An Architecture for Evaluating Distributed Application Deployments in Multi-Clouds

Antonis Papaioannou
Institute of Computer Science (ICS)
Foundation for Research and Technology–Hellas (FORTH)
Heraklion GR-70013, Greece
Email: papaioan(at)ics.forth.gr

Kostas Magoutis
Institute of Computer Science (ICS)
Foundation for Research and Technology–Hellas (FORTH)
Heraklion GR-70013, Greece
Email: magoutis(at)ics.forth.gr

***Abstract*—In this paper we present an architecture for the modeling, collection, and evaluation of long-term histories of deployments of distributed multi-tier applications on federations of Clouds (Multi-Clouds). Our goal is to capture several aspects of application development and deployment lifecycle, including the evolving application structure, requirements, goals, and service-level objectives; application deployment descriptions; runtime monitoring, and quality control; Cloud provider characteristics; and to provide a Cloud-independent resource classification scheme that is a key to reasoning about Multi-Cloud deployments of complex large-scale applications. Since our target is capturing the continuous evolution of applications and their deployments over time, we ensure that our metadata model is designed to optimize space usage. Additionally, we demonstrate that using the model and data collections over varying deployments of an application (using the SPEC jEnterprise2010 distributed benchmark as a case study) one can answer important questions about which deployment options work best in terms of performance, reliability, cost, and combinations thereof.**

I. Introduction

The problem of evaluating a multitude of possible deployments of complex distributed applications on heterogeneous infrastructrures with the aim of selecting feasible or even optimal ones dates from the early days of data centers [1]. Solutions to this problem often rely to either performance prediction models [2], simulations, or a limited set of experiments [3] to estimate performance of the underlying infrastructure under different load assignments to it. The problem of how to systematically explore a large collection of past execution histories of a multitude of real deployments of applications is less well studied. Such a facility has the potential of answering a variety of key questions about the runtime behavior of different deployments of complex applications, such as: which infrastructure (Cloud providers and/or types of resources) works best for certain applications? what are the most cost-effective options between a variety of configurations I (or my user community) have tried in the past? What rules or policies have been effective in achieving goals (or equivalently in addressing runtime issues) during past executions?

Integration of application modeling and deployment/operations environments has been gaining traction recently. For instance, the popular GitHub code repository supports adding a Cloud provider as a *remote* application repository, on which applications can be pushed for deployment. An interoperable Cloud provider (such as Heroku) will accept the push and receive the expected directories and files, deploying the application. Bringing together developer and operation teams is the goal of a new wave of *DevOps* platforms such as Chef [4] and Puppet [5]. What is missing from such environments is a well-integrated feedback loop including monitoring information, SLA assessments, etc., support for storage and analysis of potentially vast histories of past executions, and awareness of Multi-Cloud deployments. Support for such an integrated loop is a key goal of the architecture presented in this paper.

Handling heterogeneity in the infrastructure has been a challenging undertaking since the early days of data centers. A variety of hardware vintages and suppliers selected to improve cost-efficiency over time complicates the resource picture. The state of things in the Cloud space today is not much different: heterogeneity is ever present in the form of different types of resources offered from different Cloud providers. What is more, virtual resources carry nominal or indicative characterisations of their capabilities, making comparison inaccurate. To improve on the current state of affairs, we propose and implement a Cloud-independent classification scheme where virtual machine (VM) types are grouped with similarly-rated types across Cloud providers, according to a specific dimension such as CPU, memory, or I/O capability. The VM ratings are computed using a vector-driven approach taking into account individual micro-benchmarks on the VMs. Rating is sampled across regions and periodically repeated to capture variations and changes over time in the underlying hardware of the Cloud provider. Classification may also be repeated at different times to apply different criteria on the VM ratings (such as how many classes to group VMs into).

The architecture presented here combines a unique set of features not found in existing systems. It shares the principle of modeling application structure and deployed resources with systems such as SmartFrog [6], CloudML [7], TOSCA [8], models@runtime [9] and Cloudify [10]. It features detailed component-based monitoring of application performance typically found in Application Performance Management systems [11] and products such as IBM Tivoli Composite Application Manager. While addressing important problems in an application's lifecycle (such as determining problem root-causes) such systems do not offer a way to benefit from a potentially vast past experience in order to improve future deployments. Mining past histories to gain knowledge in the form of determining rules or detecting and ranking anomalies has been applied in the case of data center event

collections [12], [13], in discovering configuration errors [14], [15], and in modeling performance characteristics of desktop applications [16]. Our approach extends to complex distributed applications and covers deployment and lifecycle aspects such as elasticity and resource classification that are relevant to Multi-Cloud setups.

II. Architecture

Figure 1 depicts the architecture of our system. The purpose of the Classifier component (described in detail in Section III) is to assess and classify periodically the resources offered by different Cloud providers in a Cloud-independent manner. It subsequently updates the corresponding information in our metadata model described in Section IV. The Explorer/Analyzer component has two major goals. First, to explore the space of application configurations and deployment possibilities based on certain criteria. Second, to perform analyses of historical data from past executions in order to mine information about performance, cost, etc. The Explorer/Analyzer is aware of the structure of applications. The Explorer part decides on which VMs the application components should be deployed, based on certain requirements (e.g the J2EEapplication server component of the application should be deployed on a VM with CPU capability labeled - generically, and in a provider-independent manner– as LARGE). The Explorer stores these deployment plans as well as the SLA requirements and the elasticity rules of the application (if any) in the metadata model.

Clearly exploration can be a time-consuming task (since deployment and execution of each run can take hours or days, or even longer), so we assume that the exploration is a background activity that grows the historical database over time, rather than issued as a response to current time-sensitive queries. Optionally, part of the input to the historical database can be coming from a user community willing to share their deployment histories in *open data repositories* such as the Stanford Large Network Dataset Collection (SNAP) [17]. The Analyzer part can pose a variety of queries to the database to perform analytics, such as comparing the performance of an application across different deployments to determine the best performing or most cost-effective configuration settings.

III. Cloud-independent classification

A. Rank resources

To rank Cloud resources in a Cloud-independent manner we first create a profile for every VM type offered by various Cloud providers that describes the performance capabilities of the VM. In this way we can categorize resources to different classes of service, such as SMALL, MEDIUM, LARGE, etc. The VM profile is based on a vector of performance metrics focusing on three areas: CPU performance, memory size, I/O throughput. Our rationale in using a vector of performance metrics rather than a single benchmark was to simultaneously take into account multiple aspects of performance in ranking VMs. We use *k-means* cluster analysis [18] to classify resources based on benchmark results for each VM aspect. K-means takes as input the desired number of clusters to group VMs in and performs the classification automatically.

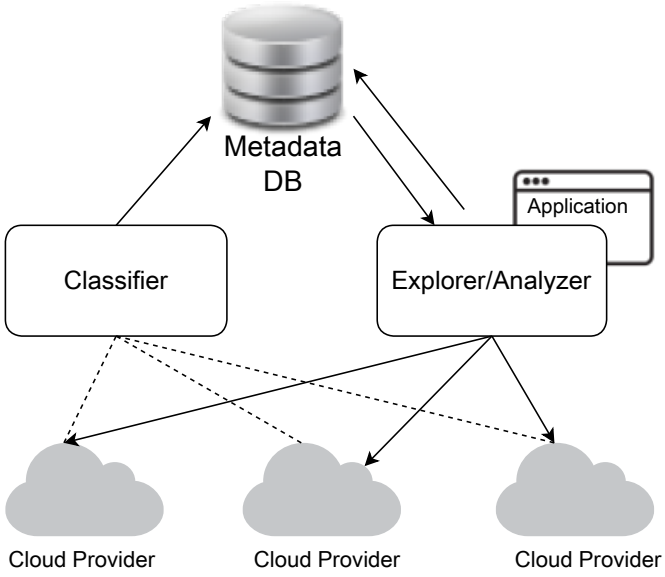

Fig. 1: System architecture

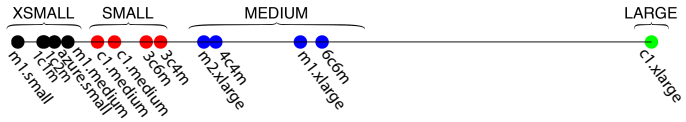

Fig. 2: CPU performance classification of different VM types

Our experience showed that existing benchmark suites such as SPEC CPU2006 [19] satisfy our objectives, thus we rely on it for CPU performance characterization. To factor-in the benefits of core parallelism we use the *rate* metric of SPEC CPU2006 that measures aggregate throughput. Classification can be optionally refined by normalizing by the cost of resources. Figure 2 presents the *k-means* grouping of 14 VM types from three Cloud providers (Amazon EC2, Azure, Flexiant) based on CPU performance. We use the standard naming scheme for Amazon VM types. We developed our own naming scheme for Flexiant VMs based on two numbers: the first denotes number of cores and the second memory size (GB); e.g: 3c6m stands for 3 cores, 6GB memory. Our Azure academic account provided us with access to a single VM type, which we refer to as *azure.small* (1 core, 1.75GB). Our *k-means* analysis classifies the 14 VM types into four groups: XSMALL, SMALL, MEDIUM, LARGE. Amazon's *c1.xlarge* is alone in the LARGE class, whereas the other classes comprise 4-5 VM types each. A common theme in the XSMALL group is that all VMs in it have a single virtual core.

Classifying VM types based on memory size can be performed via the *k-means* algorithm, just as in the case of CPU performance. Figure 3 depicts three clusters SMALL, MEDIUM, and LARGE, created for the 14 VM types used in the previous case. Notice that the number of clusters created to classify VMs in each of the dimensions (CPU, memory, I/O) can vary.

Classifying VM types based on storage performance must ensure that all storage options offer the same consistency

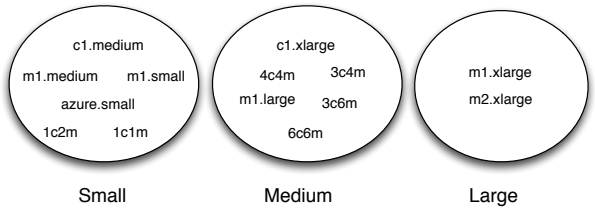

Fig. 3: Memory size classification of different VM types

Provider	VM	T/put (MB/s)	Dev (%)	Avg (MB/s)	Class
Amazon	c1.medium	109.1	3.0	102	LARGE
	c1.xlarge	95.8	3.3		
	m1.small	78.7	5.9	73	MEDIUM
	m1.medium	70.9	10.7		
	m1.large	68.7	22.0		
Flexiant	1c1m	38.4	15.3	41	SMALL
	1c2m	40.5	13.5		
	3c4m	39.1	21.2		
	3c6m	41.1	16.7		
	4c4m	38.4	14.6		
	6c6m	45.0	6.3		
Azure	small	35.3	4.0	35	SMALL

TABLE I: Disk throughput measurements and classification

semantics. To reflect industry standard practices, we chose to evaluate VMs with remotely mounted storage (without optimizations) across all providers. We used the *hdparm* benchmark to measure average disk throughput and standard deviation over ten trials. We considered storage performance of the Amazon, Azure, and Flexiant FCO VM types shown in Table I. I/O throughput to network storage seems to fall into one or two narrow bands within each Cloud provider. It thus seems to depend more on the type of storage rather than type of VM (although smaller VMs spend a higher CPU share to do I/O at full speed). An exception is Amazon EC2 where *c1* instances seem to have access to higher-performing storage compared to *m1* instances. Table I shows the classification of the VMs into SMALL, MEDIUM, LARGE. Finally we note that we consider network performance primarily a characteristic of the Cloud provider (previous work evaluated providers based on their network performance, [20] Table 1) and to a lower extent a characteristic of a VM type, we thus treat it separately from the other three dimensions (CPU, memory, storage).

IV. Metadata model

In this section we describe the metadata model at the core of our architecture (Figure 1). The model (whose schema in standard E/R notation is shown in Figure 4) is meant to capture the description of an application, its requirements and goals, rules and policies, and its provisioned resources, as well as runtime aspects of its execution histories such as monitoring information at different levels, invocations of rules and policies, and quality of service assessments. The model also captures Cloud provider characteristics, platforms, as well as users, roles, and organizations.

The system described in this paper is meant for long-term preservation for information. It is designed to associate mutations with a wall-clock timestamp and to trace the identity of the sources of mutations. It thus shares principles with archival systems [21], temporal databases [22], and provenance systems [23]. The information schema describes the applications and their deployment using principles from specifications such as CloudML [7], PIM4Cloud [24], and TOSCA [8]. The exposition of the metamodel provided here is necessarily concise and lacking detail, as an exhaustive presentation would require far more space than we have available in this paper.

A version of an application is rooted at an APPLICATION object and comprises software ARTEFACT and ARTEFACT INSTANCE objects. An ARTEFACT INSTANCE can be deployed either on another ARTEFACT INSTANCE object or on a NODE INSTANCE object, which represents a VM resource. The deployment relationship is a *temporal assocation* represented by a DEPLOYMENT ASSOCIATION object (with a start and end time). Requirements and goals are represented by ROOT_SLO (expresses non-IT constraints such as overall cost, location, etc.), IT_SLO (expresses a requirement on an IT metric [25]), and AFFINITY_GOAL (expresses dependencies between artifacts) objects. These requirements are connected to monitoring information represented by APPLICATION_MONITOR, ARTEFACT_MONITOR, RESOURCE_MONITOR, and RESOURCE_COUPLING_MONITOR. Each monitor relates to the metric specified in the corresponding objective. Rules or policies (such as ELASTICITY_RULE, connecting to an IT_SLO) are captured in the metamodel.

Information about each execution of an application is rooted at an EXECUTION_CONTEXT (with a start and end time) and SLO_ASSESSMENT (an evaluation of the degree to which an SLO was achieved) objects. Part of the execution history (beyond the monitoring info) are ELASTICITY_ACTIONs, which are recorded firings of rules in response to certain conditions. Each NODE INSTANCE is of a particular CD_VM_TYPE and CI_CM_TYPE where CD stands for *cloud dependent* and CI for *cloud independent*. A CD_VM_TYPE describes a real-world VM type offered by a Cloud provider. CI_VM_TYPEs are the result of (periodic) classifications described in Section III. Cloud providers are described in CLOUD_PROVIDER objects and their offered platforms described in PLATFORM_AS_SERVICE objects. Finally, users, roles, and organizations connected to the modeled entities are described in USERS, ORGANIZATION, and ROLES objects [26].

V. Evaluation

We evaluate our architecture using the distributed SPEC jEnterprise2010 benchmark [27] as a case study. To create a rich history of executions we deploy SPEC jEnterprise2010 to different Cloud providers under different deployment plans. SPEC jEnterprise2010 is a full system benchmark that allows performance measurement and characterization of Java EE 5.0 servers and supporting infrastructure. The benchmark models supply a chain consisting of an automobile manufacturer (referred to as the Manufacturing Domain) and automobile dealers (referred to as the Dealer Domain). The Web-based interface between the manufacturer and dealers supports browsing a catalog of automobiles, placing orders, and indicating when inventories have been sold. The SPEC jEnterprise2010 application requires a Java EE 5.0 application server and a relational database management system (RDBMS), which comprise the *system under test* (SUT).

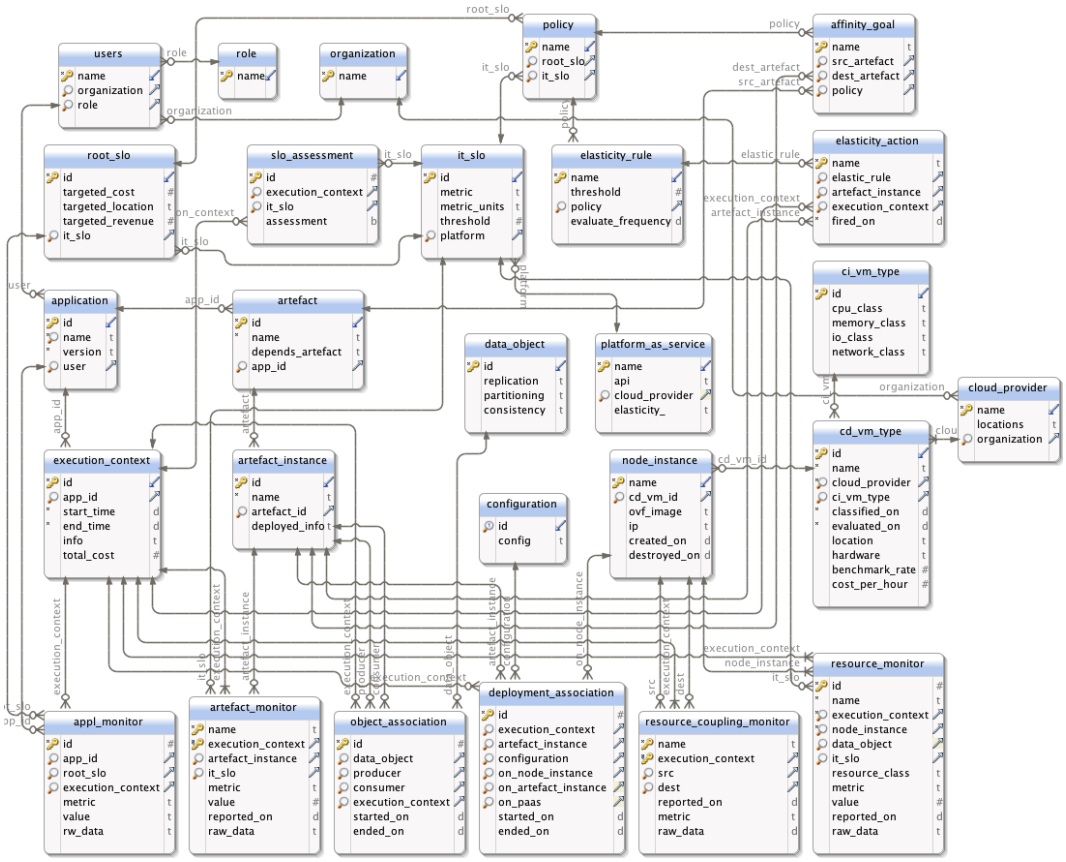

Fig. 4: Metadata model

The primary metric of the SPECjEnterprise2010 benchmark is throughput measured as jEnterprise operations per second (EjOPS). A load generator (referred to as a Driver) produces a mix of browse, manage, and purchase business transactions at the targeted injection rate (or txRate, equal to the number of simulated clients divided by 10), which aims to produce the targeted EjOPS. The Driver measures and records the response time (RT) of the different types of business transactions. Failed transactions in the measurement interval are not included in the reported results. At least 90% of the business transactions of each type must have a RT of less than the constraint threshold (set to 2 seconds for each transaction type). The average RT of each transaction type must not exceed the recorded 90^{th} percentile RT by more than 0.1 seconds. This requirement ensures that all users see reasonable response times. The Driver checks and reports on whether the response time requirements are met during a run.

We model the SPEC jEnterprise2010 application using three artefacts (Section IV) corresponding to the business logic of the application, the application server, and the RDBMS. These artefacts are instantiated as *specj.ear* and *emulator.war* files, a JBoss 6.0 application server, and a MySQL 5.5 and their corresponding node instance are associated via the DEPLOYMENT_INSTANCES table, as shown in Figure 5. The IT_SLO on response time is set to 2 sec. Our deployment plans consider the deployment of the application and database artefact instances on the same node as well as on separate node instances (i.e., same VM vs. different VMs) on one or more providers. The back-end database was populated for a targeted dataset of up to 800 clients. For every execution of the benchmark we use its EjOPS metric and the reported RTs to assess the SLO requirements of the application during the run. Simultaneously, we monitor the resource usage (CPU and disk I/O) of the node instances used. We store the raw data in a time-series database and periodic averages into the metadata database. The latter is implemented on MySQL 5.1 RDBMS hosted on a Flexiant *2c4m*-type VM.

A. Typical use cases

Table II describes selected deployments of SPEC jEnterprise2010 on different VM types from Amazon and Flexiant chosen by the Explorer and/or contributed by the user community. All deployment scenarios considered here use an injection rate of 200 clients (TxRate 20). A specific use-case of our system is to mine past executions over a user-defined time interval, select those that satisfied their SLOs, and between those select the most cost-effective configuration options. The set of those configurations can be determined by executing the following SQL query to our metadata database.

```
SELECT arm.execution_context, arm.value, ec2.
    total_cost, (arm.value/ec2.total_cost) AS ratio
FROM artefact_monitor arm, execution_context ec2
WHERE arm.name='EjOPS' AND ec2.id=arm.
    execution_context AND arm.execution_context IN
   (SELECT ec.id
    FROM execution_context ec
    WHERE app_id=1 AND ec.id NOT IN
```

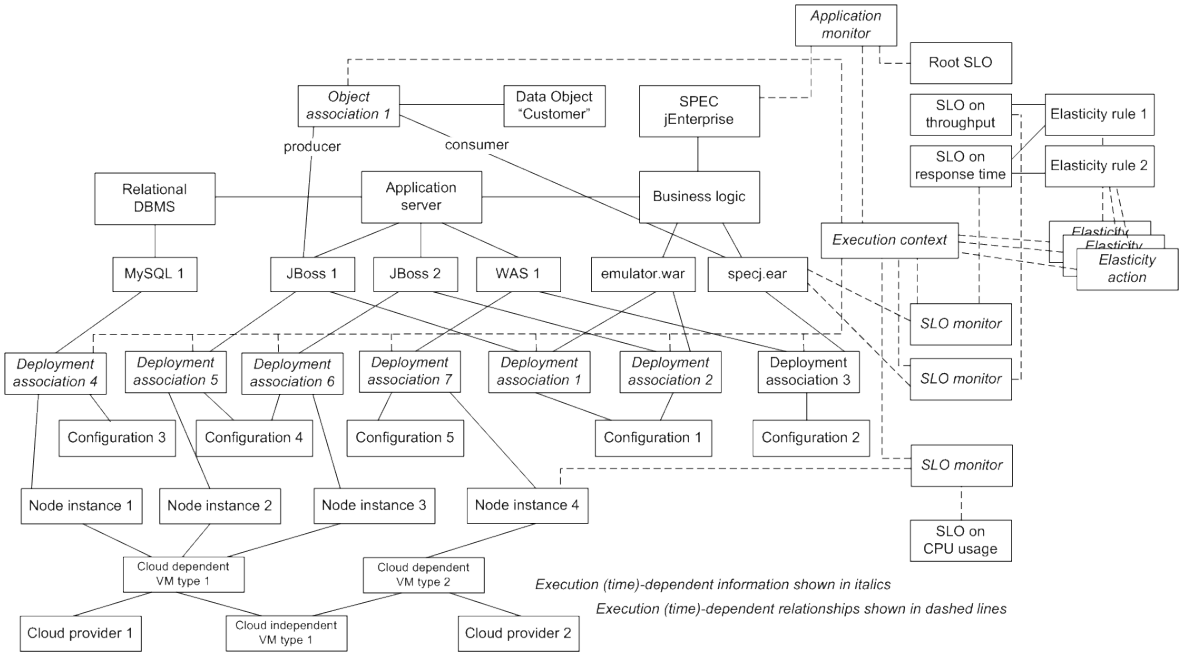

Fig. 5: Deployment and execution model of SPEC jEnterprise2010

Deployment Configuration	Artefact	Config Settings	VM type	Provider
1	App Server DB	Heap Size 2048MB Buffer Size 128MB	m1.large	Amazon
2	App Server DB	Heap Size 2048MB Buffer Size 2000MB	m1.large	Amazon
3	App Server DB	Heap Size 128MB Buffer Size 768MB	c1.medium	Amazon
4	App Server DB	Heap Size 768MB Buffer Size 128MB	m1.small	Amazon
5	App Server DB	Heap Size 2048MB Buffer Size 1100MB	3c4m	Flexiant
6	App Server DB	Heap Size 768MB Buffer Size 128MB	1c2m	Flexiant
7	App Server DB	Heap Size 768MB Buffer Size 128MB	small	Azure
8	App Server DB	Heap Size 6000MB Buffer Size 2000MB	m1.large m1.large	Amazon Amazon
9	App Server DB	Heap Size 768MB Buffer Size 2000MB	c1.medium m1.large	Amazon Amazon
10	App Server DB	Heap Size 768MB Buffer Size 2000MB	m1.small m1.large	Amazon Amazon
11	App Server DB	Heap Size 2048 MB Buffer Size 1100	3c4m m1.large	Flexiant Amazon

TABLE II: Selected SPEC jEnterprise2010 deployment plans

```
    (SELECT DISTINCT(asses.execution_context)
     FROM slo_assessment asses
     WHERE asses.assessment=FALSE)
)
ORDER BY ratio DESC
```

The results of this query are summarized in Table III. The execution offering the best absolute performance costs $0.385/hour whereas the most cost-effective deployment costs $0.145/hour (VM time is the only billable metric in our experiments). By selecting the most cost-effective rather than the best-performing selection leads to saving $0.24/hour or about $2,100 per year while still achieving SLOs.

Another use-case of our system is to determine typical causes for suboptimal performance. For example, in executions whose SLOs are violated, a look into resource usage (e.g., CPU%) may point to the root cause. For example the results of the following query reveal that a saturated CPU (underprovisioning of some deployable artefact) is often an issue.

```
SELECT *
FROM resource_monitor rm
WHERE rm.name='cpu_idle' AND
    rm.execution_context IN
      (SELECT ec.id
       FROM execution_context ec
       WHERE app_id=1 AND ec.id IN (
          SELECT DISTINCT(asses.execution_context)
          FROM slo_assessment asses
```

Deployment Conguration	EjOPS	Cost ($)	Ratio
3	20.272	0.145	139
5	20.228	0.176	114
1	20.135	0.24	83.89
2	20.123	0.24	83.84
9	20.273	0.385	52.65
8	20.239	0.48	42.16

TABLE III: Cost-effectiveness for successful executions (Ratio is price-normalized performance)

```
        WHERE asses.assessment=FALSE)
    )
```

For example, in deployments where multiple VMs are used (e.g., Execution 9) we determine that the problem is using an Amazon *m1.small* VM for hosting the application server. The other VM, an *m1.large* hosting the database server, seems to be overprovisioned with its CPU always under 10% busy. A further look to the results of this query reveals that all deployments that involve any VM from the XSMALL cluster (regardless of Cloud provider) consistently lead to SLO violations. Therefore a deployment specialist would be advised to avoid VM types classified in that cluster in future deployments of the application. Another jEnterprise2010 configuration failing its SLO is #10. This is a case of a multi-Cloud deployment involving resources that, while adequate in a single-Cloud scenario, their tight coupling via frequent communication is expensive (highly latency, high cost) in a multi-Cloud scenario. The coupling can be determined through information in the RESOURCE_COUPLING_MONITOR object.

Another use of our system is in reasoning about variations in the service quality of Cloud providers over time. Such variations can be due to different levels of contention for hypervisor resources at different times or days of the week, month, or year. Another reason is the existence of different hardware infrastructures (such as CPU, network, and disk generations) in data centers (often within the same data center) of Cloud providers, as has been previously observed [28]. Figure 6 shows the results from collecting (in APPL_MONITOR and ARTIFACT_MONITOR objects) response-time measurements for SPEC jEnterprise purchase transactions over a 6-day period.

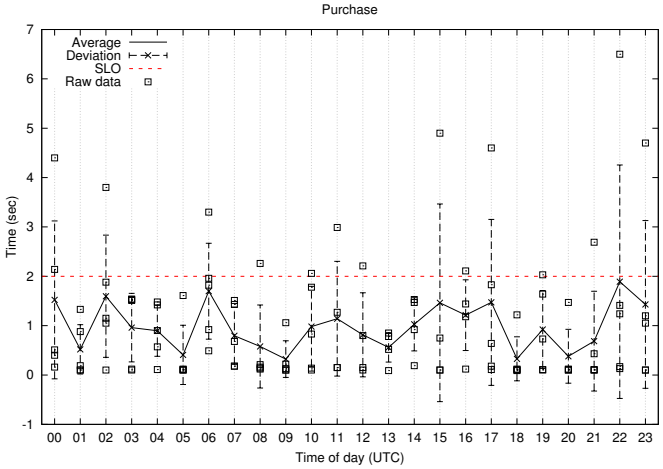

Fig. 6: Response time of purchase transations for hourly executions of jEnterprise2010 over a period of six days

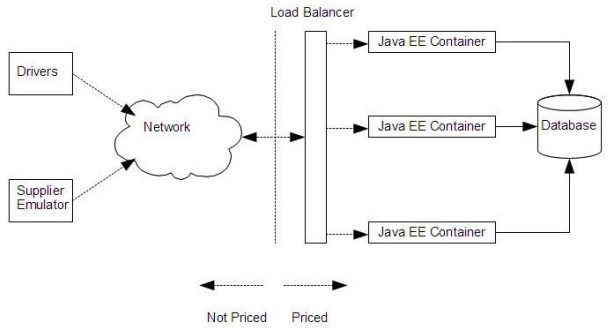

Fig. 7: SPEC jEnterprise2010 with elasticity [27]

B. Analysis of application elasticity actions

In this section we demonstrate how the Explorer can analyze the effectiveness of elasticity rules on application performance under different scenarios. Since SPEC jEnterprise2010 does not offer elasticity features in its standard setup, we modified it to place a load balancer (LB) frontend for distributing the load between several application servers as shown in Figure 7. Initially the deployment includes the LB, one VM hosting an application server, and another VM hosting the back-end database. The elasticity rule is triggered when the (periodically monitored) response time exceeds a SLO threshold (set by the benchmark at 2 sec); the action is to start a new application server VM and add it to the LB list.

During each run of our elastic SPEC jEnterprise2010 we collect all monitored information in our metadata database. The artifact and resource monitors report response time and CPU utilization every 20 seconds. Figure 8 shows the results of a run where the response-time SLO is initially violated due to an overloaded application server CPU. In this run, the load is set to 150 clients, the application server is hosted on an EC2 *m1.small* instance and the back-end database servers is hosted on a *m1.medium* VM instance. After 60 sec, the elasticity rule triggers an elasticity action (adding a second application server on a new *m1.small* VM), eventually reducing response time to acceptable levels. Looking at CPU figures at the bottom of Figure 8 we can identify that CPU overload is indeed the cause of the performance bottleneck in this case.

In a different situation, such an elasticity action may not be effective. Figure 9 depicts performance of a deployment where an application server is deployed on a *m1.large* instance whereas its back-end database is hosted on a *m1.small* VM. The benchmark simulates the load of 350 clients. In this case,

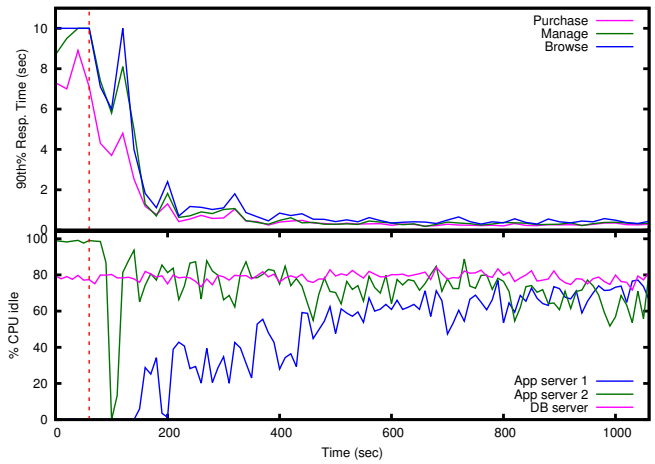

Fig. 8: Successful invocation of elasticity rule

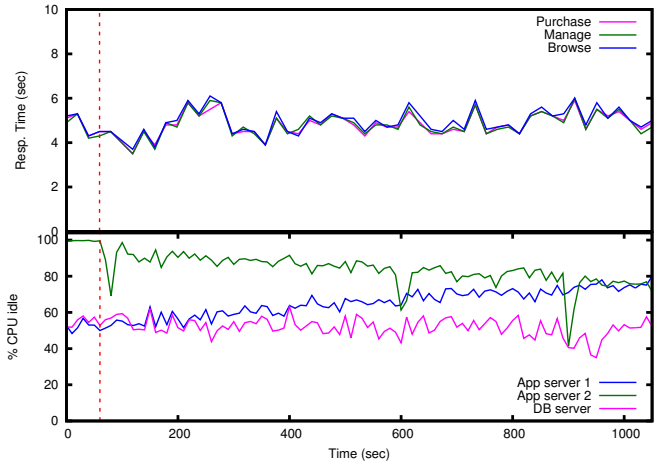

Fig. 9: Unsuccessful invocation of elasticity rule

response time is unaffected by the elasticity action. A closer look at resource monitor data shows that CPU utilization was high but not a critical factor at neither the application servers or the database. In this case, network and I/O performance of the *m1.small* VM instance hosting the database back-end is the likely limitation.

C. Managing database evolution over time

Our metadata schema (Figure 4) was designed to avoid redundancy when recording time-evolving state. They key to achieving this is to explicitly represent time-dependent associations, for example that of an application artefact with the resources used to deploy it in each execution of the application. If we recorded the time of deployment of every artefact within the ARTEFACT_INSTANCE table, we would need to create a new artefact instance for every execution of the application even though no other aspect of the artefact instance changed across executions. This design ensures that our metadata database grows at the reasonable pace. Similarly, we use the table ARTEFACT_CONFIG to correlate the configuration parameters of an artefact instance within an execution context.

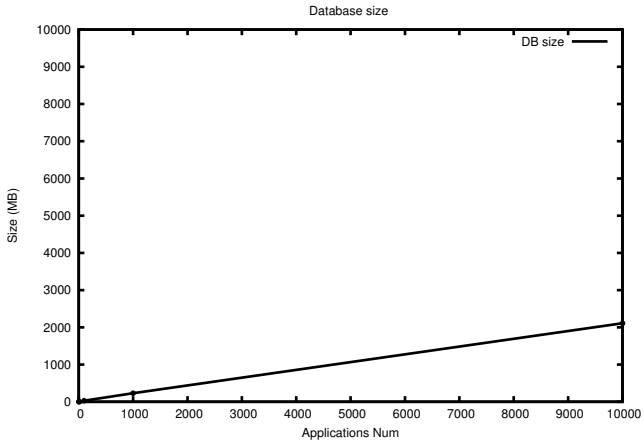

Fig. 10: Growth of metadata database size with increasing number of applications

To get a feeling of the rate of growth for the physical size of the metadata database in typical use cases, suppose the Explorer plans to deploy the SPEC jEnterprise application on 8 different node instances (VM types) across 3 Cloud providers. The Explorer further wants to try combinations of artefact configuration settings for each node instance. Examples of important settings for the JBoss application server and the MySQL RDBMS are JVM heap size and database buffer size. Trying 3 different values for each setting means the Explorer needs to execute the application at least 9 times on each VM type to cover the space of possible combinations. This would result into 72 executions of the application.

After the first execution the metadata database takes up 1.25 MB of storage, increasing to 1.31 MB after all 72 executions are in. Inserting data for another set of 72 runs of a second application (similar to the first) increases the size of the database to 1.64 MB. Repeating for up to 100 applications (similar to the first application) increases the size to 29.3 MB. All applications are distinct and therefore they do not share artefacts, deployment instances (VMs), SLAs, monitors etc.

Taking into account the duration of each execution of SPEC jEnterprise2010 (one hour in our case) means that we will need at least 72 hours to test the application for the 72 runs we described above. When 100 similar applications simultaneously populate the metadata database, the storage growth would be about 30MB every three days. This equals 3.6 GB per year or 36 GB in 10 years, which is a modest requirement. Figure 10 depicts the storage growth trend as the number of applications increases. The execution time of queries in the metadata database increases with database size: The first query described previously takes on average 0.036, 0.384 and 3.55 seconds as the database size grows from 100, to 1,000 and 10,000 applications respectively.

Note that this analysis does not take into account the raw monitor data that are typically stored in a time series database (TSDB). Only summaries (e.g., average values) are periodically stored in the metadata database. In a typical setting, the raw data would be erased over time while the metadata database information would be preserved.

VI. Related work

Previous approaches to observing system executions over time for the purpose of understanding application characteristics include PeerPressure [14], Clarify [15], and AppModel [16]. All three systems leverage deployed machines for statistical analysis. PeerPressure and Clarify aim at troubleshooting computer miscongurations and errors. AppModel focuses on performance modeling, which has the additional complication that performance is time-varying and depends on interactions among many congurations (hardware and software) as well as workload characteristics. Westermann et al. [29] relate to AppModel in their use of extensive performance measurements to derive application performance models; however their work does not rely on the collection of results from a large base of existing installations.

Our work is related to PeerPressure, Clarify, and AppModel in that we leverage a large history of application configurations in distributed environments for mining knowledge about best practices and application behavior. Although performance modeling could fit into the explorer/analyzer part and layered over our metadata DB, its full exploration is a topic of future work. The design of our metadata DB is geared towards distributed multi-cloud applications rather than single-user desktop-type applications as in the case of AppModel.

VII. Conclusions

In this paper we outlined an architecture for evaluating distributed application deployments in Multi-Clouds. The information metamodel at the core of the architecture captures the history and evolution of distributed application deployments and can support a variety of interesting analytics. The proposed classification scheme is critical in enabling a cross-Cloud categorization of resources. Our evaluation using the SPEC jEnterprise2010 application benchmark exhibits the use of our system in discovering cost-effective deployment plans, and in reasoning about elasticity policies under different assumptions. We believe that the applicability of the architecture is broader and plan to exhibiting it further in our future work.

Acknowledgment

We thankfully acknowledge the support of the PaaSage (FP7-317715) EU project. We also acknowledge the valuable feedback of Kyriakos Kritikos in the metadata schema design.

References

[1] J. Wolf, "The placement optimization program: a practical solution to the disk file assignment problem," in *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1989.

[2] R. R. Steffen Becker, Heiko Koziolek, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22.

[3] J. Tordsson et al., "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Generation Computer Systems*, vol. 28, no. 2, pp. 358–367, Feb. 2012.

[4] "Opscode Chef Automation Platform," http://www.opscode.com/chef, Accessed 6/2013.

[5] "Puppet Labs DevOps Platform," https://puppetlabs.com/solutions/devops/, Accessed 6/2013.

[6] P. Goldsack et al., "The smartfrog configuration management framework," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 16–25, Jan. 2009.

[7] N. Ferry et al., "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems," in *Proc. of the 2013 IEEE Sixth International Conference on Cloud Computing*, ser. CLOUD '13. Washington, DC, USA: IEEE Computer Society, 2013.

[8] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, "Portable cloud services using tosca," *IEEE Internet Computing*, vol. 16, no. 3, pp. 80–85, May 2012.

[9] U. Aßmann, N. Bencome, B. H. C. Cheng, and R. B. France, "Models@run.time (dagstuhl seminar 11481)," Dagstuhl Reports, Tech. Rep. 11, 2011.

[10] "Cloudify," http://www.cloudifysource.org/.

[11] G. Khanna, K. Beaty, G. Kar, and A. Kochut, "Application performance management in virtualized server environments," in *Network Operations and Management Symposium, NOMS 2006. 10th IEEE/IFIP*, 2006.

[12] J. L. Hellerstein, S. Ma, and C.-S. Perng, "Discovering actionable patterns in event data," *IBM Systems Journal*, vol. 41, no. 3.

[13] K. Viswanathan et al., "Ranking anomalies in data centers," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012.

[14] H. J. Wang et al., "Automatic misconfiguration troubleshooting with peerpressure," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004.

[15] J. Ha et al., "Improved error reporting for software that uses black-box components," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007.

[16] Eno Thereska, Bjoern Doebel, Alice X. Zheng, Peter Nobel, "Practical Performance Models for Complex, Popular Applications," in *SIGMETRICS'10*, June 14-18, 2010.

[17] "Stanford Large Network Dataset Collection," http://snap.stanford.edu/data/, Accessed 6/2013.

[18] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observation," in *Proc. of 5th Berkeley Symposium on Mathematical Statistics and Probability*. Univ. of California Press, 1967.

[19] "SPEC CPU2006 Benchmark," http://www.spec.org/cpu2006/, Accessed 6/2013.

[20] I. Kitsos, A. Papaioannou, N. Tsikoudis, and K. Magoutis, "Adapting data-intensive workloads to generic allocation policies in cloud infrastructures," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012.

[21] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in *Proc. of the 1st USENIX conference on File and storage technologies*, ser. FAST'02. Berkeley, CA, USA: USENIX Association, 2002.

[22] R. T. Snodgrass, *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Series in Data Management Systems.

[23] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006.

[24] "REMICS Deliverable D4.1: PIM4Cloud," http://www.remics.eu/system/files/REMICS_D4.1_V2.0_LowResolution.pdf, 2012.

[25] P. Bianco, G. Lewis, and P. Merson, "Service level agreements in service-oriented architecture environments," Software Engineering Institute, Tech. Rep. CMU/SEI-2008-TN-021, September 2008.

[26] "CERIF metadata model," Accessed 6/2013. [Online]. Available: http://www.eurocris.org/Index.php?page=featuresCERIF&t=1

[27] "SPEC jEnterprise2010 Benchmark," Accessed 6/2013. [Online]. Available: http://www.spec.org/jEnterprise2010/

[28] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, Sep. 2010.

[29] D. Westermann et al., "The Performance Cockpit Approach: A Framework For Systematic Performance Evaluations," in *Proceedings of 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Lille, France, September 2010.