A provider agnostic approach to multi-cloud orchestration

Daniel Baur
Institute of Information Resource Management
Ulm University
Ulm, Germany
daniel.baur@uni-ulm.de

Abstract

Orchestrating workloads in a multi-cloud environment is a challenging task, as one needs to overcome vendor lock-in and select a matching offer from a large and heterogenous market. Yet, existing cloud management tools rely on provider dependent models and manual selection, making runtime changes to the selection in case of provider failures impossible. We propose a provider agnostic, workload centric approach to multi-cloud orchestration relying on a constraint language that allows automatic selection and runtime management of cloud resources overcoming e.g. provider failures.

CCS Concepts • **Computer systems organization → Cloud computing**;

Keywords cloud computing, multi-cloud, orchestration

ACM Reference Format:
Daniel Baur. 2018. A provider agnostic approach to multi-cloud orchestration. In *Proceedings of Middleware Doct Symposium.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Problem Statement

Cloud computing and its computing as a utility paradigm offers on-demand resources to its users following a pay-as-you-go approach. The ongoing commercialization of cloud computing lead to a vast provider landscape offering a highly heterogeneous set of services that are differentiated by using different service models [13] like IaaS, PaaS or SaaS, their access level (private or public), individual features and the (programming) interface they provide. This increasingly complex cloud market makes it difficult to select a matching offer for the workload the user wants to execute. This is aggravated by the fact that once an offer has been selected missing interoperability between providers causes vendor lock-in making it difficult to reevaluate the decision. Furthermore, recent outages of providers [11] have shown that only relying on a single provider may have negative impacts on the availability of the executed service [15]. It is therefore desirable to achieve an architecture that *i)* allows the user to acquire resources across multiple providers in a seamless way, *ii)* deploys a distributed workload automatically on those resources and *iii)* is capable to manage the workload and the underlying resources during runtime. Existing tools apply model driven engineering to solve this problem. The user describes his application using a domain specific modeling language that is enacted by an orchestrator. Yet, these tools have shortcomings [4] that we aim to overcome with our approach. Most tools rely on provider dependent models, meaning that the user has to manually select a matching offer and provide the identifiers together with cloud specific information like firewall configurations or virtual network management at design time. This requires the designer not only to have a complete overview of the cloud market, but also to have knowledge about particularities of specific providers. This counteracts portability, as the decision for an offer can not be revised (automatically) at runtime to e.g. react on provider failures. Additionally, existing tools follow a deployment centric view, where an application is decomposed to several components that are hosted on the described infrastructure. However, typical use cases like data processing require multiple levels as the workload is executed within frameworks managing the underlying resources and may change during runtime as new workloads are submitted to the system.

2 Related Work

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [17] is an OASIS standard for describing topologies consisting of a (virtual) infrastructure and hosted applications. While being generic enough to support various concepts, the concrete realization is left to the implementation of the orchestrator. Cloudify[1] e.g. heavily relies on custom types for its TOSCA implementation. As these

Middleware Doct Symposium, December 2018, Rennes, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]https://cloudify.co/

types directly reference a cloud provider, the initial manual selection can not be changed at runtime. Alien4Cloud[2] on the other hand relies on TOSCA's normative types and the concept of abstract and concrete templates. The abstract template defines requirements that need to be fulfilled by the concrete type. While this achieves portability across providers, the selection of the concrete provider is a manual process at design time that can not be changed during runtime. A similar approach is taken by [1], that converts abstract TOSCA templates to concrete Cloud Application Management for Platforms (CAMP) [9] plans. The approach by [6] also uses abstract templates that are refined during the deployment process, but not during runtime.

Roboconf [18] and Occopus [14] both use a proprietary domain specific language that is not provider agnostic and thus can not achieve portability across providers. In contrast to previous approaches [10] relies on models@runtime, allowing it to reflect changes in the model to the running system by an adaptation engine. However, their model is only partly provider agnostic as it still requires manual interaction e.g. to change the provider.

Similar to the approach, presented in this paper, Google Borg [21] and Apache Mesos [12] follow an application-centric view. While Mesos itself handles only resource management, frameworks like Apache Aurora[3] or Marathon[4] offer deployment capabilities. However, both are designed to share a static resource pool across multiple users, while our approach manages a dynamic set of resources across multiple providers.

3 Approach

To overcome the prior depicted shortcomings our approach builds upon three main concepts: *i)* a provider agnostic modeling language that allows the user to describe the resource demand without referencing a specific provider, *ii)* a workload centric description that allows us to automatically derive the required infrastructure to host the workload, *iii)* a runtime management system relying on the provider agnostic model that is capable to recover from provider errors and failures. An overview of our approach is given in Figure 1.

Provider agnostic To achieve a provider agnostic model, we separate the provider information from the workload that needs execution. Our provider model contains the type of the provider respectively the API it provides and authentication information. Using this information we repeatably query the provider to discover his resource offers ensuring up-to-dateness. We depict eligible combinations of those offers and their properties, e.g. the computational resources or its (geographical) location by node candidates. The user may also manually register additional node candidates to represent

[2]http://alien4cloud.github.io
[3]http://aurora.apache.org/
[4]https://mesosphere.github.io/marathon/

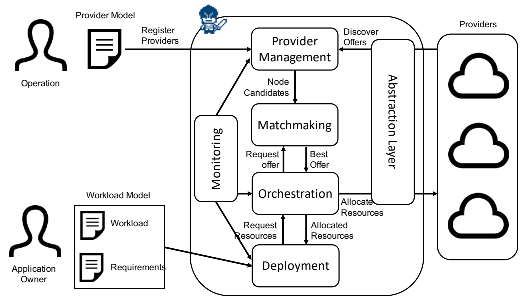

Figure 1. Overview of the Cloudiator approach

already existing nodes. To be able to select matching node candidates, we provide a constraint language [3] that allows the user to express requirements for his workload by referencing the properties of the node candidates. The constraints *nodes->forAll(hardware.cores >= 4)* and *nodes->size() >= 2* will e.g. express that this workload requires at least two resources with four CPU cores. To break ties we allow the user to specify an optimization criteria, e.g. *minimize(nodes.sum(price))* that will minimize the price of all selected nodes. Using the node candidates and the requirements expressed by the user we generate and solve a constraint satisfaction problem (CSP) and allocate the selected resources and deploy the workload. We accompany the constraint language with an abstraction layer, hiding semantic and syntactical differences across different providers.

Workload Centric In contrast to other approaches, we propose a workload centric view on cloud orchestration. Instead of separating the application into individual components that are hosted on resources, the user depicts a workload and its requirements. This approach keeps the description portable across different cloud service levels and environments and also allows us to support runtime behaviors like continuously running services or repetitively running batch jobs. The workload is described as *Jobs* that act as a logical group of *Tasks* describing the actual workload. A *Task* can have different *Behaviors* specifying if it depicts a continuously running service or a (repetitively) running batch job. Each task exposes *Interfaces*, depicting how it can be executed in different environments. *Tasks* may reference other *Tasks* to depict *Communication* dependencies. Each *Task* may express multiple constraints and an optimization criteria (cf. previous paragraph) expressing its resource demand that can be fulfilled by one or multiple resources. Additionally, each *Task* expresses runtime requirements, either explicitly defined by the user or implicitly defined by the exported *Interface*. We e.g. support a *SparkInterface* that requires an Apache Spark[5] cluster for execution. We match these requirements with the runtime capabilities expressed by each node candidate the providers offer. In addition, we host a software catalogue, used to enrich the provided environments

[5]http://spark.apache.org/

of node candidates. The runtime of the *SparkInterface* thus can be either fulfilled e.g. by a PaaS offer directly offering an Apache Spark cluster, by an user registered Spark installation or by an IaaS offer where we install Apache Spark using our software catalogue.

Runtime Management and Recovery Our experience shows, that deploying distributed workloads in a cross-cloud environment is error prone. Not only may the cloud provider itself fail, but also network connectivity between providers, the deployed applications or cloud specific details like quotas may hinder the deployment during runtime. To be able to react on errors at runtime, we continuously monitor the providers, the allocated resources and the running workloads [8]. If we detect errors, we try to categorize the failure by its cause analyzing e.g. the return code issued by the provider's API. Based on these categories we mark the affected entities as failed. Failures may either affect the available node candidates (e.g. if a provider or an availability zone fails all relating offers are removed), the running nodes and workloads or both. The same mechanism is used to represent changes in the models that the user may submit at any time. Should the user e.g. update a specific workload, all instances relating to the workload will be marked as failed. Whenever a running resource is marked as failed, a new matchmaking process is triggered to derive a new solution. As the existing solution may only be partly invalidated, we import the existing solution into the matchmaking process thus preferring new solutions that reuse still running resources. Finally, the new solution will be applied by acquiring the resource differences and deploying the workload.

4 Implementation

We implement our approach with the Cloudiator[6] [5, 7] framework using the architecture depicted in Figure 2. As unification of an heterogeneous environment is a complex task, we rely on a micro-service architecture to be able to plug in additional features without affecting existing components and achieve loose-coupling of the different services via a message queue. For each architectural level, we typically use a coordination agent and worker agents executing the tasks. For resource and provider management level this e.g. means that the *Node Agent* is responsible for the coordination of resource allocation, while the *PaaS Resource Agent* and the *IaaS Resource Agents* are responsible for allocating the resources. We furthermore allow each worker agent to scale, meaning that different *IaaS Resource Agents* could handle different providers. Decoupling the different functionalities (especially the deployment from the resource management layer) also ensures that they can be used separately, e.g. allowing the user to only acquire virtual machines from a set of registered providers. Cloudiator offers multiple interfaces with which the user may interact. Based on his preferences he may either

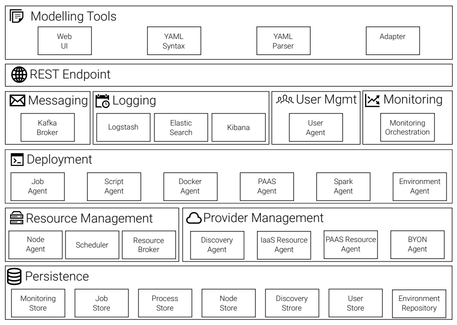

Figure 2. Architecture of Cloudiator

express the model using a YAML syntax that is parsed and validated and transmitted to the REST endpoint by an adapter or he can directly access the REST interface using a CRUD based web interface. The REST interface is described using the OpenAPI-Specification[7] allowing easy integration with different systems. For the development of the abstraction layer supporting multiple clouds, we rely on Apache jclouds[8]. However, as we identified multiple shortcomings [2] we provide a wrapper implementation overcoming those issues. We have recently enriched our constraint language by relying on the Object Constraint Language (OCL) [16] achieving higher expressiveness. We rely on the EMF OCL[9] implementation to parse the OCL constraints. For the final solving of the CSP we rely on the Choco Solver [19].

5 Evaluation

The evaluation of our approach is two-fold. First, we provide a quantitative evaluation proofing that our approach is feasible. This part of the evaluation mainly targets the matchmaking mechanism, showing that it is capable of solving the resulting CSP in reasonable time. Initial results [3] show that it is capable to select offers even from a large offer space of multiple providers. Other metrics will include the recovery time in case of errors and the overall deployment time using our approach. While these metrics largely depend on external properties (e.g. the resource start time of providers) we use these metrics to evaluate the overhead introduced by our approach. The second part will provide a qualitative evaluation of our approach, showing the practical applicability and generality of our approach using multiple case studies. To represent the functionalities of our approach, the case studies should span different application domains, be deployed on at least four different providers across different services levels. While our Cloudiator framework was already successfully used to deploy multiple applications ranging from typical three-tier web services to more complex flight scheduling or MPI-based Computational Fluid Dynamics (CFD)

[6]https://github.com/cloudiator

[7]https://www.openapis.org/

[8]https://jclouds.apache.org/

[9]https://wiki.eclipse.org/OCL/OCLinEcore

simulations we have yet to evaluate deployments spanning over multiple different environments and large scale deployments. To evaluate the runtime recovery, we plan to extend our deployment and orchestration layer with the capability to induce artificial errors and use it to evaluate multiple recovery strategies, e.g. a strategy that prefers moving to different providers against strategies that prefer recovering the application at the initially selected provider.

6 Conclusion and Future Work

We have presented an approach for workload management in a heterogenous environment using resources provided by multiple providers across different service levels. In contrast to existing solutions, our approach applies a workload centric view that is capable to overcome vendor lock-in and thus enables runtime management overcoming failure of providers. While our approach is theoretically able to abstract the cloud environment, requiring minimal knowledge of the user about single providers the verbosity of providers' APIs sometimes hinders practical applicability. Especially when it comes to meta-data information about the offers, manual or automatic enrichment from other sources is required. Similar the interchangeability of workload descriptions between different service levels would be a desired feature that requires future research. Our generic approach and flexible architecture together with an extensive API giving access to a provider independent orchestration layer enables other research exploring the behavior of systems in a distributed cloud environment, e.g. availability of distributed database systems [20].

Acknowledgements

This work is done under the supervision of Stefan Wesner. The author would like to thank Jörg Domaschka and Daniel Seybold for their support. The research leading to these results has received funding from the EC's Framework Program HORIZON 2020 under grant agreement numbers 731664 (MELODIC) and 732258 (CloudPerfect).

References

[1] K. Alexander, C. Lee, E. Kim, and S. Helal. 2017. Enabling End-to-End Orchestration of Multi-Cloud Applications. *IEEE Access* 5 (2017), 18862–18875.

[2] Daniel Baur and Jörg Domaschka. 2016. Experiences from Building a Cross-cloud Orchestration Tool. In *3rd Workshop on CrossCloud Infrastructures & Platforms.*

[3] Daniel Baur, Daniel Seybold, Frank Griesinger, Hynek Masata, and Jörg Domaschka. 2018. A Provider-Agnostic Approach to Multi-cloud Orchestration Using a Constraint Language. In *2018 18th IEEE/ACM CCGRID*. 173–182.

[4] Daniel Baur, Daniel Seybold, Frank Griesinger, Athanasios Tsitsipas, Christopher B Hauser, and Jörg Domaschka. 2015. Cloud Orchestration Features: Are Tools Fit for Purpose?. In *UCC*.

[5] Daniel Baur, Stefan Wesner, and Jörg Domaschka. 2014. Towards a model-based execution-ware for deploying multi-cloud applications. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 124–138.

[6] Miguel Caballer, Sahdev Zala, Álvaro López García, Germán Moltó, Pablo Orviz Fernández, and Mathieu Velten. 2018. Orchestrating Complex Application Architectures in Heterogeneous Clouds. *Journal of Grid Computing* 16, 1 (01 Mar 2018), 3–18.

[7] Jörg Domaschka, Daniel Baur, Daniel Seybold, and Frank Griesinger. 2015. Cloudiator: a cross-cloud, multi-tenant deployment and runtime engine. In *9th Symposium and Summer School on Service-Oriented Computing*.

[8] Jörg Domaschka, Daniel Seybold, Frank Griesinger, and Daniel Baur. 2015. Axe: a novel approach for generic, flexible, and comprehensive monitoring and adaptation of cross-cloud applications. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 184–196.

[9] Jacques Durand, Adrian Otto, Gilbert Pilz, and Tom Rutt. 2014. Cloud Application Management for Platforms Version 1.1. *OASIS* (2014).

[10] Nicolas Ferry, Franck Chauvel, Hui Song, Alessandro Rossini, Maksym Lushpenko, and Arnor Solberg. 2018. CloudMF: Model-Driven Management of Multi-Cloud Applications. *ACM Transactions on Internet Technology (TOIT)* 18, 2 (2018), 16.

[11] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. 2016. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 1–16.

[12] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 295–308.

[13] Steffen Kächele, Christian Spann, Franz J. Hauck, and Jörg Domaschka. 2013. Beyond IaaS and PaaS: An Extended Cloud Taxonomy for Computation, Storage and Networking. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC '13)*. IEEE Computer Society, Washington, DC, USA, 75–82.

[14] József Kovács and Péter Kacsuk. 2018. Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures. *Journal of Grid Computing* 16, 1 (01 Mar 2018), 19–37.

[15] R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente. 2018. Orchestrating the Deployment of High Availability Services on Multi-zone and Multi-cloud Scenarios. *Journal of Grid Computing* 16, 1 (01 Mar 2018), 39–53.

[16] Object Management Group (OMG). 2014. Object Constraint Language Specification, Version 2.4.

[17] Derek Palma and Thomas Spatzier. 2013. Topology and orchestration specification for cloud applications (TOSCA). *OASIS* (2013).

[18] L. M. Pham, A. Tchana, D. Donsez, N. de Palma, V. Zurczak, and P. Gibello. 2015. Roboconf: A Hybrid Cloud Orchestrator to Deploy Complex Applications. In *2015 IEEE 8th International Conference on Cloud Computing*. 365–372.

[19] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2017. *Choco Documentation.* TASC - LS2N CNRS UMR 6241, COSLING S.A.S. http://www.choco-solver.org

[20] Daniel Seybold, Christopher B Hauser, Simon Volpert, and Jörg Domaschka. 2017. Gibbon: An Availability Evaluation Framework for Distributed Databases. In *OTM*.

[21] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.