Received 7 February 2018; | Revised ; | Accepted

DOI: xxx/xxxx

RESEARCH ARTICLE

Orchestrator Conversation: Distributed Management of Cloud Applications

Merlijn Sebrechts | Gregory Van Seghbroeck | Tim Wauters | Bruno Volckaert | Filip De Turck

[1] Ghent University - imec, IDLab, Department of Information Technology, Ghent, Belgium

Correspondence
Merlijn Sebrechts, Ghent University - imec, IDLab, Department of Information Technology, Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium. Email: merlijn.sebrechts@ugent.be

Summary

Managing cloud applications is complex, and the current state of the art is not addressing this issue. The ever-growing software ecosystem continues to increase the knowledge required to manage cloud applications at a time when there is already an IT skills shortage. Solving this issue requires capturing IT operations knowledge in software so that this knowledge can be reused by sysadmins who do not have it. The presented research tackles this issue by introducing a new and fundamentally different way to approach cloud application management: a hierarchical collection of independent software agents, collectively managing the cloud application. Each agent encapsulates knowledge of how to manage specific parts of the cloud application, is driven by sending and receiving cloud models, and collaborates with other agents by communicating using conversations. The entirety of communication and collaboration in this collection is called the *orchestrator conversation*. A thorough evaluation shows the orchestrator conversation makes it possible to encapsulate IT operations knowledge that current solutions cannot, reduces the complexity of managing a cloud application and happens inherently concurrent. The evaluation also shows that the conversation figures out how to deploy a single big data cluster in less than 100 milliseconds, which scales linearly to less than 10 seconds for 100 clusters, resulting in a minimal overhead compared to the deployment time of at least 20 minutes with the state of the art.

KEYWORDS:
Cloud modeling languages, Orchestration, Distributed management, Configuration management, TOSCA, Big Data

1 | INTRODUCTION

Managing cloud applications is complex. System Administrators (sysadmins) need to have an in-depth understanding of all the components of the cloud application such as the operating system, webserver, X.509 certificates and more. Having such deep knowledge about how to deploy, configure, monitor and manage these components is almost impossible in the field of big data because of the size of the ecosystem, the complexity of the tools involved and the rapid pace of innovation. This would not be such a big problem if it was not for the large skills shortage in the fields of IT operations[1] and big data[2]. There is thus a need for the ability to share and reuse the knowledge of sysadmins across teams and companies.

Sharing IT knowledge is not a new concept. The field of software development, for example, has a big focus on sharing and reusing knowledge in the form of code libraries. Over the years, a vast number of code libraries have been created that encapsulate an enormous amount of knowledge. Developers use these libraries to quickly write software without having to know each and every detail of how the software works. As an example, a programmer writing a piece of software that communicates over HTTPS does not need to know the intricate details of the TCP/IP protocol, X.509 certificates and SSL encryption. By simply using a library that implements all these functions, the programmer can focus on the actual novel parts of the application.

Two properties of programming languages are key enablers for this knowledge reuse: the ability to encapsulate code and to create new abstractions. **Encapsulation** allows developers to group code with a common function into a reusable module. By creating an **abstraction**, the developers expose the functionality of that module over an API that hides the inner complexity. An important property of these two is that they are stackable: a module can have varying levels of abstraction where each level encapsulates and hides the complexity of the level below it. As an example, a library for HTTPS communication might encapsulate three libraries: one for TCP/IP communication, one for X.509 certificate management and one for SSL encryption. Note that it is not sufficient for a programming language to be an abstraction of machine code, because it only allows developers to reuse the knowledge of the creators of the programming language, not the knowledge of other developers. It is vital the programming language itself allows for developers to create *new* abstractions *with* the language instead of *in* the language, so that they can easily encapsulate their own knowledge in a reusable way.

Knowledge reuse is regrettably a lot less prevalent in IT operations since it has historically been a mostly manual job which makes capturing knowledge hard. The rising popularity of configuration management automation provides new opportunities. Automation tools such as Chef and Puppet allow sysadmins to develop code that manages a cloud application. Although this code captures the sysadmin's knowledge, it does not enable knowledge reuse because building on top of automation code requires the same knowledge as creating the code in the first place. This issue stems from the foundational theory behind configuration management tools: converging towards a predefined end-state, as popularized by Burgess et al.[3] The idea of convergence is that a sysadmin specifies the desired end-state of an application and the configuration management tool executes the necessary actions to get the application into that state. The automation code is in that sense a description of the desired end state of the application. The same code thus always results in the same end state. Having the code figure out what the end state needs to be, is not possible in this convergent approach to configuration management. This has the advantage of consistency and reliability, but this presents a big issue for creating reusable automation code modules. When an automation code module changes a property of an application, there are two options. Either the module hides the value of that property in an abstraction, but then the value is static so the module cannot be used in a scenario where that property needs a different value, or the module exposes the property in its API, causing the module to leak its complexity. As a result, a module is either *understandable*, or *flexible*, but it cannot be both, even though both are important for successful reuse according to a systematic mapping study by Bombonatti et al.[4]

Note that it is entirely possible, and common, for the desired end-state to be an abstract description of the actual end-state. However, it becomes the responsibility of the configuration management tool to translate the abstract description into a concrete description. The only abstractions possible in configuration management tools are those provided by the creators of the tools. Sysadmins can thus only reuse the knowledge of the creators of the automation tools, not the knowledge of other sysadmins.

A recent addition to cloud application management is the concept of topology-based cloud modeling languages such as the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA). On top of automation, these languages also provide ways to model the cloud application as a set of interdependent services, and they abstract the underlying cloud to prevent cloud vendor lock-in[5]. These languages are typically paired with an *orchestrator*, a program that interprets the model and performs the necessary management actions.

Reducing the complexity towards end-users, however, requires more than abstracting the cloud itself: it requires creating new abstractions using the cloud modeling language, which is still not possible. As an example, modeling and managing a Hadoop cluster as a single entity is not possible. System administrators need to model each individual component of a Hadoop cluster and their dependencies: the Namenode and Datanode to get an HDFS cluster, and the ResourceManager and NodeManager to get a YARN cluster, the Datanode and NodeManager must be co-located, etc. This also causes sysadmins to be responsible for translating higher-level objectives such as "scale the Hadoop cluster" into actions on the individual components of the cluster and requires them to have in-depth knowledge of the inner workings, i.e. on what it means to scale a Hadoop cluster.

Simply making it possible to create abstractions in cloud modeling languages is not the whole solution, however, because someone still needs to create these abstractions and encapsulate knowledge. The answer to the "who" question is complex because there are multiple parties involved in the creation and management of a cloud application:

- the system administrator that manages the cloud application,
- the Independent Software Vendor (ISV) of the software that makes up the cloud application,
- and the vendor of the orchestrator that interprets the cloud model.

The sysadmin is the expert on how the software is used in the cloud application, but it is the ISV who has the actual knowledge on how to manage the individual software artifacts, so the ISV is the prime candidate to encapsulate the knowledge. Note that the transfer of knowledge from ISV to sysadmins currently happens almost exclusively using documentation and tutorials. The ISV writes documentation and tutorials, and the sysadmin uses that documentation to manage the software. This means that the most important role of the cloud modeling language and orchestrator is to provide a platform to enable sysadmins to use the expertise of the ISVs. The orchestrator vendor cannot decide what the requirements are, since that is the role of the sysadmin, and the orchestrator vendor should not decide how individual components are best managed, since that is the role of the ISV. This is regrettably not the case with current cloud modeling languages. As an example, if a sysadmin uses the current declarative capabilities of the TOSCA cloud modeling language[6], then it is up to the orchestrator to interpret that model and manage the cloud application accordingly. This is the exact opposite of the ideal scenario, where the role of the orchestrator is as minimal as possible.

In summary, the issue with the current generation of cloud application management tools is that these tools do not enable sysadmins to reuse the knowledge of ISVs. The orchestrator conversation proposed in this article tackles this issue by introducing a new and fundamentally different way to approach cloud application management: the *orchestrator conversation*, a hierarchical collection of independent software agents that collectively manage the cloud application. Each agent is an orchestrator that manages a specific part of the cloud application, collaborates with other agents over peer-to-peer conversations and deploys new agents to delegate tasks to.

Three features of the orchestrator conversation are key to addressing the aforementioned issue.

a) The ability to encapsulate automation code that **manages a single service** and to provide that functionality over an abstract API makes it possible to reuse the knowledge of how to manage a single service.

b) Encapsulation of automation code that **orchestrates a number of services** and providing that functionality over an abstract API, which enables reusing the knowledge of how to orchestrate multiple interconnected services.

c) The orchestrator conversation enables multiple independent ISVs to encapsulate their knowledge **in an interoperable way** so that sysadmins can build a cloud application using multiple components from multiple ISVs.

The remainder of this article is structured as follows. Section 2 gives an overview of the state of the art concerning this field and identifies how the related work has influenced this article. Section 3 explains the concepts behind the orchestrator conversation, and how they address the identified issues. Section 4 evaluates the solution using simulations and Section 5 concludes this article.

2 | RELATED WORK

There is a lot of different terminology being used regarding the management of cloud applications. Since these terms are not always used in a consistent manner, this section starts with a description of how these terms are used in the context of the presented research. This generally follows the cloud resource orchestration taxonomy proposed by Weerasiri et al.[7]

A *cloud application* consists of a number of *services* connected by *relationships*. A relationship generally denotes a dependency and/or an interaction between two services. An *application topology* is a description wherein the cloud application is described as a graph of nodes (the services), and edges (the relationships). *Cloud resource orchestration* is the process of ***selecting***, ***describing***, ***deploying***, ***configuring***, ***monitoring*** and ***controlling*** the infrastructure and services that make up the cloud application. Cloud resource orchestration is abbreviated as *orchestration* and is interchangeable with *cloud application management* in the context of this research. An orchestrator is a piece of software that performs orchestration tasks. [1]

[1]Note that "orchestration" does not imply a central controlling entity. There is thus no distinction between orchestration and choreography in this context.

A number of efforts from different fields try to tackle the issues regarding management of cloud applications. The remainder of this section explores how each field addresses different issues, how this relates to knowledge reuse, abstraction and collaboration, what our research takes away from these efforts, and how it goes beyond the state of the art.

2.1 | Resource scheduling

The lines between cloud application management and resource scheduling are starting to blur, resulting in a number of innovative solutions regarding cloud application management to come out of the resource scheduling field. The reason for this evolution is that there is a lot more to the management of a cluster than simple job scheduling. Jobs are part of larger applications that have complex topologies and dependencies. Jobs have a complex lifecycle, they need to communicate with each other and they need to be configured[8].

One of the big lessons learned from this field is that monolithic schedulers evolve into complex hard-to-maintain systems because of the increasing heterogeneity of resources and jobs and the widening range of requirements and policies. This problem is addressed by pulling the monolithic cluster manager apart into a number of specialized schedulers that work together on shared resources and job queues[9]. Apart from solving the complexity issue, this approach also makes it possible to have multiple schedulers from different vendors manage shared resources[10].

The big shortcomings of the state of the art in cluster management are the lack of native support for grouping workloads into application topologies[8], automatic dependency management and dynamic reconfiguration of workloads[11]. The need for dynamic reconfiguration is inherent to configuration itself since configuration solely exists to make hard-coded parameters changeable. If a parameter needs the same value for every deployment of an application, then this parameter will simply be hard-coded in the application's source code. The advent of microservices has only increased the complexity of application topologies and their configurations and dependencies, making it impossible to manage these manually[12].

2.2 | Cloud modeling languages

Cloud modeling languages provide a standardized format to ***describe*** cloud applications and their metadata. The sysadmin creates a model that describes the desired state of the cloud application and the orchestrator ***deploys*** and ***configures*** the cloud application according to that description. These languages thus have enormous potential to encapsulate cloud application management knowledge in a reusable manner. The latest generation of cloud modeling languages is centered around describing the application as a topology of components and their relationships to each other. The TOSCA language is an effort to reduce vendor lock-in by separating the cloud modeling language from the cloud provider and cloud infrastructure platform. This effort resulted in a push towards abstraction in cloud modeling languages. Abstraction of individual components is possible using TOSCA "node templates". The orchestrator substitutes a node template by concrete node types before deployment[6]. Brogi et al. identified four different matching strategies for transforming these abstract node templates into concrete node types[13]. These strategies can be used to combat vendor lock-in by using standardized vendor-neutral node-templates[14]. The downside is that this substitution is a one-way process which results in critical information loss. The resulting topology does not contain any information about what node templates were present nor what node types correspond to what node templates. As a result, a sysadmin can only use node templates during the deployment phase. After the deployment is done, the abstractions are lost and the sysadmin is exposed to the full complexity of the cloud application making the monitoring and controlling phases very complex.

A useful feature of the TOSCA approach is that it is possible to create a topology that contains both node types and node templates, thus having multiple levels of abstraction in a single model. This allows sysadmins to choose the appropriate level of abstraction for each part of the topology. As an example, a sysadmin can use the "SQL Database" node template for parts of the cloud application that are database agnostic and use the "MariaDB Database" node type, in the same model, for parts of the cloud application that are tied to that specific database.

A big advantage of topology-based cloud modeling languages is that each individual component is isolated. Interaction between components is only possible using relationships. This makes it very easy to create reusable components, resulting in community-driven capturing and reuse of configuration management code[15,16,17], which is very important in the field of cloud resource orchestration[7]. However, the actual reuse of knowledge is limited because of the issues explained in the introduction.

The following research efforts have made progress towards formalizing the concepts behind topology-based cloud modeling languages. Andrikopoulos et al. propose a set of formal definitions to reason on topology-based cloud applications with the goal

of ***selecting*** the optimal distribution[18]. The Aeolus component model makes it possible to formally describe several characteristics of a cloud topology such as dependencies, conflicts, non-functional requirements and internal component state[19]. Brogi et al. propose a petri net-based approach to formally model the relationships of TOSCA cloud models[20].

A big shortcoming of cloud modeling languages is the limited support for creating abstractions using cloud models. This has caused TOSCA orchestrators such as Cloudify to build their own methods to enable this. Cloudify's solution to this is Cloudify Plugins[21]. A Cloudify plugin basically allows adding additional orchestration logic into the Cloudify Orchestrator. These plugins allow to define new base node types and control how the orchestrator handles them. Although this method provides a way for Cloudify users to create new abstractions, it is still lacking. For instance, this method does not make it possible to stack abstractions: you cannot create new abstractions by combining and encapsulating a number of existing abstractions. Furthermore, this method nullifies an important property of TOSCA models: their portability. A TOSCA model that uses a plugin-specific node type cannot be interpreted by an orchestrator that does not support the specific plugin. Since plugins are developed using a Cloudify-specific API, this essentially re-introduces vendor lock-in into the TOSCA ecosystem. This is a big issue, given that TOSCA's main selling point is that it eliminates vendor lock-in. Note that this method of extending orchestrators to enable abstraction is neither limited to Cloudify nor to TOSCA. The alien4cloud project[22] provides an abstraction layer on top of TOSCA models, and the conjure-up project[23] provides an abstraction layer on top of the Juju[24] cloud modeling language discussed in Section 2.4.

2.3 | Models at runtime

Cloud modeling languages can also be used to ***monitor*** and ***control*** runtime state[25]. The *models at runtime* (M@RT) approach is to have a model that is causally connected to the running application: the runtime model is constantly updated to mirror the runtime state[26,27]. Another approach called *self-modeling*[28] is to dynamically generate a model from the current state using generic building blocks, which has been shown to be useful for self-diagnosis and root-cause analysis[29]. These approaches, however, are limited in that they only support a one to one mapping between the runtime model and the runtime state. This is an issue because complex abstractions can violate this constraint: a single abstraction can represent different runtime states in different topologies and different abstractions can represent the same runtime state in different topologies, as is the case with the "Spark on Hadoop" example used for evaluation in Section 4. Solving this issue requires more complex translations between the running application and the model than the M@RT approach currently provides.

2.4 | Agent-based management of cloud applications

The approach of converging towards a predefined end-state as popularized by Burgess et al.[3] is inherently inflexible as explained in the introduction. We believe that agent-based cloud management addresses this inflexibility. An agent-based cloud application manager consists of a number of independent agents that each manage a specific part of the cloud application i.e. a service. Each agent independently decides what the desired end-state is for that service and executes the necessary actions to get into that state. Dependencies between services are resolved by communication between the agents. As an example, a cloud application consisting of two services, a website and a database, is managed by two agents. One agent is responsible for the website and the other one for the database. If the website needs a connection to a running database, the agent responsible for the website will wait until it receives a message from the agent responsible for the database saying that the database is running. In this approach to config management, the global end-state of the cloud application emerges from the collective behavior of the agents.

One of the big advantages of an agent-based approach is the reliability and resiliency against failures as shown by Xavier et al.[30] and Kirschnick et al.[31]. Lavinal et al. have shown that the local autonomy of each agent combined with their organizing behavior enables global management autonomy in a distributed environment[32]. The flexibility of the agent-based approach allows it to manage not only analytical platforms but also the workloads running on top of those platforms as shown by the authors' previous work[33]. The state of the art in this area however does not address the need for abstraction, collaboration and reuse.

Juju[24] is a cloud modeling language and orchestrator that can be seen as a hybrid between agent-based management and cloud modeling languages. The sysadmin creates a Juju model describing the entire topology of the cloud application, the Juju orchestrator interprets that model, but the actual management of the individual services is done by agents co-located with the services. Juju makes it possible to encapsulate automation code that manages a single service in a charm, an entity similar to a TOSCA node type, but charms are not stackable and can only manage a single service.

2.5 | General limitations and lessons learned

A recent survey on cloud orchestration by Weerasiri et al.[7] identifies a number of general limitations across the field of cloud application management. Although concerns such as conformance to QoS and SLA requirements are reasonably addressed by the state of the art, the importance of knowledge reuse is underestimated and there is too much fragmentation resulting in sysadmins having to use different tools to manage different aspects of the application lifecycle[7]. The survey furthermore proposed the idea of *orchestration knowledge graphs*, where *"common low-level orchestration logic can be abstracted, incrementally curated and thereby reused by DevOps"*[7].

For this reason, the presented research's focus is not in finding new orchestration and scheduling techniques, but in developing a specification that makes it possible for the existing cloud schedulers, modeling languages and orchestrators to work together, enabling encapsulation of cloud application management knowledge in orchestration knowledge graphs.

There are a number of concepts in the state of the art that are very useful in achieving this. The presented research uses the following concepts as a foundation for the orchestrator conversation as explained in the next section.

a) The concept of a meta-scheduler as a way for different schedulers to work together solves real problems and should be applied to orchestration as well.

b) Topology-based cloud modeling languages make it easy to represent and reason over a cloud application and enable communities to collaborate around encapsulated knowledge.

c) Runtime models are a great way to represent the current state of a cloud application.

d) The agent-based approach to deployment of cloud applications provides a lot of benefits to the resiliency of the management system.

3 | ORCHESTRATOR CONVERSATION

We propose the orchestrator conversation as a fundamental new way to approach cloud modeling languages and orchestrators. This section gradually introduces the key concepts behind the orchestrator conversation. As a running example, the management of a Hadoop cluster is used. The section concludes with a summary of the entire conversation.

The need for knowledge reuse has heavily influenced the design decisions in this section. Specifically, the systematic mapping study of software reuse by Bombonatti et al. is used as a guideline for the non-functional requirements of the orchestrator conversation: *understandability*, *modularity* with *loose coupling*, *flexibility* and *abstractness*.

3.1 | Request and runtime models

Two pieces of information are key to the management of cloud applications: what state should the application be in, and what state is the application currently in. In the orchestrator conversation, that information is embedded in two types of cloud models.

Definition 1. *The **request model** is a structured description of the desired state of part of the cloud application.*

Definition 2. *The **runtime model** is a structured description of the actual state of part of the cloud application at a specific point in time.*

These models follow a predefined schema that is both human- and machine-readable so both system administrators and the management platform itself can *understand* them. Cloud modeling languages are perfectly fit as the schema for these models, since they provide a way to describe cloud applications satisfying both constraints. The request and response models are always linked in the sense that the runtime model describes the state in the context of the request model. It must be clear from the runtime model what the status is of the requests in the request model. As the example in Figure 1 shows, the runtime model uses the same names for the service and its properties so that the sysadmin understands the runtime model.

These two models are the primary way for a system administrator to communicate with the management platform. The sysadmin creates a request model to tell the management platform what the desired state of the cloud application is and uses the runtime model to keep track of the runtime state of the cloud application.

Capturing the complete and current state of a highly distributed cloud application requires strict consistency, which degrades performance immensely because of global locks. Furthermore, locking the state of a running cloud application without any

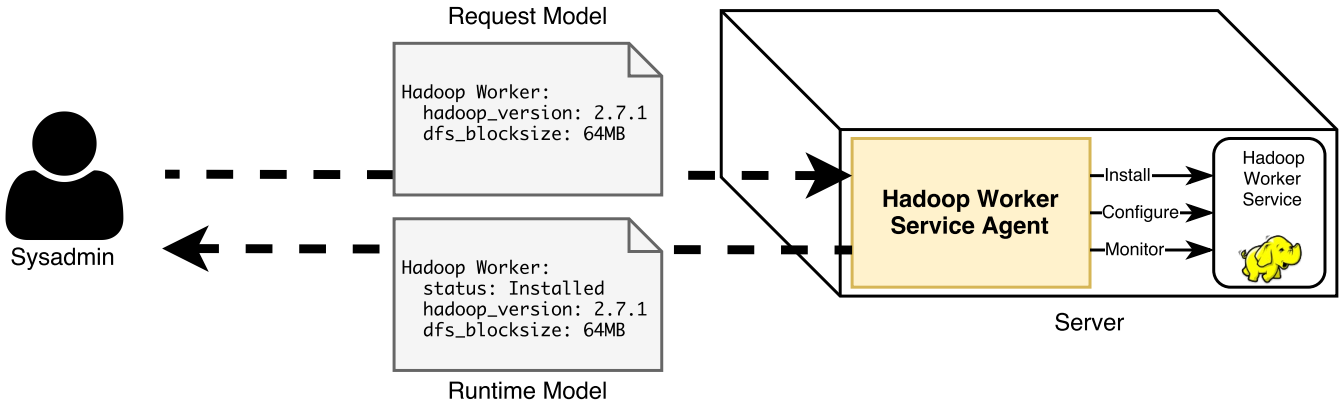

FIGURE 1 The Hadoop Worker service agent runs on the same machine as the Hadoop Worker service. The sysadmin defines the desired state of the service in the request model and the service agent notifies the sysadmin of the current state using the runtime model.

downtime is arguably impossible since state changes such as a running process crashing unexpectedly cannot be prevented. This is the reason why the runtime model does not represent the state of the cloud application in the present, but at some time in the past. The orchestrator conversation follows the eventual consistency model[34]: if the state does not change, the runtime model will eventually reflect the current state. This is achieved by ensuring the following three properties.

- The runtime model represents the state at some point in the past.
- If the state has changed since that point, the runtime model will be updated at some point in the future.
- Write conflicts are handled using the "last writer wins" approach.

3.2 | Service Agent

The actual management of the individual services is the responsibility of the service agent (SA).

Definition 3. *A **service agent** is an event-based program that manages a single service to get it into the state described in the request model and creates the runtime model that reflects the runtime state of the service.*

A service agent is fully event-based, thus it only reacts to changes, which can be internal such as a service crash, or external such as an update to the request model. Each change generates an event that the service agent needs to process. Each service agent has a specialized role. For example, Figure 1 shows a service agent that is specialized in managing a Hadoop Worker service. The service agent is running on the same machine as the service itself. This is an easy way to give the service agent access to the service, although this is not required as shown in the authors' previous work[35].

In the scenario shown in Figure 1, the process is the following.

1. The system administrator **selects which service agent to use.**
2. The sysadmin deploys the service agent onto a server.
3. The sysadmin sends the request model to the service agent and subscribes to its runtime model.
4. The service agent deploys and configures the service to get it into the state described in the request model.
5. The service agent updates the runtime model to reflect the current state and sends it to the sysadmin.

A key difference between the service agent approach and conventional configuration management tools is that the system administrator not only chooses the end state, but also the entity that will interpret that end state, i.e. the service agent. This

solves the issue explained in the introduction that the languages used to describe the end state do not support creating flexible *abstractions*. Service agents provide that functionality and can contain arbitrary processing logic to translate an abstract request model into a practical set of operation actions that need to happen. As a result, sysadmins can encapsulate knowledge in a service agent so it can be reused. Figure 1 shows an example where the request and response models only specify a few configuration options. The service agent decides what the optimal values are for the unspecified configuration options. As a result, *flexibility* and *understandability* are not mutually-exclusive in this approach. Aside from enabling knowledge reuse, the service agent also has a number of other advantages:

- The need for a one-size-fits-all cloud modeling language is removed because different languages can be used for management of different services. As an example, modeling a Virtual Network Function (VNF) is quite different from modeling a big data service so it is useful to model each in a modeling language specific to its domain.
- Service agents allow for fine-tuned translations from declarative request models into imperative steps instead of having to rely on generic translation rules provided by the management software.
- System administrators can reuse existing "legacy" models such as a TOSCA topology in the orchestrator conversation. This allows them to capitalize on their existing investments in model-driven management of cloud applications and it provides an easy migration path from conventional management tools.
- The approach enables a heterogeneous ecosystem where cloud modeling languages can compete with each other and evolve quickly.

3.3 | Collaborator relationship

The previous subsection focused on managing a single service. However, cloud applications generally consist of multiple interconnected services. A standard Hadoop setup for example requires three services working together: the Namenode, the ResourceManager and the Hadoop Worker. Connecting these services requires communication between the respective service agents in order to exchange information such as IP addresses and port numbers. The collaborator relationship enables this communication between service agents.

Definition 4. *A **collaborator relationship** is an isolated, two-way communication channel that connects two service agents and allows a **conversation** between them. Each service agent has a **role** in this conversation, which denotes how the service agent acts. Two types of conversations are possible: unary and binary. In a **unary** conversation, both service agents have the same role. In a **binary** conversation, each service agent has a different role. A collaborator relationship between two service agents is possible if either both implement the same role of a peer-to-peer conversation or if they both implement opposite roles of a directional conversation.*

The term *conversation* is chosen to differentiate from the simple static exchange of properties possible in languages such as TOSCA. Similar to its meaning in Business Process Model and Notation (BPMN), a conversation can consist of multiple interactions and messages resulting in negotiation and resolution of complex dependencies. The Hadoop cluster shown in Figure 2 shows three directional conversations. In each conversation, both connected service agents have a distinct role. For example, the Hadoop Namenode and the Hadoop ResourceManager have two distinct roles in their conversation simply because they provide information about two distinct services. The collaborator relationships in this example are used to check Hadoop version compatibility, exchange IP addresses and port numbers, setup shared credentials, and coordinate service starts and restarts. A sysadmin creates a relationship by specifying a request for the relationship in the request models of both service agents. Each relationship request contains the address of the other service agent, the roles of both service agents and the conversation protocol.

Note that a relationship only denotes that two agents can communicate. Whether or not that communication will be fruitful is not specified. As an example, during the conversation between the Hadoop Namenode and the Hadoop ResourceManager, it might become clear that the ResourceManager's Hadoop version is incompatible with the Namenode's Hadoop version. If the Hadoop version is specified in the request model, then the service agents are not allowed to change the Hadoop version themselves. It is then the responsibility of both service agents to explain this issue in their runtime models so that the sysadmin can intervene, e.g. by changing the ResourceManager's Hadoop version.

The collaborator relationship is a key piece to enabling collaboration because it allows multiple parties to develop service agents independently, while still allowing these service agents to collaborate and communicate. A developer can implement

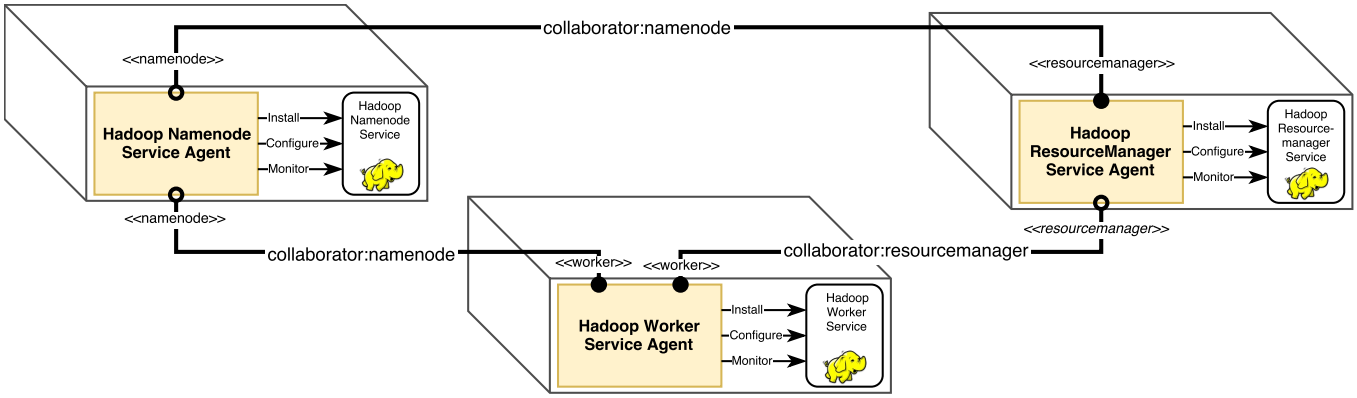

FIGURE 2 Illustrative example: the collaborator relationships between SA's of a Hadoop cluster.

the collaborator conversation without knowing the implementation details of the service agent on the other side because the conversation acts as a generic protocol that hides the implementation. As a result, ISVs can create service agents that manage their own software and connect to software from other vendors. Orchestrator vendors facilitate this collaboration by creating a set of standardized collaborator conversation protocols that ISVs can program against. This will result in a rich ecosystem of interoperable service agents at the fingertips of system administrators who use them as building blocks for their cloud application.

3.4 | Controller

Deploying multiple service agents introduces a new issue: the system administrator needs to create a request model for each service agent, which would be very cumbersome to do manually. This is where the topology-based cloud modeling languages come in. Such languages provide a way to describe a cloud application as a set of interconnected services. They are thus ideal for creating the combined request model including the relationship request between the different service agents. Dividing the topology model into a number of request models is quite straightforward: each vertex is a separate request model. Each edge is described in both request models of the nodes it connects. Doing this division is the job of the *controller*. The controller receives the entire request topology model, divides the request model and sends each part to the service agent responsible for that service.

3.5 | Operator relationship

The service agent as described in the previous sections enables abstraction of one single service, but it does not allow abstracting an entire topology into a single component. As an example, when setting up the Hadoop cluster, the sysadmin still needs to know that a Hadoop cluster consists of a Namenode, Datanode and a Worker, and how they need to be connected. As explained in the introduction, there is a need for an abstraction layer that can represent a cluster of services as one service, so the system administrator can request a deployment of "a Hadoop cluster" and have the management platform figure out what services are required for a Hadoop cluster. As shown in the state of the art section, it is important that this abstraction is two-way: both the request and response model the sysadmin interacts with need to represent the individual Hadoop services as one Hadoop cluster.

Figure 3 shows the orchestrator conversation's solution to this: a service agent that takes over the job of the system administrator. This service agent receives the abstract request for the Hadoop cluster and fulfills that request by creating new service agents, sending them request models, listening for runtime models, translating those runtime models into one overall state of the Hadoop cluster, and sending a runtime model reflecting that state to the sysadmin. The key to enabling this is to characterize the interaction between the sysadmin and a service agent and create the operator relationship that allows such interaction between service agents themselves.

Definition 5. *An **operator relationship** is an isolated, two-way communication channel between two service agents that allows one service agent to send request models to and receive response models from the other service agent. The conversation going over the operator relationship has two roles: the **operator** sends request models to drive the behavior of the **executor**, which sends runtime models back to the operator to inform the operator about the runtime state. A service agent can be the executor in only one operator relationship, but it can be the operator in multiple relationships.*

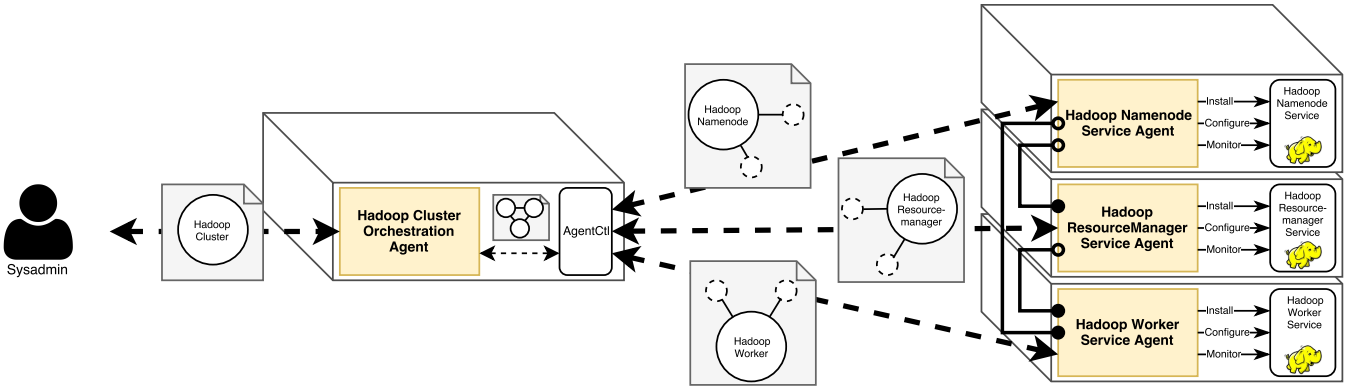

FIGURE 3 The Hadoop cluster orchestration agent manages multiple service agents by sending them individual request models.

This article refers to a service agent that manages a number of other service agents using an operator relationship as an "**orchestration agent (OA)**". The interaction between a system administrator and an OA can be seen as the system administrator having an operator relationship to the service agent. In fact, there is no practical difference between a system administrator managing a service agent or another service agent managing that service agent. As a result, it is possible to create multiple layers of abstraction by chaining service agents using operator relationships.

This does have implications on the visibility of the cluster of service agents to the system administrator. Only the service agents that are directly connected to the sysadmin are fully visible, which is wanted behavior since it enables abstraction: the visibility of the service agents in lower abstraction layers is curated by the service agents in the upper abstraction layers. Subsequently, each service agent has complete control over its executors.

The operator relationship is the second key piece for enabling a rich ecosystem of orchestration knowledge. Just as the collaborator relationship, the operator relationship serves as a generic protocol that service agent developers can program against. ISVs can now create service agents that manage entire clusters of their software and orchestrator vendors can create service agents that translate higher-level requests to lower-level setups.

3.6 | Summary

The orchestrator conversation consists of a hierarchical collection of *service agents* that collaborate to deploy a cloud application. Each service agent is an orchestrator specialized in managing a specific part of the cloud application. A sysadmin deploys service agents and sends them *request models* to specify what the desired end-state is. These service agents then communicate using *collaborator relationships* to connect different parts of the cloud application and delegate work by deploying new service agents and managing them using the *operator relationship*. The service agents report back to the sysadmin using eventually-consistent *runtime models* that show the state of the cloud application in the context of the request model.

Service agents encapsulate orchestration logic and expose their functionality over abstract APIs, thus enabling knowledge reuse. Relationships between service agents use agreed-upon conversation protocols, enabling a vibrant ecosystem of multiple vendors creating interoperable service agents. Orchestration vendors can ease this collaboration by standardizing conversation protocols and providing generic orchestration agents that perform tasks such as auto-scaling. The end-result is a *modular* system with *loosely coupled* components.

The orchestrator conversation can be seen as a swarm in the sense that it consists of a number of locally interacting SA's that collectively achieve the goal of managing the cloud application without a centralized control structure. An SA's interactions are strictly local because communication can only happen over relationships, the local neighborhood of a SA is determined by its relations and the management of the cloud application is an emergent behavior since no single SA has the knowledge to manage the entire cloud application.

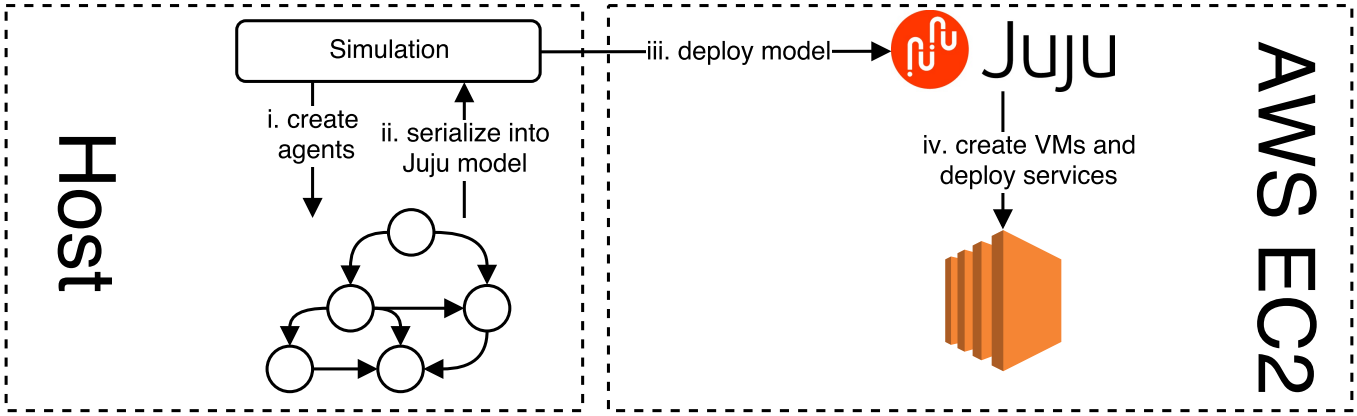

FIGURE 4 The evaluation setup. The orchestrator conversation is simulated on the host. The resulting topology is serialized into a Juju model which is deployed on AWS EC2.

3.7 | Aside: declarative and imperative modeling

The topic of declarative versus imperative modeling is subject of an ongoing debate in the field of cloud modeling and orchestration[36 37 38]. This also introduces the issue of uncertainty since different orchestrators can interpret a model in different ways, causing the orchestration to fail in unexpected ways. For this reason, ISVs have a strong preference for imperative models.

ISVs see cloud modeling languages as a way to enhance the experience of their customers by accompanying their software with a model that encapsulates the knowledge of how to deploy and manage the individual software components. Imperative models allow them to fine-tune the models and have more control over the quality of experience of their customer, regardless of what orchestrator the customer uses.

The abstraction capabilities introduced by the orchestrator conversation are a great way to have the best of both worlds. The ISVs create the orchestration agents in an imperative way so they have full control over what the orchestration actions are and the customer interacts with the orchestration agent using declarative request models. The ISVs have full control over the orchestration actions and in turn over the quality of experience of their customers, while the complexity is still hidden to the user.

4 | EVALUATION

4.1 | Evaluation Setup

To evaluate the proposed orchestrator conversation, a number of proof-of-concept orchestration agents and simulated service agents are created using Python and the "Pykka" actor model framework. The full source code is available on Github[39]. The service agents are simulated in the sense that they do not actually deploy and manage services, they only track what state these services need to be in. When the orchestrator conversation finishes, the agents recursively serialize into a Juju bundle that describes the desired cloud application state. This bundle is then used to actually deploy the cloud application as shown in Figure 4.

All benchmarks of the orchestrator conversation were run on an Ubuntu 17.10 host with a 4-core Intel i5-7440HQ CPU and 16 GB of RAM. Deployment benchmarks were run using Juju 2.2.5 on AWS EC2 virtual machines of type " m3.medium" running Ubuntu 16.04 with 1 vcpu and 3.840 GB of RAM. The numbers of runs of the experiments and simulations are chosen so that there is sufficient convergence in the results and the standard deviation is small enough to show the significance of relevant trends.

4.2 | Complexity towards the sysadmin

The first evaluation checks whether the orchestrator conversation makes it possible to hide the complexity of managing a cloud application to the sysadmin. The primary way of interfacing with a model-based cloud management platform is the model itself. Thus, the complexity of the model that the sysadmin interfaces with, is a good approximation of the complexity of using that management platform.

We compare the request model for a Hadoop cluster in the orchestrator conversation simulation with three models from the following state of the art projects: DICE, INDIGO DataCloud and Apache Bigtop. The exact models used are available on Github[39].

- DICE is a European Horizon 2020 project aimed at defining a framework for quality-driven development of Big Data applications[40], which leverages TOSCA models to deploy and manage big data applications. This evaluation uses their example Hadoop models[41].
- The INDIGO - DataCloud project develops an open source data and computing platform targeted at scientific communities[42]. TOSCA is used as the method to interface between the INDIGO platform and end users. This evaluation uses the example model to request a Hadoop cluster from the INDIGO platform[43].
- Apache Bigtop is an Apache Foundation project that is the de-facto standard for packaging, deployment and testing tools for open-source big data frameworks[44]. It is the upstream of many commercial big data offerings such as Cloudera's CDH Hadoop distribution. This evaluation uses the Bigtop reference bundle for deploying Apache Hadoop using Ubuntu Juju[45 46].

This evaluation uses four indicators of the complexity of a model. Each object is a variable that needs to be specified by the user, thus increasing the complexity of the model. Moreover, each individual object acts as a multiplier to the overall complexity because the values of different objects need to be correct in combination. The number of objects can thus be seen as the degrees of freedom of the model. The indicators are as follows.

1. The number of **nodes** in the topology model. This is the number of declared node types for a TOSCA model and the number of declared applications and machines for a Juju model. Node types referenced but not defined are not counted since defining these node types is the role of the platform and thus provides no additional complexity for the sysadmin.
2. The number of **relationships** in the topology model. Each relationship is only counted once, even if it is declared at both endpoints such as in certain TOSCA models. Machine placement directives in Juju models are also counted as relationships since these signify a relationship between the application and the machine.
3. The number of **outputs**. In TOSCA models, the request model defines what runtime properties the sysadmin is interested in, and how these runtime properties should be formatted. Juju request models do not contain outputs because it is up to the Charm to decide what information should be shown to the sysadmin and how it should be formatted.
4. Number of **properties** present in the model. This maps to the number of properties in the TOSCA models, and the number of configuration values, constraints, and scale declarations in the Juju models.

Figure 5 compares the complexity of the four evaluated models and Figure 6 shows the topologies of these models as a graph. The indicators diverge a lot between the different state-of-the-art models because each model makes a different trade-off between flexibility and complexity: more information exposed in the model means greater flexibility but also more complexity. This trade-off is inherent to the state-of-the-art, since it is not possible to have both, as explained in the introduction. The trade-offs of the presented models are further investigated below.

- The DICE model as a large number of nodes and relationships because the IP, firewall, and virtual machine are also modeled as separate nodes with relationships. Although this is TOSCA-compliant and improves the reusability of the nodes, it greatly increases the complexity of the model.
- In the INDIGO model, the Hadoop services are only represented by two services: "hadoop_master" and "hadoop_slave" instead of the four separate services "Namenode", "Datanode", "NodeManager" and "ResourceManager". This causes the INDIGO model to be less complex, but it hampers the flexibility and reusability of the components in the model.

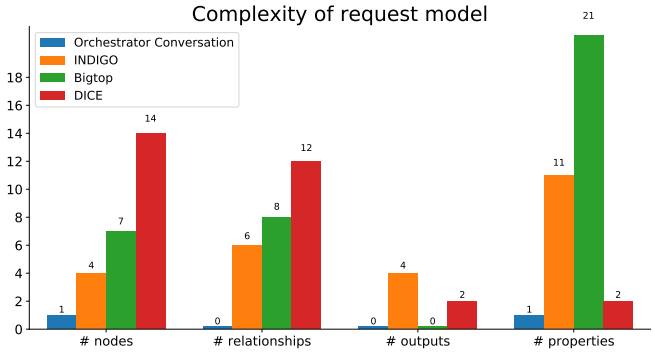

FIGURE 5 The request model for the orchestrator conversation only contains one node, the Hadoop cluster node, and one property, the scale of the Hadoop cluster.

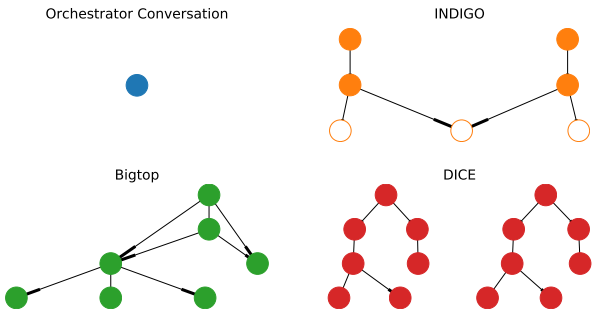

FIGURE 6 The request models of a Hadoop cluster in the different formats. The unfilled circles in the INDIGO model represent node types that are referenced in relations but not defined in the model itself.

- The Bigtop and INDIGO models contains a large number of properties because the requirements of the host machines such as CPU, RAM and root disk space are part of the model. These properties are not set in the other models, which will result in the orchestrator deciding these values, causing the user to have no control over this..

The request model for the orchestrator conversation scores the best on every indicator which proves that this approach indeed makes it possible to hide the complexity of managing a Hadoop cluster. On the other hand, the sysadmin can still choose to manually model the Hadoop cluster out of individual components, or use a mix of components with different abstraction levels, should that need arise. Thus, the trade-offs between complexity and flexibility are not needed in the model of the orchestrator conversation.

Note that, as explained in Section 2, it is technically possible to make the TOSCA models easier using Cloudify plugins or node templates, but these methods have great limitations: node templates do not exist after deployment of the model and Cloudify plugins are neither standards-based nor stackable. In contrast, in the orchestrator conversation, the sysadmin can still manage the cloud application at runtime using the "Hadoop Cluster" abstraction, and the abstraction is completely stackable, the underlying individual service agents and their request models are still present.

4.3 | Overhead of the orchestrator conversation

Some studies report that the reusability of software has a negative impact on its performance due to the overhead of abstraction and the absence of context-specific optimizations[4]. In order to see whether this is true for the orchestrator conversation, this series of evaluations investigates the overhead of turning an abstract request model into a full topology that satisfies it.

Since the main interest is in the overhead of the abstraction itself, the orchestrator conversation is only used to figure out what needs to be deployed in order to satisfy the initial request model. The orchestration agents create, configure and connect the required service agents but these don't actually deploy or manage any services, they simply figure out the desired state of the service and immediately report in their runtime model that the request model has been satisfied, which propagates through the topology until the initial request model is completely satisfied. At this point, the simulation stops and the desired state of all services is serialized into a Juju topology model. This model is then used to confirm if the required state as described by the service agents is correct, and deployed using Juju to get the time required to deploy the cluster using a state-of-the-art orchestrator.

The initial request model for this simulation is shown in Figure 7 as the "initial topology": a Hadoop orchestration agent connected to a Spark orchestration agent. The goal is to create the complete topology of orchestration and service agents which satisfies the request model. The request model of each agent contains a requested number of workers, referred to as k for Hadoop and n for Spark. During the simulation, the orchestration agents create new orchestration agents, service agents and relationships in order to achieve the desired state of the request model. The simulation finishes when all orchestration agents notify the user that the request model has been satisfied, thus when the complete topology is created. During the simulation, the two orchestration

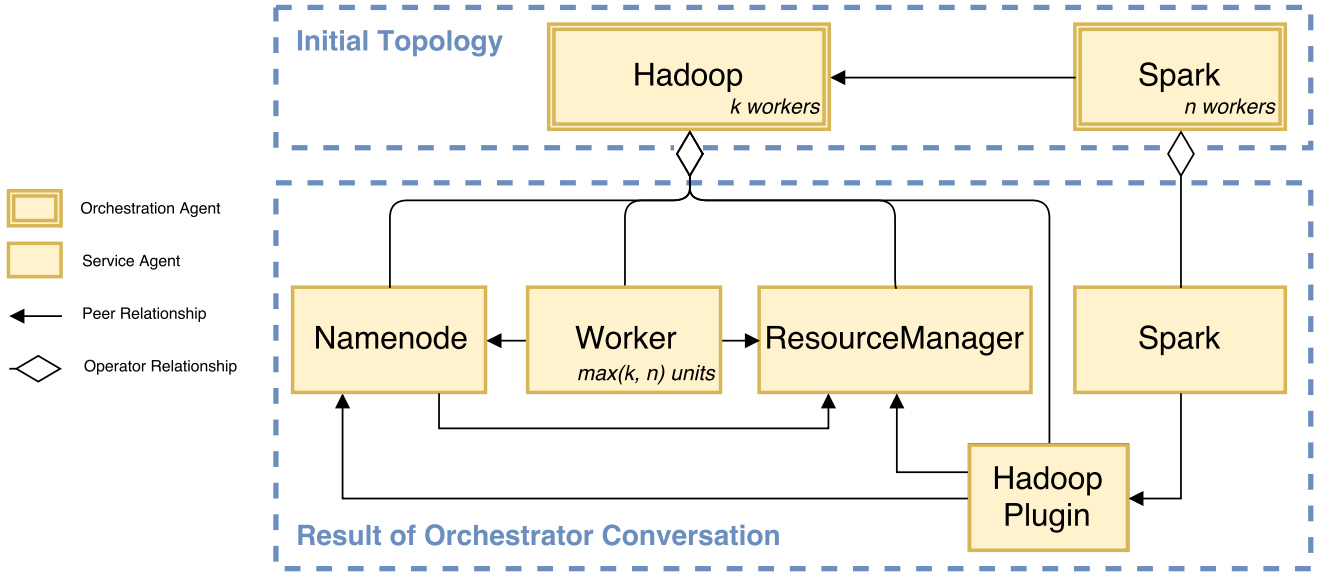

FIGURE 7 The simulation starts with a Hadoop OA and a Spark OA connected to each other. The goal is to create a Spark cluster running on a Hadoop cluster that consists of a Namenode, a ResourceManager and a Worker.

agents use the relationship to figure out what other agents need to be created and what the desired state is to fulfill the request. What follows is a summary of the actions and decisions that have to be made by the orchestration agents in this simulation.

1. The Hadoop OA creates service agents that deploy and manage a Hadoop cluster.
2. The Hadoop OA also creates the Hadoop plugin service agent. This plugin SA provides its peers with the correct information to connect an application to Hadoop. The Hadoop OA then sends the address of the plugin SA to the Spark OA so that the Spark OA can create a relationship between the plugin SA and the Spark SA.
3. The Spark OA creates a single Spark Client service agent and connects it to the plugin SA. Since Spark has to run on Hadoop, there is no need to create a full Spark cluster.
4. The Spark OA requests the Hadoop OA to create n workers since each Hadoop worker functions as a Spark worker when Spark runs on Hadoop.
5. The Hadoop OA compares n to k, the amount of workers from its request model, and updates the request model of the worker SA to create $max(n,k)$ workers.

This interaction clearly shows the strength and flexibility of the orchestrator conversation. The abstraction of the Spark OA cannot be provided by a TOSCA node template because there is no one-to-one translation from the abstract "Spark cluster"node to what will actually be deployed: if the Spark OA is connected to a Hadoop OA, it will only deploy a single Spark client, otherwise it will deploy a full Spark standalone cluster. Similarly, the conventional M@RT approach will not work here because there is no one-to-one mapping between what the Spark OA reports as "number of Spark workers" and what is deployed: when it is connected to the Hadoop OA, the Spark OA will report the number of Hadoop workers, otherwise it will report the scale of the Spark standalone cluster. After the simulation finishes, all service agents serialize the desired state of the service into a Juju bundle, which is deployed to get the deployment time of the cloud application.

Figure 8a shows the time required by the orchestrator conversation to create the complete topology, starting from the request model. Each dot represents the aggregated result of 10 simulations of the orchestrator conversation. Figure 8b shows the deployment time: the time required by Juju to deploy the topology model created by the orchestrator conversation simulation. Each dot represents the aggregated results of two deployments. While the simulation time stays under 100ms, the deployment time ranges from around 20 minutes to over one hour. This demonstrates that the overhead of using orchestration agents as an abstraction is negligible compared to the deployment time of the actual cloud application.

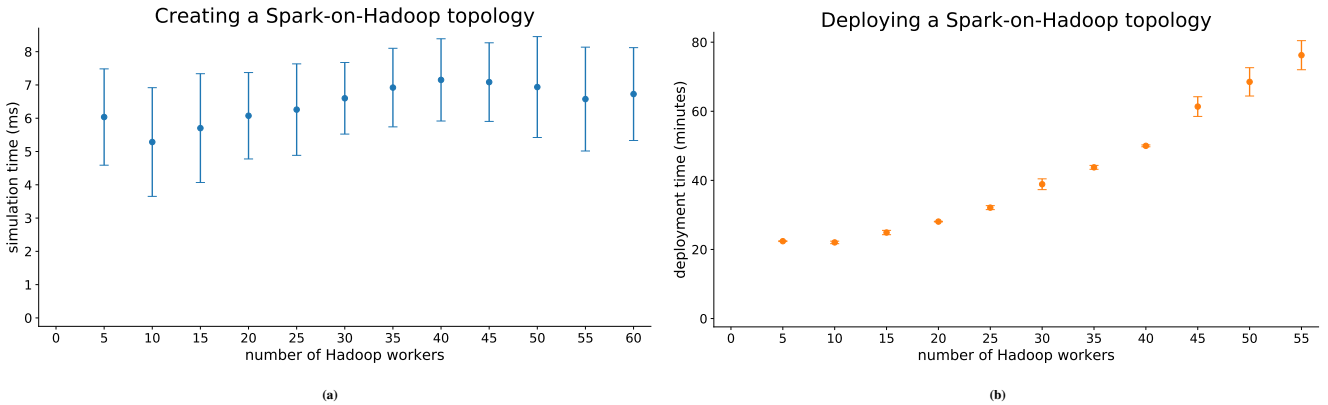

FIGURE 8 The overhead of the orchestrator conversation in (a) is less than 0.10 seconds which is insignificant compared to a minimum deployment time of around 20 minutes shown in (b). The simulation time remains constant because the amount of workers is represented in the Hadoop Worker service agent as an integer value.

4.4 | Scalability of the orchestrator conversation

This series of evaluations focuses on how the orchestrator conversation scales when creating the topologies for multiple different cloud applications at the same time. It is a common scenario that cloud infrastructure is shared over multiple teams, projects, and environments such as development, staging and production.

These evaluations simulate such a scenario by deploying multiple Spark-on-Hadoop clusters simultaneously. It does not matter what the actual orchestrated clusters are since these evaluations focus on the scalability of the orchestrator conversation and not the scalability of the orchestration actions themselves.

Unlike scaling a single cluster, creating multiple clusters actually creates multiple orchestration agents. Figure 9 shows that the simulation time scales linearly in function of the number of clusters. This linearity is expected since the clusters don't have any dependencies on each other. Consequently, if all the clusters are equal, the time to create the topology for cluster $n+1$ is equal to that for cluster n, thus each cluster adds a constant overhead to the overall simulation time.

4.5 | Concurrency in the orchestrator conversation

Concurrency is a strong point of a distributed service orchestrator because the orchestrator conversation allows all orchestration agents to run at the same time. This is inherent to the orchestrator conversation and does not require code changes in the orchestration agents. This stands in stark contrast with the monolithic nature of current state of the art orchestrators. A single-process orchestrator cannot orchestrate a topology concurrently, and enabling parallelization in a monolithic orchestrator creates a complex system that is hard to adapt and fine-tune[9]. This series of evaluations has the goal to find out if the orchestrator conversation actually uses the concurrency potential of the topology.

These evaluations use a simple orchestration agent that creates a number of children and waits until these children are ready. This process continues recursively until the requested amount of orchestration agents is created. This creates a tree of orchestration agents connected by the operator relationship. The leaves of the tree immediately go into the ready state as soon as they are created. This causes the ready states to propagate through the tree from the leaves to the root. When the root orchestration agent is in the ready state, the simulation stops and the duration of the simulation is used for evaluation. The duration of two topologies with the same number of nodes but different concurrency potential is compared.

Changing the number of children of an orchestration agent has a big impact on the concurrency potential of the topology. This simulation uses a unary tree of the aforementioned orchestration agents as a topology without concurrency potential. Both the creation of the unary tree and the propagation of the ready state has to happen sequentially, one node after the other. A binary tree is used as an example of a topology that has a lot of potential for concurrency.

The big performance increase when concurrency is possible, as depicted in Figure 10, shows that the distributed service orchestrator does indeed use the concurrency potential of the topology. This has a big impact on the real-world performance of a distributed service orchestrator because the main use of the operator relationship is to abstract; to connect one operator to multiple executors, allowing the executors to run concurrently. Note that in a real-world topology, the number of children can

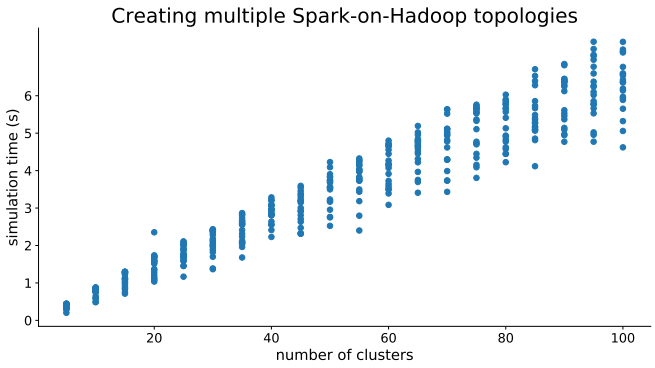

FIGURE 9 The simulation time scales linearly as a function of the number of unconnected clusters of orchestration agents. This graph shows the result of 10 runs for each x value.

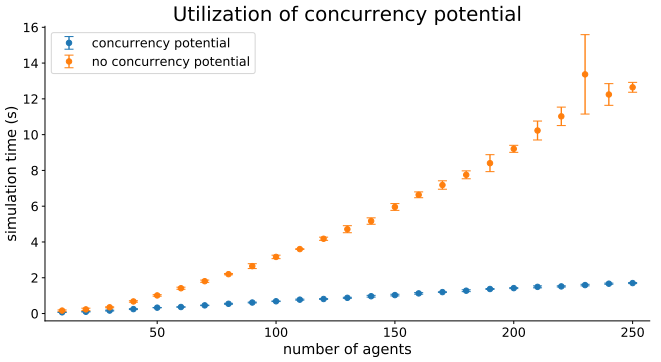

FIGURE 10 The orchestrator conversation happens as concurrently as the topology allows, without any code change required, which significantly improves the scalability. Each dot represents the aggregated results of 5 simulation runs.

differ between orchestration agents and can exceed two, as is the case with the Hadoop OA in the Spark-on-Hadoop cluster, which has 4 children. A topology where each node has 3 or more children contains even more concurrency potential, but our simulations have shown no significant performance improvements between a binary and tertiary tree because at that point, the number of cpu cores is the bottleneck, not the number of concurrent threads.

The optimal usage of concurrency potential also explains the scalability of orchestrating multiple unconnected clusters as discussed in Subsection 4.4: the clusters are not dependent on each other, thus the orchestration of all clusters runs concurrently.

5 | CONCLUSION AND FUTURE WORK

This article proposes the orchestrator conversation as a way to introduce abstraction of the cloud application to topology-based cloud modeling languages. The focus on the conversation instead of the orchestrator itself makes it possible for software vendors to create components that translate abstract, declarative models into management actions on their software. The evaluation shows that the orchestrator conversation can be used to create much smarter abstractions than possible with the state of the art, the overhead of the conversation is minimal compared to the actual time to deploy the cloud application, and that the request model is indeed less complex while the underlying full topology is still present. The evaluation also shows that the decentralized nature of the conversation enables the management of the cloud application to happen inherently concurrent. This greatly enhances the scalability of the solution and alleviates the need for the complex and error-prone process of manually programming concurrency into an orchestrator.

Hierarchical abstraction layers are possible by creating a tree of “operator" relationships. System administrators specify their request in the same way as orchestration agents: using the operator relationship, which means that orchestration agents themselves can fully utilize the abstractions of other orchestration agents in order to reason about and manage the cloud application on a higher level. This also makes it possible for system administrators to gradually automate more and more management tasks by creating orchestration agents that use the operator relationship to drive the behavior of other agents. It is important to note that this approach of a conversation instead of an orchestrator does not take orchestrator vendors out of the picture. Markets can still emerge around creating and selling orchestration agents where orchestrator vendors can use their expertise to create auto-scalers or specialized orchestration agents that can take SLA and QoS requirements into account.

This article treats OAs and SAs as a black box, however it is valuable for future work to see how QoS and SLA requirements can be reasoned about in such a distributed manner. The state machine approach currently used by cloud modeling languages to model the lifecycle of a single component is not flexible enough for agents with multiple independent sub-states. Future work will investigate more flexible ways to model SAs and OAs and investigate the advantages of the orchestrator conversation when used as an alternative rather than a supplement to existing cloud resource orchestration frameworks. An interesting topic to further investigate is whether this decentralized nature has a positive impact on the solution’s ability to cope with network

segmentation and its resiliency. Another important question is whether the hierarchical nature of the operator relationship has an adverse effect on the solution's ability to self-heal.

ACKNOWLEDGMENTS

This research was performed partially within the FWO project "Service-oriented management of a visualized future internet".

References

1. Dave Russel , Mike Chuba . Top Challenges Facing I&O Leaders in 2017 and What to Do About Them. Tech. Rep. G00324370Gartner 2017.

2. Rinnen P, McArthur J, Zaffos S, et al. 2017 Strategic Roadmap for Storage. Research Note G00324339 2017. Published by ClearSky Data, Inc.

3. Burgess M, College O. Cfengine: a site configuration engine. In: *in USENIX Computing systems, Vol* 1995.

4. Bombonatti D, Goulão M, Moreira A. Synergies and tradeoffs in software reuse – a systematic mapping study. *Software: Practice and Experience* 2016;47(7):943–957. doi:10.1002/spe.2416

5. Committee TT. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) Technical Committee | Charter. 2013.

6. Palma D, Rutkowski M, Spatzier T. TOSCA Simple Profile in YAML Version 1.0. 2015. OASIS Committee Specification Draft 04 / Public Review Draft 01.

7. Weerasiri D, Barukh MC, Benatallah B, Sheng QZ, Ranjan R. A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Comput. Surv.* 2017;50(2):26:1–26:41. doi:10.1145/3054177

8. Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J. Large-scale Cluster Management at Google with Borg. In: *Proceedings of the Tenth European Conference on Computer Systems*EuroSys '15(New York, NY, USA):18:1–18:17ACM 2015

9. Schwarzkopf M, Konwinski A, Abd-El-Malek M, Wilkes J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In: *Proceedings of the 8th ACM European Conference on Computer Systems*EuroSys '13(New York, NY, USA):351–364ACM 2013

10. Hindman B, Konwinski A, Zaharia M, et al. 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(Berkeley, CA, USA):295–308USENIX Association 2011.

11. Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J. Borg, Omega, and Kubernetes. *ACM Queue* 2016;14(1):70–93. doi:10.1145/2898442.2898444

12. Fazio M, Celesti A, Ranjan R, Liu C, Chen L, Villari M. Open Issues in Scheduling Microservices in the Cloud. *IEEE Cloud Computing* 2016;3(5):81–88. doi:10.1109/MCC.2016.112

13. Brogi A, Soldani J. Finding available services in TOSCA-compliant clouds. *Science of Computer Programming* 2016;115:177–198. doi:10.1016/j.scico.2015.09.004

14. Brogi A, Cifariello P, Soldani J. DrACO: Discovering available cloud offerings. *Computer Science - Research and Development* 2017;32(3-4):269–279. doi:10.1007/s00450-016-0332-5

15. Tsakalozos K, Johns C, Monroe K, VanderGiessen P, Mcleod A, Rosales A. Open big data infrastructures to everyone. In: *2016 IEEE International Conference on Big Data (Big Data)*:2127–2129 2016

16. Endres C, Breitenbücher U, Leymann F, Wettinger J. Anything to Topology – A Method and System Architecture to Topologize Technology-Specific Application Deployment Artifacts. In: *{Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017), Porto, Portugal*(Porto, Portugal)SCITEPRESS 2017.

17. Wettinger J, Breitenbücher U, Falkenthal M, Leymann F. Collaborative gathering and continuous delivery of DevOps solutions through repositories. *Computer Science - Research and Development* 2017;32(3-4):281–290. doi:10.1007/s00450-016-0338-z

18. Andrikopoulos V, Sáez SG, Leymann F, Wettinger J. Optimal Distribution of Applications in the Cloud. In: *Advanced Information Systems Engineering*:75–90Springer, Cham 2014

19. Di Cosmo R, Mauro J, Zacchiroli S, Zavattaro G. Aeolus: A component model for the cloud. *Information and Computation* 2014;239:100–121. doi:10.1016/j.ic.2014.11.002

20. Brogi A, Canciani A, Soldani J, Wang P. A Petri Net-Based Approach to Model and Analyze the Management of Cloud Applications. In: *Transactions on Petri Nets and Other Models of Concurrency XI*Lecture Notes in Computer ScienceSpringer, Berlin, Heidelberg 2016:28–48. DOI: 10.1007/978-3-662-53401-4_2.

21. Cloudify Plugins documentation. https://docs.cloudify.co/4.2.0/plugins/overview/. Accessed: Februari 8, 2018.

22. ALIEN 4 Cloud https://alien4cloud.github.io/. Accessed October 1, 2017.

23. conjure-up: Get started with big software, fast https://conjure-up.io/. Accessed October 1, 2017.

24. Ubuntu Juju: Operate big software at scale on any cloud https://jujucharms.com/. Accessed October 3, 2017.

25. Blair G, Bencomo N, France RB. Models@ run.time. *Computer* 2009;42(10):22–27. doi:10.1109/MC.2009.326

26. Shao J, Wei H, Wang Q, Mei H. A Runtime Model Based Monitoring Approach for Cloud. :313–320IEEE 2010

27. Seybold D, Domaschka J, Rossini A, Hauser CB, Griesinger F, Tsitsipas A. Experiences of Models@Run-time with EMF and CDO. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*SLE 2016(New York, NY, USA):46–56ACM 2016

28. Hounkonnou C, Fabre E. Empowering self-diagnosis with self-modeling. In: *2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm)*:364–370 2012.

29. Sánchez Vílchez JM, Ben Yahia IG, Lac C, Crespi N. Self-modeling based diagnosis of network services over programmable networks. *International Journal of Network Management* 2017;27(2):n/a–n/a. doi:10.1002/nem.1964

30. Etchevers X, Salaün G, Boyer F, Coupaye T, De Palma N. Reliable self-deployment of distributed cloud applications. *Software: Practice and Experience* 2017;47(1):3–20. doi:10.1002/spe.2400

31. Kirschnick J, Alcaraz Calero JM, Goldsack P, et al. Towards an Architecture for Deploying Elastic Services in the Cloud. *Softw. Pract. Exper.* 2012;42(4):395–408. doi:10.1002/spe.1090

32. Lavinal E, Desprats T, Raynaud Y. A multi-agent self-adaptative management framework. *International Journal of Network Management* 2009;19(3):217–235. doi:10.1002/nem.699

33. Sebrechts M, Borny S, Vanhove T, et al. Model-driven deployment and management of workflows on analytics frameworks. In: *2016 IEEE International Conference on Big Data (Big Data)*:2819–2826 2016

34. Vogels W. Eventually Consistent. *Commun. ACM* 2009;52(1):40–44. doi:10.1145/1435417.1435432

35. Sebrechts M, Vanhove T, Van Seghbroeck G, Wauters T, Volckaert B, De Turck F. Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration. In: *2016 IEEE International Conference on Mobile Services (MS)*:156–159 2016

36. Breitenbucher U, Binz T, Képes K, Kopp O, Leymann F, Wettinger J. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*:87–96IEEE 2014.

37. Lauwers C. Declarative vs. Imperative Orchestrator Architectures. 2017. http://blog.ubicity.com/2017/06/declarative-vs-imperative-orchestrator.html. Accessed August 18, 2017.

38. Endres C, Breitenbücher U, Falkenthal M, Kopp O, Leymann F, Wettinger J. Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In: *Proceedings of the 9'th International Conference on Pervasive Patterns and Applications*Xpert Publishing Services (XPS) 2017.

39. Orchestration Agents Code on Github. 2017. https://github.com/IBCNServices/oa. Accessed: Februari 8, 2018.

40. Casale G, Ardagna D, Artac M, et al. DICE: Quality-driven Development of Data-intensive Cloud Applications. In: *Proceedings of the Seventh International Workshop on Modeling in Software Engineering*MiSE '15(Piscataway, NJ, USA):78–83IEEE Press 2015.

41. DICE-Deployment-Examples: Example blueprints used with the DICE Deployment Service. 2016. https://github.com/dice-project/DICE-Deployment-Examples. Accessed October 2017.

42. Salomoni D, Campos I, Gaido L, et al. INDIGO-Datacloud: foundations and architectural description of a Platform as a Service oriented to scientific computing. *arXiv:1603.09536 [cs]* 2016. arXiv: 1603.09536.

43. tosca-types: YAML description of new types added to extend TOSCA Simple Profile in YAML Version 1.0. 2017. https://github.com/indigo-dc/tosca-types. Accessed October 1, 2017.

44. Anthony B, Boudnik K, Adams C, Shao B, Lee C, Sasaki K. Ecosystem at Large: Hadoop with Apache Bigtop. In: *Professional Hadoop*®John Wiley & Sons, Inc 2016:141–160. DOI: 10.1002/9781119281320.ch7.

45. Apache Bigtop and Juju: A Charming Approach to Big Data http://bigdata.juju.solutions//2016-06-07-apache-bigtop-and-juju/. Accessed October 1, 2017.

46. Apache Bigtop Github project. 2017. https://github.com/apache/bigtop. Accessed October 1, 2017.