MiCADO – Towards a Microservice-based Cloud Application-level Dynamic Orchestrator

Hannu Visti, Tamas Kiss, Gabor Terstyanszky, Gregoire Gesmier, Stephen Winter
Centre for Parallel Computing, University of Westminster, London, UK
(H.Visti, T.Kiss, G.Z.Terstyanszky,G.Gesmier, S.C.Winter)@westminster.ac.uk

***Abstract*—In order to satisfy end-user requirements, many scientific and commercial applications require access to dynamically adjustable infrastructure resources. Cloud computing has the potential to provide these dynamic capabilities. However, utilising these capabilities from application code is not trivial and requires application developers to understand low-level technical details of clouds. This paper investigates how a generic framework can be developed that supports the dynamic orchestration of cloud applications both at deployment and at run-time. The advantages and challenges of designing such framework based on microservices is analysed, and a generic framework, called MiCADO – (Microservices-based Cloud Application-level Dynamic Orchestrator) is proposed. A first prototype implementation of MiCADO to support data intensive commercial web applications is also presented.**

Keywords—Cloud applications, application-level orchestration, microservices-based architectures, container technologies.

I. Introduction

Many scientific and commercial applications require access to computation, data or network resources based on dynamically changing requirements. Applications running on distributed computing infrastructures, such as grids or clouds, typically fall into this category. End-users can access these applications via desktop or web-based high-level user interfaces, such as science gateways. When executing applications or accessing services via high-level user environments, users and providers both require these applications or services to dynamically adjust to fluctuations in demand and serve end-users at required quality of service and speed, and also at optimized cost. The challenge of developing such dynamically adaptable applications without requiring application developers to deal with low level details of the underlying distributed computing infrastructure is the topic of this paper.

One of well-documented benefits of cloud computing [1] is its ability to supply a variable amount of resources (computational power, storage, network capacity), which can scale dynamically up and down, forming the supply side of figure 1. On the demand side, we can see applications that are likely to be formed of one or more services. Services can be either in-house developed or provided by external suppliers or open-source communities. Services could also be shared between applications.

Services consume *baseline resources* that can be defined as resources consumed by the component in idle state. *Variable resources* are consumed when the component performs its duties. This can include heavy computation or storage based on application use, which can vary significantly based on the nature of the application.

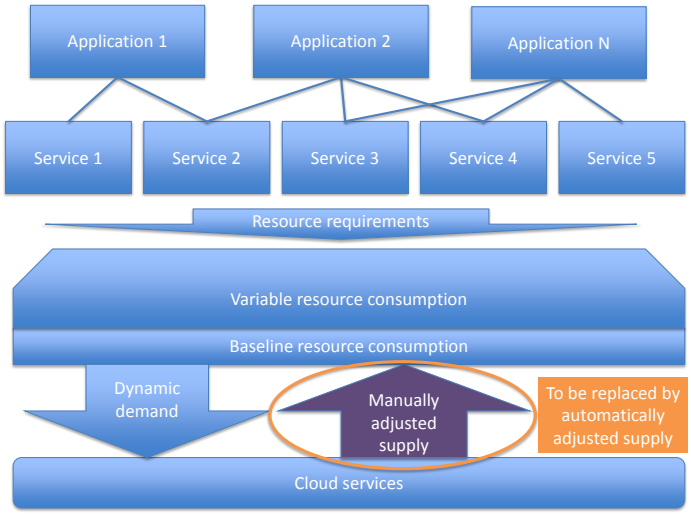

Fig. 1: Resource demand and supply of cloud applications

Overall resource demand is the sum of baseline and variable components, and the presence of a variable component makes the overall resource demand variable as well. IaaS (Infrastructure as a Service) clouds are elastic and have the ability to supply a variable amount of resources. However, applications need to be specifically programmed in order to utilise this elasticity and dynamically vary the amount of resources provided. Customising dynamic provision for each application individually for a specific cloud environment is costly to do. Our ultimate aim is to implement a generic service or layer that provides this functionality for any application automatically, and in a cloud resource agnostic way.

As shown in Figure 1, cloud service providers are indifferent to the resource needs of applications since they have no means of predicting application capacity demand behaviour. In practice, the operator of an application requests cloud resources based on static predictive estimates (typically, worst-case estimates), but after being commissioned, these resources remain static without operator intervention. If demand exceeds supply at a given point of time, applications do not function within their required parameters. If supply exceeds demand, resources are wasted, which generally has a cost impact.

This work investigates possibilities of replacing manually adjusted supply of cloud services with an automatically adjusted supply, as also illustrated in Figure 1. The aim is to create a framework, where automatically adjusted cloud service supply can be arranged, based on application component demands. Such framework would allow cloud application developers to build cost and performance optimization mechanisms into their application code through a high-level API.

The suggested solution is based on microservices and their dynamic orchestration in a cloud computing environment. The rest of this paper defines a generic microservices-based architecture for application-level cloud orchestration, called MiCADO (Microservices-based Cloud Application level Dynamic Orchestrator), and describes its first prototype implementation utilizing container-based open source cloud technologies.

II. Related work

The problem of application-level orchestration has been recognized and a number of solutions have been designed and implemented. Marpaung et al. [2] discuss Altocumulus, AppScale, Cloudify and mOSAIC. Altocumulus focuses on deploying web applications to variety of public clouds, which limits its usability in private or hybrid clouds. It does not provide monitoring or dynamic changes to services. AppScale is an open-source product that supports execution of Google Application Engine applications and therefore restricted to this particular technology. mOSAIC provides a set of APIs to application developers to tackle cloud deployment issues. The limitation of mOSAIC is the implementation of these APIs; the application developer needs to integrate these to application components.

Cloudify [3] combines a TOSCA editor with deployment and orchestrator. It provides access to multiple clouds and a complete framework to describe microservices and execute them either in Docker containers or on cloud metal. Cloudify also provides dynamic service upscaling and downscaling based on microservice dependent parameters, for example number of transactions, number of threads, etc. Cloudify does not provide a container portability framework, nor do its metrics span dockerised microservices and the cloud metal executing them.

Pham, Tchana et al. discuss the problem of distributed applications [4]. They focus on application orchestration that can span several different clouds. They recognize the need to deliver microservice-specific parameters, for example port numbers and IP addresses, to other microservices, and provide a description language framework to do this. However, this solution does not support service discovery tools and dynamic relocation of microservices.

Amazon CloudFormation [5] provides to system administration developers an easier way for the collection, creation and management of related AWS resources through templates. These templates describe the AWS resources and associated dependencies. After the deployment of AWS resources, CloudFormation ensures the start of services in the correct order.

OpenTosca [6] [7] provides an open source ecosystem for the OASIS Topology and Orchestration Specification for Cloud Applications developed by Stuttgart University. OpenTosca is divided into three parts: a TOSCA runtime environment (OpenTosca container), a graphical modelling TOSCA tool (Winery) and a self-service portal for the application available in the container (Vinothek).

Although effective in particular narrow circumstances, none of these systems provide a fully-automated, cloud-agnostic solution for all, or even a wide range of applications and for wide variety of clouds.

III. Opportunities and challenges of microservice-based architecture design in clouds

In order to support application-level orchestration and dynamic resource provision in clouds, a microservices-based architecture is proposed. This section highlights the motivations behind this approach and collects challenges that the architecture design needs to overcome.

Software design based around microservices, in contrast to a traditional monolithic architecture, is a relatively new concept, and the literature does not yet have an agreed definition for microservices. According to Newman “Microservices are small, autonomous services that work together”. His definition includes further characteristics, chiefly “focused on doing one thing well” and being “autonomous” [8]. Balalaie et al write “Microservices is [*sic*] a new architectural style [9] that aims to realize software systems as a package of small services, each deployable on a different platform, and running in its own process while communicating through lightweight mechanisms like RESTFull APIs.”

Both Newman and Balalaie et al. see microservices mainly as an agile development concept. A small, focused team can develop one component of an application independently and deploy new versions without changes to other components. In this paper, we focus on deployment of microservices-based architecture, where connectivity between microservices becomes challenging.

Cloud computing is a natural platform for microservices. Microservices achieve decoupling of independent components from a monolithic application. Clouds enable execution and resource allocation of these independent components based on their specific needs. One microservice might require a lot of storage while another could be CPU-intensive. Cloud execution offers the possibility to optimise resource allocation, and thus resource cost, dynamically. The alternative would be to allocate a monolithic infrastructure, the size of which would be large enough for a worst-case requirements scenario. However, for most of the time, the worst-case scenario does not prevail and allocated resources of the monolithic infrastructure are wasted.

As discussed earlier, microservices provide APIs to enable communication with them, and rely on APIs of other microservices to access other services, on which they are depending. A typical API connection is a TCP (Transmission Control Protocol) socket. To set up a TCP connection, the connecting computer needs two parameters: the IP address of the server host and a port number of the particular service on that host. Many port numbers are defined either officially in RFCs (Request For Comments), or by convention, and even in case of a completely new service, the port number would remain static within the particular architecture. The problem arises from dynamic IP address allocation strategy employed by the cloud.

To form an API connection, the client application needs to know the dynamically assigned IP address of the server host.

In a microservices-based architecture commissioned on cloud, the IP address information may change during the life-cycle of the application. In a microservices-based model, microservices can and will be developed independently. Moreover, cloud computing provides dynamic allocation of hardware resources. From performance and financial optimisation, it would be occasionally necessary to allocate more capacity by migrating a microservice to a more powerful platform, or to consolidate less regularly used microservices to a single platform and shutting down some cloud capacity. By doing this, the IP address of the server host would change.

An additional challenge arises if multiple microservices providing the exact same service are deployed on the same host. If the service API is provided in an established port, only one of the microservices could occupy this, and the others would need to be remapped. This would normally require changes to a configuration file. For example, if a single cloud instance hosted three MySQL implementations, only one of them could use the default port 3306. Running the other two would require configuration file changes, including additional changes to configuration files and service start-up scripts to assign a different location to data and configuration files.

One workaround would be to ensure that only one microservice of a kind could run on each cloud instance. This would remove the need to reconfigure port numbers and file locations, but it might waste resources if new cloud instances would be needed just for this purpose. Moreover, this would require building an additional coordination layer to keep track of running cloud instances and currently running microservices, to be used to allocate new microservices to cloud instances where such microservices do not currently run. It would also be entirely possible for two different services to occupy the same network port. While not common, there is no mechanism or policy that would prevent this, especially if microservices came from different sources and their development were independent. Thus, the aforementioned mapping would need to consider network services as well as file system locations to ensure there is no overlap between two microservices designed to share the same cloud instance. In Section V we will show such steps are unnecessary.

Obtaining benefits from the dynamic nature of cloud requires an understanding of current and/or predicted resource usage, and the possibility to scale infrastructure up or down on demand. A mechanism is needed to analyse usage, allocate microservices to cloud instances best suited to serve them, and allow optimisation based on given parameters, such as performance or cost. Microservices can set different requirements to physical cloud instances and resources based on the nature of the service. For example, an authentication service would require little CPU power or storage, an HTTP application server would require considerable CPU power but little storage, and a database server could require both powerful computation *and* storage capabilities.

Setting up a static microservices based architecture on cloud would be possible manually, by designing the correct launch order of services, and configuring manually IP addresses of the already started services to those depending on them. If more capacity were needed or any changes to the infrastructure were initiated by changes in cloud services provisioning, an operator would need to reconfigure microservices API information manually. This could in many cases lead to suboptimal service performance or waste, as from cost and workload management viewpoint it might be easier to react to changed requirements only when absolutely necessary. Automated monitoring and control could thus enhance benefits enabled by the use of cloud and microservices.

The above detailed challenges can be summarized as follows:

C1. Discovery of IP addresses and port numbers of running microservices.
C2. Ability to run several microservices of the same kind on a single cloud instance.
C3. Allocation of microservices to cloud instances based on available resources.
C4. Restriction mechanism to ensure microservices can only be allocated to cloud instances that have the functionality and capabilities of serving them.
C5. Ability to dynamically scale up and down. If cloud infrastructure usage exceeds given parameters, the condition must be detected and more cloud resources need to be allocated.

This work proposes an architecture that helps mitigate the aforementioned challenges. The approach is also agnostic to chosen applications, cloud providers, and microservices needed to provide applications.

IV. MiCADO – THE PROPOSED SOLUTION

The proposed solution places microservices in lightweight virtualisation containers in worker nodes. These containers can be hosted on any of the different worker nodes, and one node can run one or more containers. An orchestration and coordination mechanism is required to enable service discovery and performance management. The overall MiCADO (Microservices-based Cloud Application level Dynamic Orchestrator) architecture is illustrated in Figure 2. The solution takes the following overall approaches to challenges identified in section III:

C1. To address dynamic discovery of IP addresses and ports, use a service discovery framework. Several such frameworks exist, for example Consul [10], Zookeeper [11] and Etcd [12].
C2. To enable deployment of several microservices of the same kind on the same instance, use kernel namespace based lightweight virtualisation solution to run microservices in containers. These solutions (for example Docker [13] built on Linux containers) provide separation of application files and allow port mapping functionality to masquerade standard microservice API ports to dynamically allocated ones.
C3. To support the allocation of microservices to cloud instances based on available resources, form a cluster of worker nodes that is aware of the capacity and status of each node, and is able to make service allocation decisions based on a documented logic.
C4. The chosen clustering mechanism must be aware of the hardware configuration of cloud instances, and allow constraints to be set to limit automatic allocation decisions

(C3) to those nodes that have the requisite physical capabilities, for example a public IP address, to serve a particular microservice.

C5. To support automatic scaling up and down, the cluster needs to have an alert mechanism, to help cluster management detect nodes and services that exceed thresholds or nodes that appear to be underutilised. Decision-making logic needs to be programmed based on this information or obtained within a software component. The logic needs to have interfaces to clouds to start and shut down instances, and to container start-up and shutdown mechanisms.

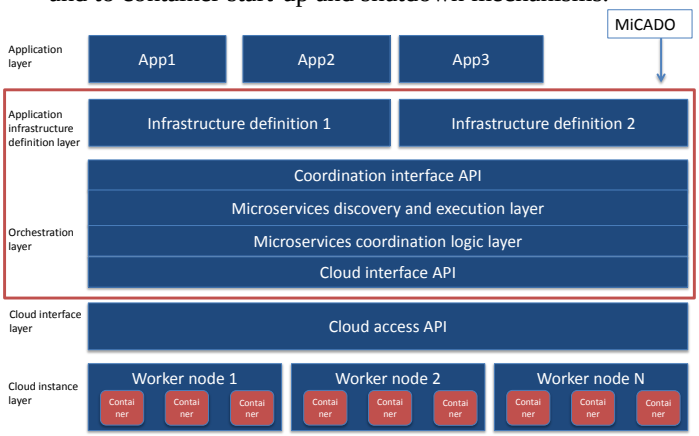

Fig 2: Generic MiCADO architecture

Based on the above described generic approaches to handle the five identified challenges, the following layered architecture is proposed for the application level orchestration of cloud applications. The layers in the box entitled MiCADO represent the actual orchestration architecture. Layers below MiCADO provide access to cloud resources, while the top layer represents actual applications to be optimized. The layers in Figure 2 are described from bottom to top.

1) **Cloud infrastructure layer.** This layer contains cloud instances, which in turn run containers that execute actual microservices. One instance can run one or more containers.
2) **Cloud interface layer.** This is a set of APIs that provides means to launch and shut down cloud instances. There can be one or more cloud interfaces to support multiple clouds. Either native interfaces of targeted clouds can be applied (e.g. EC2) or generic cloud access layers that provide access to multiple heterogeneous clouds.
3) **Microservices orchestration layer.** This is divided into four sub-layers.
   a) *Cloud interface API.* This layer is needed to abstract cloud access from the layers above. Cloud access APIs can be complex interfaces, as they typically cater for a large number of services provided by the cloud provider. On the other hand, the microservices execution and coordination logic layers (see 3b and 3c) only need to shut down and start instances. Abstracting this to a cloud interface API simplifies the implementation of the aforementioned layers, and equally, if new Cloud access APIs are implemented, only this layer needs to change.
   b) *Microservices coordination logic layer.* With large infrastructures, and to reap the benefits from cloud-based execution, it becomes necessary to understand how the current execution environment is performing. Information needs to be gathered and processed. If bottlenecks are detected or the currently running infrastructure appears underutilised, it may be necessary to either launch or shut down cloud instances, and possibly move microservices from one physical worker node to another.
   c) *Microservices discovery and execution layer.* This layer manages the execution of microservices and keeps track of services running. Execution management combines both start-up and shut-down of microservices. Service management gathers information about currently running services. Information gathered includes service name, IP address and port where the service is reachable, and optional service tags to help in service coordination.
   d) *Coordination interface API.* This layer provides access to orchestration control and decouples the orchestration layer from the application infrastructure definition. This set of APIs enables application developers to utilise the dynamic orchestration capabilities of the underlying layer and support the convenient development of dynamically and automatically scalable cloud-based applications.
4) **Application infrastructure definition layer**. This forms the basis for creating a functional application infrastructure. At this level, software components and their requirements as well as their interconnectivity are defined. This layer does not contain any application-specific data. For example, to provide HTTP based services, this layer can define that to provide this functionality, a MySQL database, Apache HTTP server and Nginx proxy server are needed, and that Nginx needs connection to Apache, which in turn needs connection to MySQL. As the infrastructure is agnostic to the actual application using it, this definition can be shared with any application that requires such an environment.
5) **Application layer**. This layer contains actual application code and data to make an incarnation of a defined application infrastructure (4) function in such a way that the desired functionality is achieved. For example, this layer could populate a database with initial data, and configure an HTTP server with both look and feel and application logic.

V. MiCADO Reference Implementation

A prototype of the proposed MiCADO architecture has been implemented via a combination of open-source tools and in-house developed extensions, as is illustrated in Figure 3. Although it would be possible to develop a platform to implement MiCADO functionality from scratch, reusing and utilizing existing open-source components has significantly speeded up the implementation process. As a restriction, this first MiCADO prototype does not include Cloud interface and Coordination interface APIs. The Coordination interface API is replaced by a simple command line, and the Cloud interface API is represented by direct calls towards the cloud access layer (in this case the CloudBroker Platform [14] [15]). Moreover, instead of a generic approach towards infrastructure definition, a concrete architecture has been realized implementing a real-life case-study. Applying these limitations, a proof-of-concept implementation has been completed to justify the validity of the proposed approach when handling the five challenges detailed in section III. For infrastructure management and orchestration the

following tools have been selected to implement the previously described MiCADO layers.

Microservices discovery layer was implemented based on Consul [10]. Consul is an open-source service discovery tool that also includes health check and alerts functionality. An additional component, Registrator [16] is used on worker nodes to register information on running containers to Consul.

Consul agents can work either in a server or serf role [17]. Servers keep track of events happening in the network and notify other servers of changes, while serfs create a server connection and maintain a list of backup servers. Consul servers create network overhead by communicating between themselves using Consensus protocol [18]. In large environments, most nodes should be configured as serfs to minimise this traffic. In the proof of concept example, the master and all nodes are configured as servers, to keep node configurations identical.

When a Consul agent starts, it needs to connect to an existing network. This can be provided either by a specific Consul server set in *bootstrap mode*, by connection to a node or set of nodes already running Consul, or by connection to Atlas service [19]. Atlas is a service that registers running Consul infrastructures based on a secret token. When a Consul agent configured to use Atlas starts, it registers itself with the service and queries for other nodes already present. The service is provided via a REST API, making it easy to use from firewalled networks. The first node registering does not receive anything. When the second node joins the infrastructure, it registers itself and receives in response the details of the first node, allowing it to initialise connection to it. When a server finds one Consul server, it communicates with it and shares information about the entire network. Thus, a Consul network does not need to be described in nodes, only the bootstrap information needs to be present.

Registrator is started on all worker nodes. Registrator is a helper process to Consul that registers all running Docker containers to it. Registrator runs in a Docker container itself, and it can be started either locally or via the Swarm launch mechanism. The roles of Docker and Swarm will be explained in detail under the Microservices coordination logic layer.

Microservices coordination logic layer was implemented based on Docker and Swarm [20]. Swarm is a clustering mechanism built on Docker that is aware of worker nodes and their current workload, and is able to allocate new containers to the node currently least used. Docker and Swarm allow the capability of giving worker nodes "tags" to enable the use of constraints. For example, giving a different tag to nodes with a lot of disk space allows allocation of database containers to these particular nodes, while worker nodes tagged for proxy use will have a public IP address. To fulfil requirement C4, worker nodes are tagged with their roles. The tags used in the current application scenario are *proxy*, *application* and *database* (see Application infrastructure definition layer later on). This tag is set in the Docker start-up configuration file.

Cloud Access API layer is represented by calls to the Cloudbroker Platform. The Cloudbroker Platform is a production level tool developed by a CloudBroker GmbH that allows interfacing to various commercial IaaS clouds, for example Amazon and CloudSigma, and also to private cloud infrastructures based on Openstack, OpenNebula and Eucalyptus [21]. An installation of the platform that is operated by the CloudSME European Project [22] was utilized in our experiments. Using the CloudBroker Platform, even this very first MiCADO implementation is capable of interfacing with a large variety of clouds without further cloud interface plug-in development.

For **cloud instances**, an Ubuntu 14.04 image was prepared with Docker, Consul, Swarm and Registrator installed. The CloudBroker platform provides a mechanism for preparing such images to be later launched in participating clouds. In addition to pre-installed applications, the cloud image needs a start-up script to initialise and launch the above mentioned tools. Consul and Docker create, upon initial start, random identifiers to distinguish between nodes in the same cluster. When a cloud instance launches the first time, it needs to remove these identifiers to force initialisation.

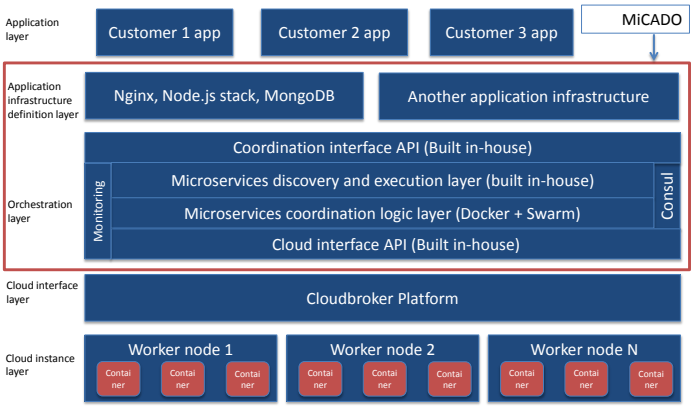

Fig. 3: MiCADO reference implementation

Application infrastructure definition layer currently implements one particular topology based on a specific set of use-cases provided by Outlandish LLP [24]. Outlandish is a small employee-owned digital agency specialising in middleware, usability, search and scalable data applications. Outlandish develops and hosts various web applications for multiple corporate clients. Usage estimates vary greatly between applications and customers, and in some cases, usage activity can be expected to vary greatly based on external triggers, such as calendar, time of day, outside events and news. The company wishes to serve its clients to agreed quality of service but also tries to minimise its costs on the provision side. Therefore, a cloud-based application deployment and hosting environment that is capable of making such optimisation choices dynamically at run-time is highly desired.

A typical Outlandish application is formed of three components: proxy servers with public IP address, application servers and database servers, as illustrated in Figure 4. The concrete implementation applies Nginx reverse proxy servers, Node.js stacks for application servers, and MongoDB databases. This set-up is generic for several Outlandish-developed and -hosted applications and as a consequence, several application scenarios and clients can be served based on this application pattern.

The currently implemented MiCADO prototype successfully deploys the above described generic architecture for several Outlandish applications into Docker containers, clusters these containers based on Swarm, and provides the required service discovery mechanism via Consul and Registrator. Outlandish

application developers need only provide the application code and the corresponding Docker configuration files. The optimised deployment to available cloud instances is managed by MiCADO. Therefore, the implemented solution successfully addresses challenges C1-C3 (IP address and port number discovery, running same kinds of microservices on a cloud instance, and allocating microservices to cloud instances based on available resources), and also by tagging the resources it provides mechanisms to support C4 (allocate microservices to specific instances that have the capabilities to serve these). However, please note that this current prototype does not offer run-time monitoring and automated optimisation capabilities (C5) that is part of our future work.

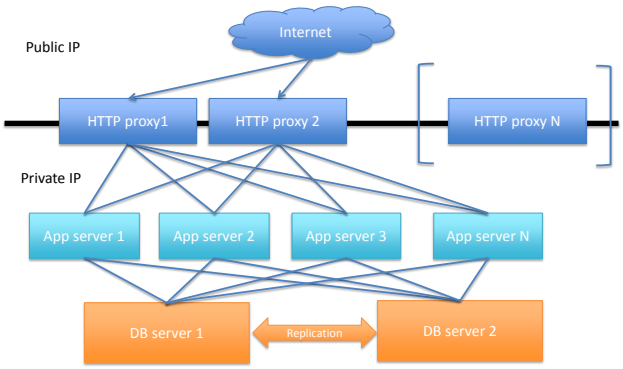

Fig. 4: Web-based application infrastructure example implemented for the Outlandish use-cases

VI. Deploying and Managing Applications with MiCADO

Based on the above described first proof-of-concept reference implementation of the MiCADO framework, a short overview on how this implementation supports the deployment and management of cloud applications is provided in this section. Please note that the current implementation supports only command-line access. However, the objective of our future research is to develop a set of well-defined APIs that can be conveniently embedded into desktop or web-based science gateway solutions.

Before an application developer can utilize MiCADO services, certain infrastructure components need to be deployed and set-up by an administrator. A specific set of scripts has been developed [25] that can be used to create and deploy the necessary virtual machine images on the CloudBroker Platform. MiCADO differentiates between two types of virtual machines: manager and worker nodes. The manager node runs permanently until the infrastructure is required and can be used to deploy or terminate microservices running in containers, or cloud instances. Worker nodes are hosting various Docker containers running different microservices (e.g. applications, databases or proxies, as illustrated in figure 4). Both manager and worker node images include Docker, Consul and Swarm.

In order to create a new MiCADO installation, the administrator needs to launch a manager node instance through the CloudBroker platform in the target cloud. Please note that as the CB platform is used at the cloud access layer, the solution is cloud-agnostic, and the manager node can be launched in any targeted could in which an image has been created. Once the manager node is launched, application developers can log-in to this instance to manage their applications formed of multiple microservices.

Application developers first need to create a Docker image of the specific microservice they want to deploy (for this process no specific support is provided currently in MiCADO). This Docker image is then uploaded to a private Docker registry operated by the CloudSME project. In order to launch a new Docker container with a required image, application developers need to execute a Python script from [25] providing the name of the image and the type of node (e.g. node with public IP address for a Proxy server) as parameters. The container will be launched to the least utilised node automatically by Swarm. When a new container is launched, MiCADO checks whether there is enough memory available on current cloud instances to run the container. If there is not enough memory available, then a new instance is launched automatically, and the launch of the container will be held until the new cloud instance is up and running on the infrastructure.

The current MiCADO prototype also supports automated downscaling of the infrastructure at run-time. There is a cron job running continuously to determine if the infrastructure needs to be downscaled. This cron job will first establish which instance on the infrastructure is the least utilized. It will then list all the running containers on the least utilized instance and get their memory usage. The cron job will check if the memory usage on the instance is less than the availability on the rest of the infrastructure. If there is enough memory available on the other running instances then it will check if each container can be relocated on another instance. If the containers can be relocated then these are launched on the other instances, the containers on the least utilized instance are stopped and removed, and finally the least utilized instance is stopped and removed from the infrastructure. With this functionality MiCADO provides some basic capabilities to optimize resource utilization and therefore save unnecessary expenses for application providers.

VII. Conclusions and Future Work

This paper collected and analysed challenges towards a microservices-based implementation of a dynamic application level cloud orchestrator called MiCADO. Moreover, it presented a first proof-of-concept implementation of the architecture that successfully addresses four out of the five identified challenges, and also facilitates a future implementation to tackle the fifth challenge. The MiCADO concept succeeds in decoupling application level logic from cloud orchestration logic, while preserving the elastic and dynamic nature of the cloud. The experiment shows it is possible to narrow the gap between IaaS and SaaS models.

While the solution was tested on the infrastructure described earlier, there is no fundamental reason why the principles could not be applied to other operating systems, monitoring tools, service discovery tools and cloud APIs. The most significant limitations are in security, performance measurement and downscaling. No work has been done on infrastructure level security. Network overhead incurred by Consul and Swarm has not been investigated, nor has the CPU overhead caused by Docker. Consul has a built-in mechanism to reduce network overhead by designating only a few nodes as servers and the rest as serfs. Therefore, more work would be needed to define the optimal configuration. Moreover, performance measurement has

been done on a most basic level, to prove a concept instead of providing a ready solution.

As this paper only analysed the MiCADO concept and presented a proof of concept implementation, future work is rather versatile. On the one hand it should evolve around measurement and optimization logic of dynamic scalability. As this is the core of the MiCADO platform, making this more versatile and intelligent without sacrificing usability is a challenge. Another aspect includes infrastructure definition reusability and sharing. This could lead to a definition of application topology and description templates that are applicable for a wide range of application scenarios. For such purposes we are looking at existing standardized description formats, such as TOSCA (Topology and Orchestration Specification for Cloud Applications) by OASIS [23]. Another angle for future work is information security related to service up/downscaling and portability, as this has not been touched at all.

VIII. Acknowledgements

This work was funded by the CloudSME Cloud-Based Simulation platform for Manufacturing and Engineering Project No. 608886 (FP7-2013-NMP-ICT-FOF). We also acknowledge the contribution of Outlandish LLP whose staff provided us with vital information regarding their requirements and valuable feedback throughout the implementation.

References

[1] Thomas, D. (2009). Cloud Computing — Benefits and Challenges! *The Journal of Object Technology*, *8*(3), 37.

[2] Marpaung, Sain, & Hoon-Jae Lee. (2013). Survey on middleware systems in cloud computing integration. *Advanced Communication Technology (ICACT), 2013 15th International Conference on Advanced Communications Technology*, 709-712.

[3] Anon, (2016). *Cloudify* [online] Available from: <http://getcloudify.org> [Accessed 16/03/2016]

[4] Manh Pham, L., Tchana, A., Donsez, D., Zurczak, V., Gibello, P., & De Palma, N. (2015). An adaptable framework to deploy complex applications onto multi-cloud platforms. *The Institute of Electrical and Electronics Engineers, Inc. (IEEE) Conference Proceedings*, 169-174.

[5] Anon. (2016). *AWS CloudFormation* [online] Available from: <https://aws.amazon.com/cloudformation/> [Accessed 16/03/2016]

[6] Anon. (2015). *OpenTosca* [online] Available from: <http://www.iaas.uni-stuttgart.de/OpenTOSCA/#ecosystem> [Accessed 16/03/2016]

[7] Binz, Breiter, Leyman, & Spatzier. (2012). Portable Cloud Services Using TOSCA. *Internet Computing, IEEE, 16*(3), 80-85.

[8] Newman, Sam. (2015). *Building Microservices*. O'Reilly.

[9] Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2015). Migrating to Cloud-Native Architectures Using Microservices: An Experience Report.

[10] HashiCorp. (2016). *CONSUL by HashiCorp* [online] Available from: <https://www.consul.io> [Accessed 16/03/2016]

[11] Apache. (2016). *Apache Zookeeper* [online] Available from: <https://zookeeper.apache.org> [Accessed 16/03/2016]

[12] Coreos. (2016). *Etcd* [online] Available from: <https://coreos.com/etcd/> [Accessed 16/03/2016]

[13] Anon. (2016). *Docker* [online] Available from <https://www.docker.com> [Accessed 16/03/2016]

[14] Anon. (2016). *CloudBroker Platform* [online] Available from: <http://cloudbroker.com/platform> [Accessed 15/03/2016]

[15] Taylor SJE, Kiss T, Terstyanszky G, Kacsuk P, Fantini N. (2014). Cloud Computing for Simulation in Manufacturing and Engineering: Introducing the CloudSME Simulation Platform. *Proceedings of the 2014 Annual Simulation Symposium, ANSS '14, Society for Computer Simulation International: San Diego, CA, USA, 2014*; 12:1–12:8.

[16] Anon. (2016). *Registrator* [online] Available from: <http://gliderlabs.com/registrator/latest/> [Accessed 16/03/2016]

[17] Anon. (2016). *Consul Architecture* [online] Available from: <https://www.consul.io/docs/internals/architecture.html> [Accessed 16/03/2016]

[18] Anon. (2016). *Consensus Protocol* [online] Available from: <https://www.consul.io/docs/internals/consensus.html> [Accessed 16/03/2016]

[19] Anon. (2016). *Atlas Integration* [online] Available from: <https://www.consul.io/docs/guides/atlas.html> [Accessed 16/03/2016]

[20] Anon. (2016). *Docker Swarm Overview* [online] Available from: <https://docs.docker.com/swarm/overview/> [Accessed 16/03/2016]

[21] Baset, Salman, A., Carey, Michael, & Hand, Steven. (2012). Open source cloud technologies. *Cloud Computing Proceedings of the Third ACM Symposium*, 1-2.

[22] Kiss, T, Dagdeviren, H, Taylor, JES, Anagnostou, A, Fantini, N. (2015). Business Models for Cloud Computing: Experiences from Developing Modeling & Simulation as a Service Applications in Industry, *Proceedings of the 2015 Winter Simulation Conference*, Pages 2656-2667, ISBN: 978-1-4673-9741-4, IEEE Press

[23] Oasis committee specification (2013). *Topology and Specification for Cloud Applications version 1.0* [online] Available from: <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html> [Accessed 15/03/2016]

[24] Anon. (2016). Outlandish [online] Available from < http://outlandish.com/ > [Accessed 19/03/2016]

[25] Anon. (2016). *MiCADO setup scripts* [online] Available from < https://github.com/gregGes/DSCR-infrastructure> [Accessed 08/05/2016]