Axe: A Novel Approach for Generic, Flexible, and Comprehensive Monitoring and Adaptation of Cross-Cloud Applications

Jörg Domaschka, Daniel Seybold, Frank Griesinger, and Daniel Baur

University of Ulm, Institute of Information Resource Management,
Albert-Einstein-Allee 43, 89081 Ulm, Germany
{joerg.domaschka,daniel.seybold,
frank.griesinger,daniel.baur}@uni-ulm.de
http://www.uni-ulm.de/in/omi

Abstract. The vendor lock-in has been a major problem since cloud computing has evolved as on the one hand side hinders a quick transition between cloud providers and at the other hand side hinders an application deployment over various clouds at the same time (cross-cloud deployment). While the rise of cross-cloud deployment tools has to some extend limited the impact of vendor lock-in and given more freedom to operators, the fact that applications now are spread out over more than one cloud platform tremendously complicates matters: Either the operator has to interact with the interfaces of various cloud providers or he has to apply custom management tools. This is particularly true when it comes to the task of auto-scaling an application and adapting it to load changes. This paper introduces a novel approach to monitoring and adaptation management that is able to flexibly gather various monitoring data from virtual machines distributed across cloud providers, to dynamically aggregate the data in the cheapest possible manner, and finally, to evaluate the processed data in order to adapt the application according to user-defined rules.

1 Introduction

Since the beginning of cloud computing, vendor lock-in has been a major problem. It is still around mainly due to the fact that cloud standards such as CIMI [6] and OCCI [16] have not been widely adopted by cloud providers. Tools abstracting the differences between cloud providers, and thus allowing *multi-cloud* deployment—the capability to deploy one application at different cloud platforms using the same application specification—have been a first step to overcome vendor lock-in. Yet, it is only *cross-cloud deployment*—the capability to spread a single application instance across different cloud providers—that enables users to take full advantage of different providers and their capabilities. In particular, it enables trading off the properties of application requirements against the offerings on a per-component or even per-component instance basis. This for instance allows a *hybrid-cloud deployment* where a database containing

sensitive data is deployed in a private cloud, while the rest of the application resides in different public clouds.

Both approaches, multi-cloud and cross-cloud, give the application operator the chance to change its current application deployment and to adapt to changed conditions such as the workload, e.g., more load than originally anticipated, and changed environmental conditions, e.g., the prices of other operators have changed. In order to benefit from these features, however, the application operators need to be able to actively judge the quality of the current deployment. For pure multi-cloud systems the application operator may refer to the monitoring tools of the currently selected cloud operator. While basic monitoring data may come for free on some cloud providers, often the user needs advanced metrics that either cost (Amazon, Rackspace) or require him to set up own monitoring tools. In addition to that, he has to familiarise with the user interfaces of various cloud providers.

For cross-cloud deployment using the providers' monitoring infrastructure is technically feasible, but tremendously increases the effort as multiple tools have to be used in parallel. Moreover, it is difficult to access metrics that involve the crossing of provider domains (such as network traffic from provider A to provider B). Furthermore is hard to access application-specific or component-specific metrics. Also, a sophisticated and configurable aggregation on the metrics is currently not easily possible. Finally, while most cloud providers support a simple approach to auto-scaling for application adaptation, e.g. metrics-based scale-out, there is currently no built-in mechanism that supports a cross-cloud adaptation of applications.

In this paper, we address these issues by introducing Axe, a generic, flexible, and extensible monitoring and adaptation engine for cross-cloud deployments. Besides the fact, that we introduce the tool, our contributions are as follows: *(i)* We present a powerful API that enables the specification of rules independent of the concrete deployment. *(ii)* We discuss a heuristic of how to reduce the cross-cloud provider network traffic and hence reduce costs. *(iii)* We introduce the first engine to deal with the *Scalability Rule Language* (SRL)[8, 12]. All of the features are embedded in Cloudiator, our cross-cloud, multi-tenancy deployment and application management tool [2, 7].

This document is structured as follows: Section 2 introduces background on Cloudiator and *Scalability Rule Language (SRL)* and further defines requirements towards our approach. Section 3 introduces our approach by presenting the individual tools of our platform and their configuration. It also discusses architectural options and introduces our architecture as well as the API. Section 4 exhibits the current status and upcoming tasks. Section 5 discusses related work, before we conclude with a report on our current status and open issues.

2 Background

The design of Axe has been heavily driven by constraints of cross-cloud environments. In addition to that, Axe builds heavily on earlier work. In the following,

we first introduce the constraints and derive requirements from them. In the next step, we present our CLOUDIATOR tool that we use as the basis for the AXE implementation [7]. Finally, we roughly describe the Scalability Rule Language that constitutes the meta-model for our monitoring and scaling solution.

2.1 Requirements and Constraints

For supporting in-depth analyses of existing deployments, several requirements have to be considered: *(a)* The fact that on the one hand, the monitoring of large-scale applications does generate huge amounts of data and on the other hand cloud providers usually charge for network traffic that leaves their data centre gives motivation that as much of data processing shall happen within the domain of individual cloud providers. *(b)* In order to avoid single points of failures, the architecture of a monitoring solution should not rely on a centralised approach, but rather favour distributed approaches with no central entity. As the amount of monitoring data usually increases with the number of allocated *virtual machines (VMs)*, the resources assigned to monitoring shall increase with the size of the application. *(c)* The operators of a cloud application may discover that they have to monitor further high-level or even low-level metrics or need monitoring to happen at a higher resolution. Hence, it is necessary that monitoring properties can be changed also after an application has been deployed. *(d)* The same considerations that hold for monitoring, have to hold for scaling rules. In addition, it is necessary that rules can be defined in a generic way without having to know the exact number of instances per component in advance. *(e)* The monitoring platform has to be able to capture application-specific metrics.

2.2 The Cloudiator Tool

CLOUDIATOR[1] is a cross-cloud deployment tool that also supports adaptation and re-deployment. In this section, we present the CLOUDIATOR architecture to the extend necessary to understand how it embeds AXE. Figure 1 summarises the architecture as of the original CLOUDIATOR tool (green), but also the enhancements of AXE (yellow). The AXE specific components *Aggregation*, *Scaling Engine*, and *Visor* are introduced in detail in Section 3.

The figure shows that CLOUDIATOR consists of a *home domain* for which COLOSSEUM is the entry point offering a JSON-based REST interface. This is used by a graphical Web-based user interface, but can also be used by adapters and automatisation tools. It also comprises various registries that store the CLOUDIATOR users, information about cloud providers, the cloud accounts of the users, and meta-information about cloud offerings such as the operating systems of images. Moreover, the home domain contains a repository of application components together with their life-cycle handlers as well as applications composed of these components. In addition, the registries contain information about started VMs and the component instances deployed on them as well as about the

[1] `https://github.com/cloudiator`

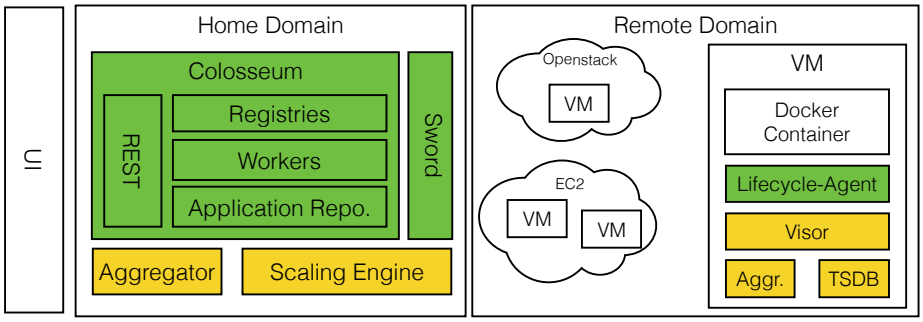

Fig. 1. The CLOUDIATOR architecture

wiring between the component instances. Finally, they hold the workers syncing the registries with the cloud provider information, and executing the provisioning of virtual machines or the installation of application components on virtual machines. The SWORD abstraction layer realises the communication with the various cloud provider APIs based on Apache jclouds[2].

The *remote domain* comprises all VMs at various cloud providers as well as the component instances running on them. In addition to that it contains CLOUDIATOR's life cycle agent on each of the VMs that the home domain uses in order to distribute component instances over VMs and to poll the status of the component instances when it needs to be shown in the user interface.

2.3 Scalability Rule Language

The SRL [8] is a provider-agnostic description language. It provides expressions to define the monitoring raw metric values from VMs and component instances and also mechanisms to compose higher-level metrics from raw metrics. Moreover, it comprises mechanisms to express events and event patterns on metrics and metric values. Finally, SRL captures thresholds on the events and actions to be executed when thresholds are violated. A simple SRL rule in prose may be *add a new instance of this distributed database if* (i) *all instances have a 5 minute average CPU load* $> 60\%$, (ii) *at least one instance has a 1 minute average CPU load* $> 85\%$, *and* (iii) *the total number of instances is* < 6.

3 Approach

This section sketches our approach in order to realise a flexible monitoring and adaptation tool that satisfies the requirements imposed on cross-cloud tooling (cf. Section 2). Basically, our auto-scaling process maps to the MAPE loop [15, 11] consisting of the following phases: monitoring, analysis, planning, and execution of changes. With respect to our setting, this means that first, we have to

[2] `https://jclouds.apache.org/`

retrieve monitoring data from the virtual machines and component instances. In a second step, the raw data gathered there has to be aggregated and processed. Third, the rule processing has to happen on the aggregated data and finally, the resulted rule has to be executed.

3.1 Visor: Gathering Monitoring Data

In order to be able to gather the raw monitoring data from the VMs and component instances, we introduce VISOR as a monitoring agent to the remote domain. Just as the life-cycle agent, VISOR is deployed on every VM and provides a remote interface the home domain uses in order to configure a particular VISOR instance. This allows VISOR to adopt to the application and to only collect the required metrics, thus saving space and bandwidth. VISOR supports the capturing of data on a per component instance basis as well as on a per-VM basis. The sooner is achieved by sensors monitoring basic system properties on virtual machine level, e.g. by accessing system properties such as CPU load. The latter is done by exploiting the fact that all component instances are run inside a Docker[3] container and the resource consumption can be retrieved on a per-container basis. By default, VISOR offers various sensors supporting system metrics such as CPU load, memory consumption, disk I/O, and network I/O.

In order to support custom metrics, VISOR supports the implementation of custom sensors, by providing an easy-to-implement Java interface. It exploits the dynamic class loading properties of Java in order to be able to add those implementations at runtime. For supporting application-specific metrics that can only be retrieved from within an application such as the length of queues or the degree to which buffers have been filled, VISOR offers a `telnet`-based interface where applications can push their metrics data to. This interface is compatible with the carbon daemon of graphite[4], thus allowing an easy migration to VISOR.

3.2 Buffering Monitoring Data

A key element when computing higher-level metrics especially over larger time-windows is the need to buffer raw monitoring data. *Time-series databases (TSDBs)* have been designed to store timestamped data in an efficient way and also to provide quick access to the stored data. Many TSDB implementations support applying functions on stored data right out of the box what makes them a perfect match not only for buffering, but also for aggregation (cf. Section 3.3). The following paragraphs first derive a strategy on how to implement buffering including the constraints and then compares TSDBs found in literature and the open source community with respect to the required properties.

[3] `http://www.docker.io`
[4] `http://graphite.readthedocs.org/en/latest/carbon-daemons.html`

Strategy With respect to our requirements (cf. Section 2) the buffering and therefore the TSDB approach needs to be able to work with limited resources, have no single point of failure, and increase available resources when more VMs are being used. In order to cope with these requirements, we use the following approach: from each VM acquired for an application, we reserve a configurable amount of memory and storage (e.g. 10%) that we further split between a *local storage area* and a *shared storage area*. Both storage areas are managed by a TSDB instance running on the VM. The Visor instance running on this VM will then feed all monitoring data to the TSDB. The TSDB will store data from its local Visor in the local storage area and further relay the data to other TSDBs where it is stored in the shared storage area. This feature avoids that a TSDB becomes a single point of failure, but still enables quick access to local data. In order to keep network traffic between cloud providers low, any TSDB will only select other TSDBs running in the same cloud to replicate its data. If not enough instances are available to reach the desired replication degree, the maximum possible degree is used. Hence, this concludes to a ring-like topology that has been introduced in peer-to-peer systems [3] and is also used by distributed databases [13].

Table 1. Details of considered times series databases

Name	KairosDB	OpenTSDB	InfluxDB
Version	1.0.0	2.1.0	0.9.0
Datastore	H2/Cassandra	HBase	BoltDB
Distributed	no/yes	yes	yes
InMemory	yes/no	no	yes

Selection of TSDB Table 1 shows a comparison of established TSDB implementations [10] and several of their properties. The results are intermediate as our evaluation is this ongoing (cf. Section 4.1).

The for us relevant details of the TSDBs are its maturity, available datastores, support of distribution and in memory storage. The TSDB should be in some mature state in order to provide a stable version, client libraries and an available documentation. Following the strategy exposed in Section 3.2 the datastores shall be lightweight and ideally support an in memory mode. Also they have to offer a distributed architecture to ensure horizontal scaling and replication.

OpenTSDB offers the best maturity regarding the version number. The underlying datastore HBase supports distribution but regarding the architecture of HBase [9] an in-memory mode is missing. Also, it is not a lightweight datastore [10] and an automated set-up as required in our scenario is not a trivial task and hard to script. Consequently, OpenTSDB is not an applicable solution.

From its capabilities InfluxDB seems suited for the outlined approach. Yet, the recently released version 0.9.0 comes with extensive changes in the stor-

age architecture and API design compared to 0.8.0[5]. Given these changes there currently are no client implementations for version 0.9.0 available.

KairosDB also provides a mature version 1.0.0. It supports the single-site, in-memory datastore H2 and the distributed Cassandra datastore supporting scalable to a hundreds of instances [13]. While Cassandra's resource usage can be limited, in-memory storage is only supported in the commercial version[6].

Following this comparison KairosDB is currently the most appropriate TSDB to use in AXE based on maturity, distribution and the possibility to limit the resource consumption of Cassandra.

3.3 Aggregation

In order to make use of the time series produced by the various raw metrics, these have to be aggregated. Aggregation includes for instance the computation of average values, of maxima, minima, or simply the normalisation of values. In addition to that, aggregation may include merging of metrics, e.g. when computing the average of averages. Hence, aggregation is always application-specific.

Strategy The strategy followed by AXE is based on the metric and metric aggregation concepts provided by SRL (cf. Section 2). In particular, it supports the hierarchical aggregation of metrics with an unlimited depth. In addition, it supports the use of time-bound or element-bound windows specifying the interval of a time series to be used for computations. Finally, the user may specify a schedule for each metric that defines how often a value of a metric shall be computed.

In order to satisfy the requirement for minimum network traffic and scale of the monitoring system, AXE performs aggregation as close to the data source as possible. Hence, all aggregations that require input data from a single VM will be performed on this VM. We refer to this computation to happen in the *host scope*. For this approach only the local storage is accessed and no communication is required which further reduces latency. Aggregations that need input only from VMs from a particular cloud are performed in *cloud scope*. Such computations exclusively access the shared space spanning a cloud. While it is desirable to distribute all computations of a particular cloud scope amongst the affected VMs the definition of a suitable heuristic is currently work in progress. Finally, computations that require input from multiple clouds happen in *cross-cloud scope* (or global scope). These are performed in the home domain of CLOUDIATOR.

It is important to note that values for higher-level aggregated, metrics have to be buffered just as the values of any other metric as well. Here, we use the following strategy to write to our storage platform: Values from local scope computations are treated just like values from raw metrics. Values from computations in cloud scope are written to the shared store of their cloud. The results

[5] `https://influxdb.com/docs/v0.9/introduction/overview.html`
[6] `http://www.datastax.com/`

from cross-cloud scope computations are stored in a possibly distributed TSDB operated at the home domain.

Using this set of hierarchical scopes, we expect to have effectively minimised latency and network traffic while at the same time having equally loaded all VMs with monitoring tasks and hence also equally spread the risk of failures. The deployment of the aggregation tasks onto the Aggregators residing in the system, and hence the decision which scope to use for it, is handled by the Scaling Engine component.

API The API provided by Colosseum in order to configure the monitoring and aggregation functionality of Axe as described above mainly supports the power of SRL. Yet, in order to ease the specification of sensors and aggregation functions independent from the number of deployed virtual machines and the cloud they are currently deployed on, we offer a richer interface.

```
Monitor doMonitorVms(AppInst app, Component comp, SensorDescription sens);
```

Fig. 2. API example. This method will trigger the monitoring of all VMs of this application instance where component `comp` has been installed using sensor `sens`.

The methods (cf. Figure 2 for an example) for defining raw metrics consist of filters (e.g. by the component type) specifying all instances to be monitored, and a sensor description defining what to monitor. The sensor description consists of scheduling information and information which sensor type to be deployed on Visor. The return value of such an invocation can be used in further methods to define higher-level metrics (cf. Figure 3). Here, a map functionality is used to specify the high level metric: That is, for each ingoing (raw) metric a new metric is created (e.g. average CPU usage in the last 5 minutes). The API also supports reduce-like semantics where a single metric is generated from all input metrics (e.g. average of above averages).

```
Monitor mapAggregatedMonitors(FormulaQuantifier quantifier,
        Schedule schedule, Window window, FormulaOperator op,
        List<Monitor> monitors);
```

Fig. 3. API example. This method will install an aggregation triggered according to a schedule, based on an operator, and using a window of elements operating.

3.4 Auto-Scaling

In general, auto-scalers can be categorised in five different classes [14]. For AXE we adopt SRL which mainly belongs to the threshold-based rules as well as time series analysis class. SRL links a set of threshold-based conditions with each other using binary operators. In addition, any set of thresholds may be linked to the values produced by the metrics. Furthermore, any of such constructs has attached a set of scaling actions to be executed whenever the condition has been satisfied. So far, AXE supports to trigger the scale out and scale in of components. Yet, the implementation of further actions is underway. The triggering of rules leads to an invocation of the CLOUDIATOR functionality to bring up a new or shut down an existing VM.

Strategy The auto-scaling functionality of AXE builds on top of the monitoring capabilities. In particular any of the conditions connected via Boolean operators is considered to be a metric on its own taking the values 0 or 1. When the metric turns to 1 the respective action will be triggered and forwarded as request to the other CLOUDIATOR tools, in particular COLOSSEUM. These tasks are executed by the Scaling Engine component.

API The scaling API provides the capability to attach an action to a monitor. The action itself is described in terms of the component to deal with, the scaling type, and its parameters. For instance for horizontal scaling, the parameters are the amount of instances to add/remove, and the allowed maximum and minimum number of instances of that component.

3.5 Architecture

Above descriptions and discussions lead to the architecture from Figure 1 and whose main components are *(i)* the Scaling Engine, *(ii)* the Aggregator, and *(*iii) VISOR. The latter has already been introduced in earlier work [2].

The Scaling Engine is the central managing environment of AXE that controls the distribution and outsourcing of the computation-heavy work to highly scalable and loosely coupled components, the Aggregators. Nevertheless, it is possible to scale the Scaling Engine up to having one instance per scaling rule.

The aggregations are managed and executed by the Aggregators in the system. Due to the design of the system, this can be done in parallel. Also, for their implementation, the focus has been set to minimise latency.

4 Current Status and Future Work

The following presents the current status and gives an outlook on our planned work. We distinguish these aspects for data collection in a TSDB, data aggregation, and scaling.

4.1 Time-series Database

The current version of Axe uses KairosDB with the Cassandra as a datastore. Cassandra is configured to use only a low portion of a VM's resources to keep the impact on the components running on that VM small. Upcoming work comprises a performance-oriented evaluation of InfluxDB and other NoSQL databases focusing on their capabilities for managing time-series data. Further, the Zipkin framework[7] will be evaluated on its suitability for cross-cloud applications.

4.2 Aggregators

Currently, the aggregation functionality is implemented for KairosDB and supports aggregation from and to arbitrary KairosDB instances. We plan to extend these capabilities to fit all predefined operators of SRL. We currently implement aggregators for other databases as well to support the TSDB evaluation.

4.3 Scaling Engine

So far Axe supports horizontal scaling actions. Vertical scaling is currently being implemented. Furthermore, we work reducing the burden for the user when implementing scaling rules. Therefore, we plan to encapsulate SRL's complexity in a simpler language possible inspired from complex-event-processing languages [17].

While SRL and with it Axe adopts concepts from auto-scaling concepts based on threshold-based and time series analysis, other concepts exist that include queuing theory, control theory, and reinforcement learning [14]. Accordingly, Axe borrows all its strengths from SRL, but also the weaknesses and could profit from the integration of other techniques. For instance, reinforcement learning might be handled in external processing tools, that constantly adjust the scaling rules. Future work includes the evaluation of such approaches.

5 Related Work

We compare related work with respect to monitoring and auto-scaling.

Cloud monitoring Lifting monitoring to the cloud comes along with various requirements compared to traditional server monitoring [1]. Tools provided by cloud providers, such as Amazon's CloudWatch[8] suffer from vendor lock-in. Also, additional tools are required when data from different cloud providers shall be aggregated. Established open source monitoring tools such as Ganglia[9] or Nagios[10] are designed to monitor large distributed systems, but struggle with

[7] urlhttps://github.com/openzipkin/zipkin
[8] `http://aws.amazon.com/en/cloudwatch/`
[9] `http://ganglia.sourceforge.net/`
[10] `https://www.nagios.org/`

the dynamic of cloud environments. More cloud-aware monitoring systems such as Zipkin—which is based on Dapper [18]—can cope with the dynamic cloud environment and offer a rich functionality. Yet, in order to scale the monitoring system manual actions or additional tools are necessary. Compared to AXE none of the mentioned tools supports a reduction of communication overhead for cross-cloud applications.

Auto-scaling techniques In contrast to similar scaling engines [4], AXE is not tied to a specific language, but targets to be open for various approaches.

Cloud orchestration tools such as Apache Brooklyn[11], the rules are simple threshold-based and any more complex rules have to be defined in an external monitoring tool. AXE in CLOUDIATOR goes beyond this, as it provides an integrated and easy-to-use solution that even allows changes of the scalability configuration at runtime.

Several projects deal with integrated auto-scaling mechanisms for cloud services. One of them is the EU project CELAR[5]. Auto-scaling in CELAR is based on a multi-level description of combined metrics. By that metrics are assigned to a certain level and when violations occur, the scaling is based on the top level of the topology. While AXE also supports a multi-level description of metrics, it goes beyond the CELAR approach due to the fact that it realises metric aggregation and analysis in a distributed and hierarchical manner.

6 Conclusions

The integrated scaling solutions of current cloud orchestration tools lack an support for sophisticated implementations of auto-scaling techniques. Only such a solution can achieve highly dynamic applications, with the ability to adjust their configuration at runtime in order to cope with unexpected changes of workload. In this paper, we introduced AXE, a novel, cloud provider-independent approach of cloud application monitoring and application adaptation management. AXE supports distributed monitoring of cross-cloud applications and also comes with a distributed, hierarchical aggregation of monitored metrics reducing the network traffic across cloud providers. The adaptation of the *Scalability Rules Language (SRL)* enables the expression of powerful scaling rules based on hierarchical metrics, complex events and threshold. The platform is scalable in itself and hence also supports large-scale applications. It has been integrated in our CLOUDIATOR deployment tool[12].

Acknowledgements The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 317715 (PaaSage) and from the European Community's Framework Programme for Research and Innovation HORIZON 2020 (ICT-07-2014) under grant agreement number 644690 (CloudSocket).

[11] `http://brooklyn.incubator.apache.org/`
[12] `https://github.com/cloudiator`

References

1. Aceto, G., Botta, A., De Donato, W., Pescapè, A.: Cloud monitoring: A survey. Computer Networks 57(9), 2093–2115 (2013)
2. Baur, D., Wesner, S., Domaschka, J.: Towards a Model-based Execution Ware for Deploying Multi-Cloud Applications. In: Proceedings of the 2nd International Workshop on Cloud Service Brokerage September 2014 (2014)
3. Clarke, I., Sandberg, O., Wiley, B., Hong, T.: Freenet: A distributed anonymous information storage and retrieval system. In: Federrath, H. (ed.) Designing Privacy Enhancing Technologies, Lecture Notes in Computer Science, vol. 2009, pp. 46–66. Springer Berlin Heidelberg (2001)
4. Copil, G., Moldovan, D., Truong, H.L., Dustdar, S.: Sybl: An extensible language for controlling elasticity in cloud applications. In: Cluster, Cloud and Grid Computing (CCGrid), 2013 13th International Symposium on. pp. 112–119 (May 2013)
5. Copil, G., Moldovan, D., Truong, H.L., Dustdar, S.: Multi-level elasticity control of cloud services. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) Service-Oriented Computing, Lecture Notes in Computer Science, vol. 8274, pp. 429–436. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-45005-1_31
6. DMTF: Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol (2013)
7. Domaschka, J., Baur, D., Seybold, D., Griesinger, F.: Cloudiator: A Cross-Cloud, Multi-Tenant Deployment and Runtime Engine. In: 9th Symposium and Summer School On Service-Oriented Computing (2015)
8. Domaschka, J., Kritikos, K., Rossini, A.: Towards a Generic Language for Scalability Rules. In: Proceedings of CSB 2014: 2^{nd} International Workshop on Cloud Service Brokerage (2014 (To Appear))
9. George, L.: HBase: The Definitive Guide. O'Reilly Media, 1 edn. (2011)
10. Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J., Breivold, H.P.: Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes. In: 2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014. pp. 602–609 (2014)
11. Jacob, B., Lanyon-Hogg, R., Nadgir, D., Yassin, A.: A Practical Guide to the IBM Autonomic Computing Toolkit. IBM redbooks, IBM Corporation, International Technical Support Organization (2004)
12. Kritikos, K., Domaschka, J., Rossini, A.: SRL: A Scalability Rule Language for Multi-cloud Environments. In: CloudCom, 2014 IEEE 6th International Conference on. pp. 1–9 (Dec 2014)
13. Lakshman, A., Malik, P.: Cassandra: A Decentralized Structured Storage System. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (Apr 2010)
14. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.: A review of auto-scaling techniques for elastic applications in cloud environments. Journal of Grid Computing 12(4), 559–592 (2014)
15. Maurer, M., Breskovic, I., Emeakaroha, V., Brandic, I.: Revealing the mape loop for the autonomic management of cloud infrastructures. In: ISCC 2011. pp. 147–152 (June 2011)
16. Open Grid Forum: Open Cloud Computing Interface - Core (2011)
17. Paschke, A., Kozlenkov, A., Boley, H.: A homogeneous reaction rule language for complex event processing. In: 33rd VLDB 2007 (2007)
18. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc. (2010)