudSight loggers require minimal domain knowledge about the cloud platform. Specifically, because CloudSight *key clustering* associates information from different data sources, a logger developer can simply dump data into an event log, without attempting to interpret the data in the broader cloud context. We describe the implementation of loggers in OpenStack platform in Section III.

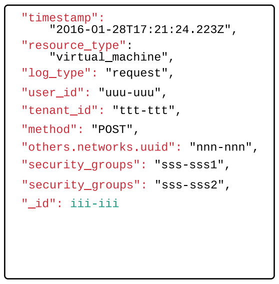
CloudSight first preprocesses the collected event logs to narrow down the range of semantics of each attribute. To be specific, each nested attribute-value structured log is flattened into a one-level attribute-value structure by concatenating the chain of attributes of each inner-most value. We call this chain of attributes a *combinational key*. For example in Fig. 2(b), the edge labels `others.networks.uuid`, `security_groups`, `method`, and so on are all combinational keys, derived from the nested log shown in Fig. 2(a). A combinational key can be understood as a *contextualized* attribute. For example, the combinational key `others.networks.uuid` in Fig. 2(b) can be distinguished from another combinational key, e.g., `instance.nova_object.data.uuid`, which cannot be distinguished if not flattened since both have the same last attributes `uuid`. The preprocessed logs are stored in the graph database, so we call the graph storing flattened logs a *log graph*.

The flattened log in Fig. 2(b) will be fed to the resource graph generator together with the resource processing schema (Fig. 2(c)), and, accordingly, the entries of the resource graph will be updated as in Fig. 2(d). In §II-B and §II-C, we explain the details of the generation of the resource graph and the resource processing schema.



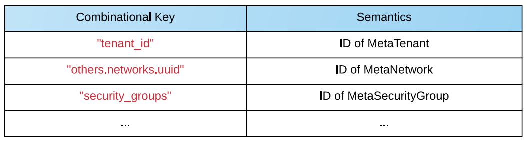
```
"timestamp":
    "2016-01-28T17:21:24.223Z",
"resource_type":
    "virtual_machine",
"log_type": "request",
"user_id": "uuu-uuu",
"tenant_id": "ttt-ttt",
"method": "POST",
"others": {
    "networks": [
        "uuid": "nnn-nnn",
    ]
}
"security_groups": [
    "sss-sss1",
    "sss-sss2"
]
"_id": iii-iii
```

(a) Example Log



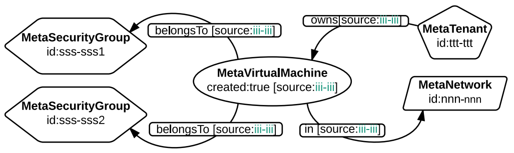
```
"timestamp":
    "2016-01-28T17:21:24.223Z",
"resource_type":
    "virtual_machine",
"log_type": "request",
"user_id": "uuu-uuu",
"tenant_id": "ttt-ttt",
"method": "POST",
"others.networks.uuid": "nnn-nnn",
"security_groups": "sss-sss1",
"security_groups": "sss-sss2",
"_id": iii-iii
```

(b) Flattened Example Log



Combinational Key	Semantics
"tenant_id"	ID of MetaTenant
"others.networks.uuid"	ID of MetaNetwork
"security_groups"	ID of MetaSecurityGroup
...	...

(c) Resource Processing Schema



(d) Example Log in Resource Graph

Figure 2: Log-to-graph conversion

(a) A nested attribute-value structured log is transformed into (b) a single level attribute-value structured log. This preprocessed log can be converted into (d) state information of resources at a specific time in the *resource graph* by referencing (c) the *resource processing schema*.

B. CloudSight Resource Graph

CloudSight follows a resource-centric approach in presenting cloud information to cloud tenants. Data from different cloud sources are combined in a resource graph. This represents a natural way for tenants to think about their cloud resources and the relationships between them.

Figure 3 illustrates how different cloud resources are associated in a graph structure. A VM created by a tenant via the cloud API becomes a node vertex in the graph ($VM1$) with a set of properties (i.e., its *ID*, *Name*, the fact that it has been *Created* and its *Type*. '`meta`' indicating that this is a cloud database object associated with a VM). In this example, the cloud controller decides to instantiate the VM on a physical machine ($PM1$), resulting in an $assignedHost$ relationship between the two vertices in the graph. Similarly, once the hypervisor on host $PM1$ instantiates the VM, a $Type : VM$ node (*vm*001) is created in the graph with an $isHosting$ and $instantiatedIn$ relationships with vertices $PM1$ and $VM1$.

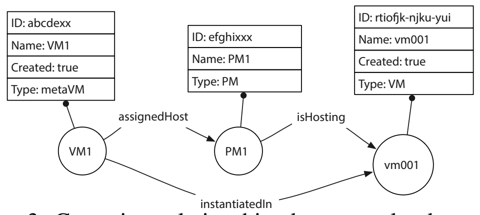


Figure 3: Capturing relationships between cloud resources

To capture the history and state transitions associated with cloud resources, CloudSight employs list-based property cardinality (allowing multiple properties with the same property key), as well as nested properties (allowing a property to be contained in another property).

Since *functional consistency* of a cloud resource may vary as time progresses, cloud troubleshooting often involves investigating states of resources at a specific time (or within a specific known time range). The resource graph can easily be *projected* onto a time-plane to facilitate such efforts. Specifically, we can create a projected time-plane subgraph by selecting vertices whose event log timestamps fall within the time range of interest.

The CloudSight resource graph maintains a complete history of the cloud resources and their state transitions over time. Allowing cloud tenants direct access to the complete graph is inappropriate as it contains information for all tenants, as well as information the cloud provider might consider proprietary. Similar to time-plane projection, the CloudSight resource graph can readily be projected onto a tenant-plane. The base approach to project the resource graph onto a tenant plane involves creating a subgraph limited to vertices that have direct edges to the tenant vertex. We can then extend the subgraph to include every connected edge and neighbor vertex and can obfuscate or hide properties associated with neighbor vertices based on cloud provider domain knowledge (applied during the learning phase).

C. Key Clustering

Event sequencing approaches based on meta-data propagation are generally invasive as it requires changes to the system itself [4], [5], [6]. Schema-based approaches are generally less invasive, but requires manual construction of schema to associate events from different data sources [7], [8]. Thus, both approaches needs extensive domain knowledge, and the need is exacerbated in cloud platforms. This is especially true for open source efforts. I.e., not only are these platforms under active development but they typically also have a large number of contributors with relatively loose coordination between them. Since different developers may use different attribute names for the same entity in different components, understanding the entire system through source code to update domain knowledge is not an easy task. For example in OpenStack, in different functions and messages, argument names `device_id` and `routers` are used to refer to the UUID of a virtual router, but `device` is used to refer to the UUID of a virtual interface; also, `uuid` is used to refer to the UUID of many entities including virtual router and virtual interface.

With CloudSight event sequencing, we essentially follow a schema based approach, but reduce the domain knowledge

Algorithm 1 Key Clustering

```
1:  $\mathbb{C} \leftarrow \phi$
2:  for each $attr \in \mathbb{A}$ do
3:      $hasCluster \leftarrow$ False
4:      for each $c \in \mathbb{C}$ do
5:          if $c.values = attr.values$ then
6:              $c.keys \leftarrow c.keys \cup \{attr\}$
7:              $hasCluster \leftarrow$ True
8:              break
9:      if $hasCluster =$ False then
10:         $c \leftarrow$ new Cluster()
11:         $c.values = attr.values$
12:         $c.keys \leftarrow \{attr\}$
13:         $c.super \leftarrow \phi$
14:         $c.sub \leftarrow \phi$
15:         $\mathbb{C} \leftarrow \mathbb{C} \cup \{c\}$
16: for each $c_1, c_2 \in Comb(\mathbb{C}, 2)$ do
17:     if $c_1.values \in c_2.values$ then
18:         $c_1.super \leftarrow c_1.super \cup \{c_2\}$
19:         $c_2.sub \leftarrow c_2.sub \cup \{c_1\}$
20:     else if $c_2.values \in c_1.values$ then
21:         $c_2.super \leftarrow c_2.super \cup \{c_1\}$
22:         $c_1.sub \leftarrow c_1.sub \cup \{c_2\}$
    return $\mathbb{C}$
```

required by automatically updating the event schema during the learning phase through *key clustering*, which is inspired by Entity Resolution research [2]. Key clustering is based on the following syllogism: if we know *(1) attribute $Attr_1$ denotes* X, and we know *(2) attribute $Attr_2$ is the same as attribute* $Attr_1$, then we can conclude *(3) attribute $Attr_2$ denotes* X. Here, the key challenge is how to obtain proposition *(2)*, i.e., semantic relations between attributes. CloudSight gathers semantic relations of attributes by clustering attributes based on the following semantic definitions. We define a *Value Space* of an attribute $Attr_x$ as *the set of values that attribute $Attr_x$ ever had* in the given learning data set, and denote it as $V(Attr_x)$. Then, the relationships between attributes are:

$$Attr_1 \text{ is a } \textbf{synonym} \text{ of } Attr_2 \Leftrightarrow V(Attr_1) = V(Attr_2)$$
$$Attr_1 \text{ is a } \textbf{hyponym} \text{ of } Attr_2 \Leftrightarrow V(Attr_1) \subseteq V(Attr_2)$$
$$Attr_1 \text{ is a } \textbf{hypernym} \text{ of } Attr_2 \Leftrightarrow V(Attr_1) \supseteq V(Attr_2)$$

where a hyponym (hypernym) means a more specific (generic) term in Semantics. Based on this definition, we can cluster entire attributes by grouping synonyms in a cluster, namely a *Synonym Cluster*. Moreover, we can define the hierarchy among clusters by making hyponym clusters and hypernym clusters to point to each other. As a consequence, if we know the meaning of an attribute, we can infer the meaning of the other attribute in the same cluster as well as the generic meaning of attributes in all hyponym clusters.

Algorithm 1 shows the steps for the CloudSight key clustering. Here, $\mathbb{A}$ refers to the set of every attribute (i.e., combinational key), which has property *values* (value space), *Comb()* refers to a function returning combination of elements of a given set, and *Merge()* refers to a function that merges two master clusters and updates their sub-clusters accordingly.

f1 :	The original function translates a security group rule into iptables rule. The injected code supports mapping from security group rules to iptables rules by inserting security group information into comment fields of iptables.
f2 :	The original function returns a network interface name by accessing dictionary object. The injected code changes a character of the returning value to support (security group) active logging.
f3 :	The original function updates iptables rules. The injected code monitors the changes of iptables, which show the real-world states of Security Group rules.
f4, f5 :	The original functions are LibVirt VM creation and deletion functions. The injected code monitors VM creation/deletion events at the edge, which show the real-world state changes of VMs.
f6, f7 :	The original functions are network resource API functions. The injected code monitors network API requests and their propagation to the cloud database.
f8, f9 :	The original functions are computing resource API functions. The injected code monitors computing API requests and their propagation to the cloud database.

Table I: The OpenStack Functions Hooked by CloudSight

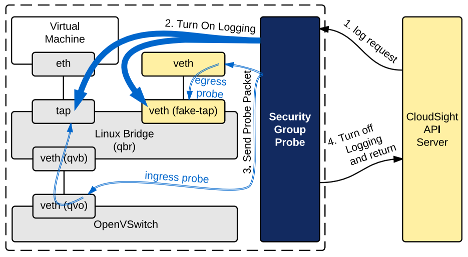


Figure 4: Security Group Active Logger in OpenStack

Once we finish key clustering, we can infer the meaning of combinational keys in a synonym cluster if we know the meaning of any combinational key in it. One of the most practical sources of initial domain knowledge is the database of the target cloud, where most of possibly existing keys are used and usually is well documented by developers. Since this metadata is included in the data used by CloudSight, this provides a reasonable starting point for providers to apply their domain knowledge.

III. Implementation

We now describe our prototype implementation of CloudSight in the OpenStack cloud computing environment.

Instrumentation. The CloudSight loggers are developed based on OpenStack Icehouse with Neutron OVS Hybrid networking (ML2) in a minimally invasive way; OpenStack components are implemented in Python and CloudSight loggers "hook" the target OpenStack components without changing the original source by replacing the entry module for each component. The substitute is essentially the same as the original, but it has additional code that dynamically patches selected functions of OpenStack by importing them in advance and wrapping them with a CloudSight wrapper. Table I summarizes the list of the functions we hooked and what each injected code does in each function. In addition to these loggers, CloudSight also monitors RPC messages exchanged among OpenStack components to trace the state changes of resources at the edge.

Unit Test: Unit testing in CloudSight checks if the target resource is functioning by effectively mimicking controlled user behavior. This enables CloudSight to monitor the functional consistency of resources in a more strict way: *if a virtual resource **X** is not only existing, as it is supposed to, but is also functioning correctly*. We have implemented example unit test functionality for Security Groups in a form of an active logger. Figure 4 depicts the security group active logger implementation. The goal of the active logger is to trace which security group rule is used for a specific packet. Once an active logging request is received, the active logger enables iptables rules tracing, sends a probe packet, and then disables tracing. The security group unit test can be realized by sending a probe packet directly into the bridge such that it passes through the tap devices. For ingress rules, the active logger can send a fabricated packet to qvo. The probe packet will go through qvb and qbr and reach the tap device. Since we cannot make a packet flow from the tap device to the bridge without aid from the target virtual machine, realizing an egress probe packet is a bit more involved. We solve this problem by temporarily creating a fake interface that has the target tap device name as its prefix, and add a wildcard character at the end of iptables rules' target device name so that the rule will also be applied on the fake device.

Platform-independent Components. We used a combination of Logstash and Elasticsearch to implement the forwarding of logs from the cloud components to the central log storage system. We used Titan 1.0 as the graph database and Gremlin [9], a graph traversal language developed under the Apache TinkerPop project, as the front-end. The internal graph components such as the graph generator, graph projectors and the key cluster learner are implemented in Gremlin-Groovy. The projected resource graphs can be delivered to tenants through a Gremlin Server. The Gremlin Server allows tenants to make queries using the Gremlin language or using applications written in the Gremlin language. *Time-traveling Cloud Debugger* (§IV-A) and *Functional Consistency Verifier* (§IV-B) are written in Gremlin-Groovy as example applications which interact with the Gremlin Server.

IV. Applications

To demonstrate the efficacy of our framework, we developed two novel tenant-oriented applications that assist tenants in troubleshooting the cloud problems.[2]

A. Time-traveling Cloud Debugger

Debugging a large scale distributed system, such as a cloud platform, often involves determining the state of the system at a specific point in time, and tracking the changes over time. We developed a novel time-traveling interactive cloud debugger on top of CloudSight to assist tenants to track their resources and troubleshoot problems. With the cloud debugger, cloud users can set a *timepoint*, similar to setting a *breakpoint* in a typical software debugger. Once the *timepoint* is set, the tenant can query the CloudSight's resource graph to probe the status of their virtual datacenter at that point of time. For instance, tenants can obtain a list of existing resources, the connections among the resources, and the properties of each of those resources at the timepoint. Moreover, tenants can also track changes in a desired resource as time progresses, or explore the entire history of the resource. Table II shows the commands that the cloud debugger currently supports.

[2]A video, based on a demonstration of the CloudSight applications [10], is available from the project website: http://www.flux.utah.edu/project/tcloud.

Table II: Cloud Debugger Commands

`> show tplist:` list all available timepoints in a tenant space.
`> show reslist:` list all resources belonging to a tenant.
`> set tp=(timepoint_index):` set the timepoint.
`> unset tp:` unset the current timepoint.
`> next:` move to the next timepoint.
`> show tp:` show the current timepoint.
`> show (resource_index):` show the properties and connections of the resource at the current timepoint.
`> history (resource_index):` show entire historical changes of the resource.
`> dump:` dump the snapshot of the virtual datacenter at the current timepoint into a graph.

An example use case for this CloudSight application involves investigating the OpenStack port disappearance bug (*OpenStack bug#1158684*) [11]. In this case a VM port (i.e., a virtual interface) silently disappears in certain versions of OpenStack. (Correct behavior involves deletion of ports implicitly created by OpenStack, while ports explicitly created by tenants are not deleted but only detached.) With the cloud debugger, the tenant could trace back to the moment when the port was deleted and identify the termination of the virtual machine associated with the port triggered the port deletion [10].

B. Functional Consistency Verifier

As discussed earlier, one of the main problems that tenants face in the cloud is to determine if their rented resources are functionally consistent. We develop a simple yet powerful application called *Functional Consistency Verifier* built atop CloudSight to determine the functional consistency history of a given resource.

Practical examples of using the functional consistency verifier include investigating policy violations of virtual machine affinity group functionality [12], or investigating security group bypassing problems (*OpenStack bug #1359691*) [3].

V. Related work

Our work relates to earlier efforts that can be classified as those that assist cloud providers to troubleshoot the cloud and those aimed at assisting tenants to debug and troubleshoot.

Tenant Domain Tools. CloudWatch [13] and Ceilometer [14] provide visibility into the performance metrics of virtual instances. Sharma et al. [15] developed a tenant level end-point based cloud monitoring tool that allows tenants to conduct customized monitoring on their resources. Wu et al. [16] presented a cloud monitoring framework especially focusing on tenant network packet tracking. Amazon AWS CloudTrails [17] offers users an API history to track their operation in a virtual datacenter, similar to CloudSight's logging requests from API servers. The scope of these cloud-level tools is typically limited to identifying symptoms of problems in their cloud resource instances. Our work is perhaps most closely related to the efforts of Wencheng et al. [18], who tried to resolve the visibility problem by offering a predefined-knowledge-based troubleshooting tool to cloud tenants. In contrast to these earlier efforts, CloudSight adopts a holistic approach that combines information from various places in the cloud, providing unique visibility to cloud tenants and assisting cloud operators to apply their domain knowledge to a complex and evolving cloud platform.

Provider Domain Tools. Ju et al. [5] built an intrusive failure injection framework for OpenStack. Their framework could trace internal task flows to narrow down the root cause for a given OpenStack error. For the same problem, Sharma et al. [19] developed an online analysis tool based on RPC and API messages. Regarding the consistency problem of cloud resources, Xu et al. [20] introduced network consistency checking based on comparison of metadata from the cloud controller and the state of actual network resources on edge nodes. Madi et al. [21] adopted graph structure similar to CloudSight to cloud security compliance auditing. Likewise, Xiang et al. [22] used a similar graph structure as a general knowledge base for debugging cloud infrastructure. Compared to these works, CloudSight is unique in that it extends the consistency problem to functionality of resources by offering unit test against resources through active logging as well as enabling tenants to troubleshoot cloud issues in a holistic manner.

VI. Conclusion and Future Work

CloudSight is a framework that provides IaaS cloud tenants greater visibility into their virtual data centers. With CloudSight, tenants can understand cloud mechanisms better, debug their cloud applications more precisely, diagnose even provider side problems, and interact with providers more efficiently. We illustrated the utility of CloudSight by showing how it addresses real world problems in the OpenStack cloud platform. Using CloudSight we plan to explore a synergistic cloud paradigm where cloud providers and tenants can cooperate more efficiently to address cloud problems.

Acknowledgment

This work was initiated when the student was an intern at AT&T Labs - Research and was supported in part by the National Science Foundation under grant number 1314945.

References

[1] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Data consistency in hard real-time systems," Department of Computer Science, University of York, Tech. Rep., 1993.

[2] L. Getoor and A. Machanavajjhala, "Entity resolution: theory, practice & open challenges," *VLDB Endowment*, 2012.

[3] "OpenStack: Security Group Bypassing issue," https://bugs.launchpad.net/bugs/1359691.

[4] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *NSDI*. Usenix, 2007.

[5] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva, "On fault resilience of OpenStack," in *SOCC*. ACM, 2013.

[6] J. Mace, R. Roelke, and R. Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," in *SOSP*, 2015.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for Request Extraction and Workload Modelling." in *OSDI*. Usenix, 2004.

[8] J. L. Hellerstein, M. M. Maccabee, W. N. Mills, and J. J. Turek, "ETE: a customizable approach to measuring end-to-end response times and their components in distributed systems," in *ICDCS*. IEEE, 1999.

[9] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *DBPL*. ACM, 2015.

[10] H. Baek, A. Srivastava, and J. Van der Merwe, "Demo: CloudSight: A Tenant-oriented Transparency Framework for Cross-layer Cloud Troubleshooting," *SC*, 2016. [Online]. Available: http://www.flux.utah.edu/project/tcloud

[11] "OpenStack: Port Disappearance Issue," https://bugs.launchpad.net/bugs/1158684.

[12] "Openstack: Migration issue with scheduler hints," https://bugs.launchpad.net/bugs/1039065.

[13] "AWS CloudWatch - Cloud & Network Monitoring Services," http://aws.amazon.com/cloudwatch.

[14] "OpenStack Ceilometer," https://wiki.openstack.org/wiki/Ceilometer.

[15] P. Sharma, S. Chatterjee, and D. Sharma, "CloudView: Enabling tenants to monitor and control their cloud instantiations," in *IM*. IEEE, 2013.

[16] W. Wu, G. Wang, A. Akella, and A. Shaikh, "Virtual network diagnosis as a service." ACM, 2013.

[17] "AWS CloudTrail," http://aws.amazon.com/cloudtrail/.

[18] Y. Wencheng and Z. Yian, "An Autonomic Capacity Management Approach with Cloud Insight towards Cost-Efficient Throughput Optimization for High Performance Computing," in *CSNT*. IEEE, 2012.

[19] D. Sharma, R. Poddar, K. Mahajan, M. Dhawan, and V. Mann, "HANSEL : Diagnosing Faults in OpenStack," in *CoNEXT*. ACM, 2015.

[20] Y. Xu, Y. Liu, R. Singh, and S. Tao, "Identifying SDN state inconsistency in OpenStack," in *SOSR*. ACM, 2015.

[21] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang, "Auditing security compliance of the virtualized infrastructure in the cloud: Application to openstack," in *CODASPY*. ACM, 2016.

[22] Y. Xiang, H. Li, S. Wang, C. P. Chen, and W. Xu, "Debugging openstack problems using a state graph approach," in *APSys*. ACM, 2016.