Cloud Orchestration Features: Are Tools Fit for Purpose?

Daniel Baur, Daniel Seybold, Frank Griesinger, Athanasios Tsitsipas, Christopher B. Hauser, Jörg Domaschka
Institute of Information Resource Management
University of Ulm, Germany
Email: {firstname.lastname,joerg.domaschka}@uni-ulm.de

***Abstract*—Even though the cloud era has begun almost one decade ago, many problems of the first hour are still around. Vendor lock-in and poor tool support hinder users from taking full advantage of main cloud features: dynamic and scale. This has given rise to tools that target the seamless management and orchestration of cloud applications. All these tools promise similar capabilities and are barely distinguishable what makes it hard to select the right tool. In this paper, we objectively investigate required and desired features of such tools and give a definition of them. We then select three open-source tools (Brooklyn, Cloudify, Stratos) and compare them according to the features they support using our experience gained from deploying and operating a standard three-tier application. This exercise leads to a fine-grained feature list that enables the comparison of such tools based on objective criteria as well as a rating of three popular cloud orchestration tools. In addition, it leads to the insight that the tools are on the right track, but that further development and particularly research is necessary to satisfy all demands.**

I. Introduction

The last decade has been dominated by the Cloud in both research and industry. Nevertheless, many problems have been around since the beginning of the cloud era and are substantially hindering the move forward. These include vendor lock-in, the incomparability of cloud providers with respect to performance per price, the weak adoption of cloud standards from the providers, etc.

Particularly, researchers from domains that make use of clouds such as software engineering and data mining, but also software architects and DevOps teams need robust and powerful mechanisms to bring their applications and innovations to the Cloud—ideally to many cloud providers. In order to perform evaluations and test the elasticity of their application, these users need to be able to seamlessly change the distribution of their application across multiple cloud providers. They also have to be able to quickly change the scaling factor of individual components. Similar considerations hold for start-ups whose demands change with the growth of the business.

In order to satisfy these demands, a powerful and reliable cloud orchestration and operation platform is needed. Indeed, there are multiple commercial and open-source tools available that promise to solve above mentioned issues. Yet, it is hard to find a visible distinguishing factor. Also, there are barely any sources available that report on the success of using these tools. This document aims at closing this information gap by the following contributions: *(i)* We establish a fine-grained feature list that enables the comparison of such orchestration tools based on objective criteria. We also provide a guideline on how to grade tools according to the features. *(ii)* We identify tools that promise to realise some or multiple of the above mentioned features. *(iii)* We rate three tools according to our feature list.

Based on our feature extraction, it becomes possible to compare existing and upcoming orchestration tools: What is more, the identified lack of supported features shall drive future research and development with respect to cloud orchestration. The remainder of this document is structured as follows: Section II introduces the terminology of the document as well as the methodology of our approach. Section III introduces background and basic capabilities of the tools. Section IV introduces the feature list and does the comparison for each introduced feature. Section V presents the comparison results from applying the list in tabular I. Section VI discusses related work before we conclude.

II. Background

In the following, we briefly introduce the terminology that we use throughout the document and further classify the methodology of our comparison.

A. Terminology

Even though more advanced terminologies have been published (e.g. [1]), we follow the well-known NIST standard [2] defining the three service models IaaS, PaaS, and SaaS.

We define *(cloud) application* as a possibly distributed application consisting of multiple interlinked application components. For illustration consider a blog application B that may consist of the three components load balancer lb, application server together with the business logic as, and a database db.

We depict cloud orchestration tool, as a software component that manages the complete lifecycle of a cloud application. It therefore needs to fulfil the following criteria: *(1)* application specification (definition); *(2)* deployment of specified applications: the tool has to acquire virtual machines and then distribute component instances over them; *(3)* collection of monitoring data from deployed instances. This data further has to be consolidated and aggregated and provided to the operators. *(4)* If application management shall be (semi-)automatic, the tool has to allow the definition of rules that capture when to execute which management task (e.g. scaling). We therefore explicitly exclude pure deployment tools like Chef.

B. Methodology

While the primary goal of this document is to present a feature list that helps rating cloud development tools a secondary goal is to show its applicability to existing software tools. For this purpose we select three tools fulfilling the above mentioned four criteria: Apache Brooklyn[1], Cloudify[2], and Apache Stratos[3].

For the evaluation we install the tools in our local data centre and apply the defined features. As a sample application we select a three-tier application consisting of *NGINX* as a load balancer, *Node.js* as an application server running the *Ghost* blogging application, and *PostgreSQL* as a database management system. The used application descriptions are available on Github[4].

III. Introduction to Tools

All three selected tools make use of third party IaaS platforms for running applications and all of them offer a PaaS-like interface to operators. Below we introduce the tools in particular with respect to their terminology and define the versions used for the evaluation. As a rule of thumb, we used the latest stable version available at the time writing this document.

Apache Brooklyn is compared using milestone release 0.7.0-M2-incubating. Brooklyn's application description is based on *blueprints* written in a *domain specific language* (DSL) in YAML format. A blueprint contains global properties (e.g. name, cloud provider configuration) as well as a services block defining application components. The definition of the components is split into abstract types and entities. The type captures the structure for entities such as defining the properties and interfaces. The entities refer to those types and provide the concrete configuration. Each type is linked to a Java implementation. Besides deployment, Brooklyn also supports a monitoring interface and elastic adaptation at runtime.

Cloudify by *GigaSpaces Technologies* is offered in a free open-source edition and a commercial Pro edition. Our comparison is based on the open-source edition version 3.2. Cloudify uses a TOSCA-aligned modelling language for describing the topology of the application which is then deployed to allocated virtual machines in the cloud environment. TOSCA-like, Cloudify splits the blueprint in a type and template definition. Types define abstract reusable entities and are to be referenced by templates. The types therefore define the structure of the template, by e.g. defining the properties that a template can have/must provide. The template then provides the concrete values. This mechanism is used for nodes as well as for relationships.

Apache Stratos is compared using the release candiate 4.1.0-RC2. Stratos makes use of an abstract virtual machine description, named *cartridge*, with an application component type (named *cartridge type*) like an application runtime container (e.g. Tomcat). An application is described by a single cartridge or/and a set of cartridges (*groups*), combined with deployment and scaling policies. The cartridges, applications and other configurations are represented in a Stratos specific JSON format. For the deployment itself, it solely relies on the DevOps tool Puppet. The application itself is subsequent cloned from a Git repository. Stratos is installed as one central controller and in all virtual machines by having a virtual machine image prepared with the necessary software (Stratos and Puppet agents) installed.

[1]https://brooklyn.incubator.apache.org/

[2]http://getcloudify.org/

[3]http://stratos.apache.org/

[4]https://github.com/dbaur/orchestration_comparison

IV. Features and Comparison

In this section we perform the actual comparison. Each of the following two sections introduces a set of features. Section IV-A addresses cloud-related aspects and Section IV-B considers application-related features.

In the individual sections, we introduce the features of the set and for each of them, *(i)* define the feature in detail and argue why it is desirable. In case sub-features exist for a feature, they are introduced as well (highlighted in italics). Moreover, we *(ii)* discuss to what extend each feature and all its sub-features are actually supported by each of the three tools.

A. Cloud Feature Set

The Cloud Feature Set relates to the cloud infrastructure. Hence, its features focus on supported deployment across multiple cloud providers and levels.

1) Multi-Cloud Support Feature: Supporting *multiple cloud providers* is one of the most crucial features for cloud application management tools, as it allows the selection of the best matching cloud offer for an application from a diverse offering landscape. Cloud providers often differ from each other regarding their API. This causes the user to suffer from a vendor lock-in once he depends on the native API of a cloud provider. For that reason cloud orchestration tools should offer a *cloud abstraction layer* which hides differences and avoids the need for provider-specific customisation thus removing the vendor lock-in.

Apache Brooklyn Support: Brooklyn uses Apache jclouds as cloud abstraction layer and therefore supports many public and private cloud providers.

Cloudify Support: Cloudify comes with plugins supporting AWS, Openstack and VMWare vCloud. It also offers a contributed plugin for Apache Cloudstack and two commercial plugins (Pro version) for VMWare vSphere and SoftLayer. Nevertheless, Cloudify does not support an abstraction layer and each model needs to explicitly reference cloud provider specific features.

Apache Stratos Support: Stratos utilises jclouds as a cloud abstraction layer, supporting multiple providers. Support is tested for AWS EC2, Openstack and Google Compute Engine. Yet, the abstraction is imperfect as application specifications still need to refer to cloud specific entities.

2) Cross-Cloud Support Feature: Cross-cloud support enhances the multi-cloud feature such that the user is able to deploy a single application in a way that its component instances are distributed over multiple cloud providers. For instance, the database may be deployed in a private cloud on the user's premises while numerous instances of the application

server run in a public cloud. The advantages of cross-cloud deployment are three-fold: *(i)* It allows a sophisticated per-component instance selection of the best-fitting offer. *(ii)* It leverages the availability of the application as it introduces resilience against the failure of individual cloud providers. *(iii)* It helps coping with privacy issues (private vs. public cloud).

Apache Brooklyn Support: Brooklyn supports cross-cloud deployments on a per-component level: Each component can be bound to a separate cloud provider by referencing its configuration.

Cloudify Support: Cloudify offers cross-cloud support. For each virtual machine defined in the model, the user can reference a different cloud provider.

Apache Stratos Support: Stratos allows the definition of network partitions which are logical groups of IaaS resources such as regions or availability zones. Network partitions enable cross-cloud scaling and deployment using policies like round robin through available network partitions. Cartridges may only be configured for a subset of network partitions.

3) External PaaS Support Feature: In addition to supporting IaaS clouds, the support of PaaS clouds (e.g. Google App Engine) is desirable. For PaaS offers ready-to-deploy application containers, it reduces complexity compared to IaaS. This also reduces the management effort for the user. On the downside, it comes at the cost of reduced flexibility, it is the provider that defines the container configuration.

Tool Support: None of the three tools allows the use of external PaaS clouds.

4) Support of Cloud Standards Feature: In addition to supporting multiple provider APIs (cf. Section IV-A1) the support of cloud API standards such as CIMI [3] or OCCI [4] enables support for any cloud provider adapting such a standard.

Tool Support: None of the three tools supports any cloud interface standard.

5) Bring Your Own Node (BYON) Feature: BYON captures the ability to use already running servers for application deployment. It enables the use of servers not managed by a cloud platform or virtual machines on unsupported cloud providers.

Apache Brooklyn Support: Brooklyn supports BYON by providing an IP address and login credentials for the server.

Cloudify Support: Cloudify supports BYON through an externally installable Host-Pool Service that works as a cloud middleware mock-up. When enabled, Cloudify requests IP addresses and login credentials from this service whenever it needs to provision a new server.

Apache Stratos Support: Stratos does not support BYON, despite the general ability of jclouds to do so.

B. Application Feature Set

This section discusses features related to the deployment and automation of applications. It starts with features related to the application description language and deployment, continues with features related to runtime adaptation, and finally discusses additional features such as support of the Windows operating system.

1) Application Standards Feature: Supporting open standards such as TOSCA [5] and CAMP [6] for modelling the application topology, the component life cycles, and the interaction with the cloud management tool facilitates the usage of the tool and further increases the reusability of the topology definition, as it avoids moving the vendor lock-in from cloud provider level to management tool level (cf. Section IV-A1). Moreover, it reduces the initial effort and costs to learn a new DSL.

Apache Brooklyn Support: Brooklyn's YAML format follows the CAMP specification, but uses some custom extensions. Yet, it is possible to deploy CAMP YAML plans with Brooklyn and via the separately provided CAMP server. Support for TOSCA is planned for a future release.

Cloudify Support: While Cloudify's DSL for the deployment description is strongly aligned with the TOSCA modelling standard it does not directly reference the standard types, but instead defines its own profile following the *TOSCA Simple Profile in YAML*. Full TOSCA support is planned.

Apache Stratos Support: Stratos does not implement any standard.

2) Resource Selection Feature: The resource selection is part of the application topology description. It defines the resources used for the deployment of a component instance in an IaaS cloud. Hence, a resource will commonly refer to the virtual machine type/flavour, an image, and a provider-specific location: $\langle location, hardware, image \rangle$. A tool has mainly four possibilities to define or derive such a tuple: *(i)* In an *manual binding* the user provides the concrete unique identifiers of the cloud entities. *(ii)* In an *automatic binding* the user defines abstract requirements regarding the defined tuple (e.g. number of cores). These are then bound to a concrete offer at runtime by the tool. Automatic binding can be enhanced by offering an *iii) optimised binding*. Here, the specification of optimisation criteria based on attributes of the cloud provider such as price or location is possible. Finally, *(iv) dynamic binding* offers a solving system that enables changes to the binding based on runtime information, e.g. metric data collected from the monitoring system. Automatic binding is a prerequisite for complex deployment and runtime adaptation scenarios, as it allows the cloud management platform to dynamically select the concrete offer during runtime. Optimised binding offers optimised selection based on simple criteria like price. Dynamic binding offers the possibility to use a solver applying an optimisation algorithm for selecting the best-fitting offer based on complex criteria (performance or performance per $).

Apache Brooklyn Support: Brooklyn supports manual as well as basic automatic binding. For the latter it supports resource boundaries for the hardware. The resource selection happens either in the global or in the component-specific parts of the blueprint.

Cloudify Support: Cloudify exclusively supports manual binding of the resources used for a virtual machine. The user needs to reference a cloud provider specific node type (e.g. `cloudify.openstack.nodes.Server` for Openstack) to provide the implementation for the chosen cloud provider, as well as the specific properties defined by this type. These include the location, the image and the flavour information.

Automatic binding of resources (like offered by TOSCA's `nodes_filter` requirements specification) is not supported by Cloudify. Due to this shortcoming optimised and dynamic bindings are also not possible.

Apache Stratos Support: The resource selection in Stratos is a manual process when configuring cartridges by referencing to *(i)* an image and *(ii)* a hardware description in an IaaS cloud.

3) Life Cycle Description Feature: The life cycle description defines the actions that need to be executed in order to deploy the application including all its component instances on started virtual machines. The basic approach for the life cycle description of the application is to provide *shell scripts* that are executed in a specific order. This approach can be extended to support *DevOps tools* such as Chef that offer a more sophisticated approach to deployment management and ready to use deployment descriptions.

Apache Brooklyn Support: In Brooklyn each defined type provides basic life cycle actions called effectors. These can be configured in the concrete application component definition. The configuration can either happen with shell scripts or by referencing Chef recipes.

Cloudify Support: Cloudify relies on the interface definition of TOSCA for defining life cycle actions. The base node type defines multiple life cycle actions as interfaces, that are executed during deployment. The actions are defined as shell scripts or by using Chef and Puppet. Support for Salt is in development. Cloudify also has the possibility to provide python scripts. This is evaluated and provides immediate access to Cloudify's API.

Apache Stratos Support: The life cycle description for managing virtual machines is done by Stratos itself, while the software setup is delegated to Puppet. Stratos cartridges have a cartridge type, which is a reference to a Puppet module. During application deployment, Stratos identifies and invokes the needed Puppet modules.

4) Wiring and Workflows: Most cloud applications are distributed applications where components reside on different virtual machines, e.g. the application server resides on a compute optimised host, while the database is on a storage optimised host. Hence, the modelling language needs to support a way to configure those communication relationships between the components by offering a way to pass the endpoint, either before the start of the dependent component (database starts before application server) or after (application server is added to already running load balancer).

A straight-forward approach to resolve those dependencies is *attribute and event passing*. That is, the tool allows the user (life cycle scripts) to lock/wait for attributes to become available or register listeners on topology change events. This is commonly achieved by a global registry shared between all component instances of an application.

Obviously, this approach offloads most complexity to the user who needs to, e.g., make sure that the database URL is only available when it already started. An improvement is a *manual workflow* definition. Here, the user defines a workflow taking care of the deployment order. Finally, the easiest way for the end user is an *automatic workflow* deduction, where the modelling language is sufficiently verbose to allow the system to automatically deduce the correct workflow from the defined life cycle actions on the virtual machines and their relationships.

Additionally, a tool may offer extensions to its model, allowing to refer to *external services* like PaaS (cf. Section IV-A3) or SaaS services, to ensure that the deployment engine is aware of this dependency and e.g., can open ports on firewalls.

Apache Brooklyn Support: Brooklyn supports wiring by attribute-and-event-passing. It offers a locking action, that waits until the dependent service provides a required attribute. The reverse way, where a later starting service needs to reconfigure a running service, is not supported out of the box. Instead, the user has to implement this functionality. Yet, for the commonly used load-balancer scenario, Brooklyn supports predefined static out-of-box load balancing. The tool neither supports workflow scenarios nor access to external services.

Cloudify Support: Cloudify uses the relationship mechanism of TOSCA. It defines a generic `depends_on` relationship type that offers the execution of custom actions on either the source or the target of the relationship on specific events. Combined with a shared configuration space available via e.g. a shell extension, this allows the user to configure endpoints before or after the start of a service. Using python the user can implement custom workflows, making sure that the life cycle actions are executed in the correct order. If the user only uses the basic life cycle actions, Cloudify is capable of automatically deducing the correct execution order. Cloudify does not support external services by default. Yet, the modular communication relationship might allow adding this feature if needed.

Apache Stratos Support: Stratos provides a metadata service where the component instances of an application can export and import variables. This basic but manual wiring using variable exchange must be implemented by the user at application setup. In case of joining or leaving component instances Stratos broadcasts a topology change event, which is used by Stratos core functionality (e.g. notify the user for a successful application deployment) or any load balancers existing in a deployed application setup, to update their state for redirecting client requests. For assessing the overall deployment workflow, support of both Stratos and Puppet have to be considered. Stratos defines a startup order of virtual machines, while Puppet has a more complex dependency expression for each single virtual machine. Puppet modules can be depending on each other and inside of one module, dependencies between different steps can be defined. This rather static deployment workflow is defined in advance of the application deployment. The flow cannot be controlled during application boot or execution time. Stratos does not support external services.

5) Monitoring Feature: Being able to track the behaviour of the application is a key to assessing the quality of the deployment. Consequently, it is necessary to monitor the current resource consumption and the quality-of-service (QoS) parameters of the application. Only if the end-user is aware of current bottlenecks he is able to remove them. The cloud management framework should therefore offer a way to measure *system metrics* like CPU usage and *application specific*

metrics like number of requests. If those predefined metrics are not sufficient, the tool should offer a well defined way to add *custom metrics*.

An *aggregation mechanism* enables users to compute higher-level metrics (e.g. 10 minute average over CPU load) and also to combine multiple metrics (e.g. average over 10 minute CPU average of all instances of a particular component). For helping users in accessing and assessing the current load on his application, it is beneficial to have the gathered metrics presented through a *dashboard*. In order to support higher-level evaluation of monitoring data *access to historical data* is desirable.

Apache Brooklyn Support: Brooklyn's uses a pulling mechanism gathering the data from the virtual machines by either executing remote actions or accessing an external tool. It is the user's responsibility to implement those actions, or to provide an interface to an external monitoring tool. Brooklyn does not store historical data and only supports access to the latest measured value impairing aggregation. The latest value of all metrics is shown in a dashboard.

Cloudify Support: Cloudify's monitoring system relies on the Diamond monitoring daemon that has built-in collectors for the most common system and application metrics. Additionally, it offers an interface for the implementation of custom collectors. The Cloudify user interface for viewing (historical) metrics is only available in the closed-source Pro version. Aggregation of metrics is possible using the policy framework (cf. Section IV-B6).

Apache Stratos Support: Stratos uses a cartridge agent residing within each virtual machine. This agent comprises a *Health Publisher* to avail itself of the machine's health statistics, load average, free memory, and the amount of in-flight requests. It is not possible to define further custom metrics. Monitoring data is sent to a central real-time processing engine where aggregation and evaluation is performed. Support for visualisation of current and historical health statistics through the Web GUI is is planned for the future.

6) Runtime Adaptation Feature: While monitoring (cf. Section IV-B5) lays one of the foundations for adapting the configuration of the application during runtime, the cloud management platform should be able to react upon deviations automatically. For this purpose, the user needs to be able to define *(i)* metric and QoS conditions that trigger *(ii)* actions if violated: For instance, scale component horizontally, if the CPU usage is $> 80\%$. In order to support that, the cloud management tool should at least offer a *simple threshold-based approach* for the detection of violations and support *horizontal scaling*. As extensions, we consider *repair* and the *custom definition* of actions on the actions side, and more complex *rule engines* with respect to QoS.

Another feature related to adaptation is *continuous delivery* of the application. It enables the user to change the topology model of an already deployed and running application (e.g. add a load balancer, or update the software version of a component). This should be possible with as few changes as needed to the running components.

Apache Brooklyn Support: Brooklyn policies enable the specification of metrics/QoS. By default a threshold-based policy is available. Scaling is enacted in so called clusters. By default Brooklyn supports horizontal scaling. Both the policies and the clusters are in general customisable by new implementations, but there is no easy way to plugin such custom extensions. Continuous delivery is exclusively possible on component level, namely by redeploying single components with updated software.

Cloudify Support: Cloudify uses the event stream processor Riemann for the definition of QoS requirements. By default, they provide policies for host failure detection, simple threshold and exponential weighted moving average threshold. It enables the definition of custom aggregations and policies using Clojure and Riemann. On the action side, Cloudify offers workflows for healing the application (uninstall/reinstall) and a scale workflow offering horizontal scaling. Complex scaling scenarios (e.g. cloud bursting, vertical scaling) are not supported out of the box. Instead, the user may define custom workflows using a DSL. This enables support for complex scenarios, but leaves the responsibility with the user. Continuous delivery of the application is currently not supported by Cloudify, meaning that the user has to un-deploy the entire application, even for minor changes in the model. Support for this has been announced for the Pro version.

Apache Stratos Support: Stratos balances the QoS requirements by using policies, that enable a multi-factored autoscaling. Using client requests and system load as health data (cf. Section IV-B5) combined with a complex event processor and the Drools rules engine, Stratos enacts horizontal scaling to the environment. Moreover, repair actions are supported, in case some tasks within virtual machines of an application topology fail (e.g. installation of required software), by automatically destroying and re-creating the affected instance. The implementation of custom actions is not foreseen. Continuous delivery is not supported. Instead, the user has to un-deploy the whole application first and change its definition.

7) Reusability and Sharing of Models Feature: When using cloud management tools, the main task of the user is the creation of the application description based on components. As this imposes a high initial effort, this task needs to be supported by sharing of existing models.

Regarding *reusability* the tool should offer a modularised approach regarding the application description. Generally speaking, each application description consists of components. In order to facilitate re-use, modularisation shall be used to an extend where the description and life cycle handling of components is mostly self-contained and independent from e.g. the application it is embedded in. At the same time the composing mechanism that forms applications from sets of components has to be powerful enough to capture the most common use cases, such as setting the port numbers wiring two components. In an ideal case, exchanging an SQL-based database in an application with a different SQL-based database should neither require changes to the invoking component definition nor to the new database component. Also, the application should be widely untouched except that the configuration parameters for the new database have to be set. Approaches to achieve this modularity include templating, parameterisation, and inheritance.

In order to facilitate easy *sharing* of entire models and parts

thereof, the tool should offer a marketplace where users can exchange their models with other users. If such a marketplace does not exist, the tool provider should at least offer application models for the most important standard services.

Apache Brooklyn Support: Brooklyn achieves the reusability of types by using inheritance. Yet types of the same parent can not be exchanged without modifying the concrete properties of connected types. Types can be shared either locally or in a Git repository.

Cloudify Support: Cloudify uses the same reusability mechanisms for its models as TOSCA: For the model is split into types and templates, defined types are in general usable in other templates. The separation of the server host and the application using the *hosted_on* relationship also decouples the server from the application description. The reusability is further increased by the import mechanism, that allows to define types in another file location as the templates and then import them. Another feature increasing modularity is the relationship mechanism, that allows a custom wiring for each type usage. Another mechanism that Cloudify shares with TOSCA is the inheritance of types. This allows the user to inherit from parent types, meaning that defined elements of the parent type are also defined in the child type. Finally, the input mechanism allows defining parameterised models. Cloudify does not offer a marketplace.

Apache Stratos Support: Since Stratos defines its configurations and applications in JSON, they can be shared as any text file. Yet, the cartridges contain references to IDs of IaaS snapshots and hardware configurations. Thus the reusability is limited to a cloud. Moreover, the definitions of an already deployed application can't be changed dynamically; it needs to be un-deployed first and then edit its definition. Similarly, Puppet is built to be reusable and shareable also. Its marketplace *Puppet Forge* contains more than $3,300$ modules, which can be added to a Stratos setup. The reusability of Puppet modules is gained by dependencies between modules, which allows splitting work in smaller but linked modules.

8) Containerisation Feature: The use of containers such like Docker is a reasonable approach for sharing a virtual machine between several component instances, while keeping them isolated. This leads to better utilisation of the virtual machine [7]. Moreover, the increased isolation offered by containers allows resource consumption to be configured, controlled, and limited on the level of component instances. This feature does not consider whether cloud providers use containers instead of hypervisors, as this is transparent for the users of the platform.

Apache Brooklyn Support: Brooklyn does not support containers out of the box. Yet, the separate project Clocker enables the usage of Docker.

Cloudify Support: Cloudify supports containerisation using Docker. The user can use a docker container node type what allows starting a docker container from a provided Docker image. The Docker container, then can be deployed on a virtual machine node using a *contained_in* relationship.

Apache Stratos Support: Stratos supports containerisation with Docker by using Google Kubernetes as a cluster orchestration framework.

9) Windows Application Support Feature: While Linux is dominating the cloud computing environment, there are many professional companies running their applications on a Windows operating system [8]. Hence, these applications should be supported by cloud management tools.

Apache Brooklyn Support: Brooklyn relies heavily on SSH which excludes native Windows support. Windows support is currently under development, though.

Cloudify Support: Cloudify supports Windows but requires that the virtual machine (image)s have WinRM enabled. Cloudify uses this protocol to install its agents on the machine. The agents then operate in an operating system independent manner.

Apache Stratos Support: Stratos cartridges for deploying applications also support Windows, e.g. for .NET framework applications. Both the Stratos agent running in the applications' virtual machines and Puppet support the Windows operating system.

V. Comparison Result

Table I depicts the achievements of the three cloud tools with respect to the features defined in Section IV. To be able to account for different partial achievement or different achievement quality we use a three-staged marker.

VI. Related Work

To the best of our knowledge there is currently no directly comparable work defining an in-detail comparison framework based on features and applying it to the given tools. The work of [9] and [10] present a general view of cloud computing defining characteristics, features and challenges of the cloud computing environment on a much higher level, from which our features are derived. There is also multiple work defining a taxonomy and doing a comparison of cloud computing systems [11] [12]. However, those comparisons focus on cloud computing in general. They hence put stress on features offered by and comparison of cloud providers. A qualitative and quantitive survey for the two IaaS management tools Openstack and Synnefo[5] is provided by [13]. [14] depicts an elaborate overview of frameworks, projects and libraries with respect to provisioning, deployment and adaptation of cloud systems, but also stays at a much coarser granularity. [15] compares multiple cloud brokerage solutions by first categorising them, and then listing their core capabilities. However, their comparison is also done on a much higher level. Finally, Paasify[6] gives a good overview of existing PaaS and PaaS-like offers doing a feature comparison on higher, quantifiable levels.

VII. Conclusion

In this paper, we have considered basic requirements of cloud orchestration tools and derived a fine-grained list of features any of the tools shall support. We also applied these results by rating three publicly available cloud orchestration tools based on our list: Apache Brooklyn, Cloudify, and Apache Stratos.

[5]https://www.synnefo.org/

[6]http://www.paasify.it

TABLE I. COMPARISON OF BROOKLYN, CLOUDIFY, AND STRATOS.

Features	Tools		
	Brooklyn	Cloudify (Pro)	Stratos
Cloud Features			
Multi-Cloud			
# of Cloud Providers	jclouds	3 (5)	jclouds
Abstraction Layer	✓	✗	**0**
Cross-Cloud	✓	✓	✓
External PaaS	✗	✗	✗
Cloud Standards	✗	✗	✗
BYON	✓	✓	✗
Application Features			
Model Standards	**0**	**0**	✗
Resource Selection			
Manual Binding	✓	✓	✓
Automatic Binding	**0**	✗	✗
Optimised Binding	✗	✗	✗
Dynamic Binding	✗	✗	✗
Life Cycle Description			
Shell Script	✓	✓	✗
# DevOps Tools	1	3	1
Wiring & Workflow			
Attribute & Event Passing	**0**	✓	✓
Manual Workflow	✗	✓	✓
Automatic Workflow	✗	✓	✗
External Services	✗	✗	✗
Monitoring			
System Metrics	✗	✓	✓
Application Metrics	✗	✓	✓
Custom Metrics	✓	✓	✗
Aggregation	**0**	✓	✓
Dashboard	**0**	✗(✓)	✗
Historical Data	✗	✗(✓)	✗
Runtime Adaptation			
Thresholds	✓	✓	✓
Rule Engine	✗	✓	✓
Horizontal Scaling	✓	✓	✓
Repair	**0**	✓	✓
Custom Action	✗	✓	✗
Continuous Delivery	**0**	✗	✗
Reusability and Sharing of Models			
Reusability	**0**	✓	**0**
Sharing	✓	✗	**0**
Containerisation	✗	✓	✓
Windows Support	✗	✓	✓

✗= not fulfilled, **0**= partially fulfilled, ✓= fully fulfilled

When looking at the results it becomes evident, that especially the resource selection feature is underrepresented in all three tools. All tools require the user to manually bind a concrete resource to the components at description time causing vendor lock-in due to missing abstraction and non-optimal placement due to an incorrect initial selection, performance unpredictability and no possibility to change it at runtime. Our own cloud orchestration tool CLOUDIATOR[7] [16], [17] has the goal to close this gap.

Future work will include the finalisation of a first release of our CLOUDIATOR tool based on the experiences gained while working with Brooklyn, Cloudify, and Stratos. In parallel, we will evaluate and rate more tools and we plan to extend this evaluation to ongoing research projects, also considering non-function features like performance, availability or security.

ACKNOWLEDGMENT

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 610711 (CACTOS), and from the European Community's Framework Programme for Research and Innovation HORIZON 2020 (ICT-07-2014) under grant agreement number 644690 (CloudSocket).

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

REFERENCES

[1] S. Kächele, C. Spann, F. J. Hauck, and J. Domaschka, "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*, ser. UCC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 75–82.

[2] P. M. Mell and T. Grance, "SP 800-145. the NIST definition of cloud computing," National Institute of Standards & Technology, Gaithersburg, MD, United States, Tech. Rep., 2011.

[3] DMTF, "Cloud infrastructure management interface (CIMI) model and RESTful HTTP-based protocol," 2013.

[4] Open Grid Forum, "Open cloud computing interface - core," 2011.

[5] D. Palma and T. Spatzier, "Topology and orchestration specification for cloud applications version 1.0," OASIS Standard, 2013.

[6] J. Durand, A. Otto, G. Pilz, and T. Rutt, "Cloud application management for platforms version 1.1," OASIS Committee Specification, 2014.

[7] K. Razavi, A. Ion, G. Tato, K. Jeong, R. Figueiredo, G. Pierre, and T. Kielmann, "Kangaroo: A tenant-centric software-defined cloud infrastructure," in *Proceedings of the IEEE International Conference on Cloud Engineering*, Tempe, AZ, USA, United States, 2015.

[8] Linux Foundation, "Enterprise end user trends report," 2014.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[10] R. Buyya, "Market-oriented cloud computing: Vision, hype, and reality of delivering computing as the 5th utility," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, ser. CCGRID '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–.

[11] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, ser. NCM '09. IEEE Computer Society, 2009, pp. 44–51.

[12] C. Höfer and G. Karagiannis, "Cloud computing services: taxonomy and comparison," *Journal of Internet Services and Applications*, vol. 2, pp. 81–94, 2011.

[13] E. Qevani, M. Panagopoulou, C. Stampoltas, A. Tsitsipas, D. Kyriazis, and M. Themistocleous, "What can OpenStack adopt from a Ganeti-based open-source IaaS?" in *2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014*, 2014, pp. 833–840.

[14] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, 2013, pp. 887–894.

[15] F. Fowley, C. Pahl, and L. Zhang, "A comparison framework and review of service brokerage solutions for cloud architectures," in *Service-Oriented Computing ICSOC 2013 Workshops*. Springer International Publishing, 2014, pp. 137–149.

[16] D. Baur, S. Wesner, and J. Domaschka, "Towards a model-based execution-ware for deploying multi-cloud applications," in *Advances in Service-Oriented and Cloud Computing*, ser. Communications in Computer and Information Science, G. Ortiz and C. Tran, Eds. Springer International Publishing, 2015, vol. 508, pp. 124–138.

[17] J. Domaschka, D. Baur, D. Seybold, and F. Griesinger, "Cloudiator: A cross-cloud, multi-tenant deployment and runtime engine," in *9th Workshop and Summer School On Service-Oriented Computing 2015*, 2015, in press.