Cloud-native Deploy-ability: An Analysis of Required Features of Deployment Technologies to Deploy Arbitrary Cloud-native Applications

Michael Wurster[1], Uwe Breitenbücher[1], Antonio Brogi[2], Frank Leymann[1] and Jacopo Soldani[2]
[1] *Institute of Architecture of Application Systems, University of Stuttgart, Germany*
[2] *Department of Computer Science, University of Pisa, Pisa, Italy*
[lastname]@iaas.uni-stuttgart.de, [lastname]@di.unipi.it

Keywords: Cloud-native Application, Automation, Deployment Technology.

Abstract: The adoption of cloud computing combined with DevOps enables companies to react to new market requirements more rapidly and fosters the use of automation technologies. This influences the way software solutions are built, which is why the concept of *cloud-native applications* has emerged over the last few years to build highly scalable applications, and to automatically deploy and run them in modern cloud environments. However, there is currently no reference work clearly stating the features that a deployment technology must offer to support the deployment of arbitrary cloud-native applications. In this paper, we derive three essential features for deployment technologies based on the current cloud-native research and characteristics discussed therein. The presented features can be used to compare and categorize existing deployment technologies, and they are intended to constitute a first step towards a comprehensive framework to assess deployment technologies.

1 INTRODUCTION

The wide adoption of cloud computing resulted in a change in how software solutions are developed and deployed. Market demands push software companies to adopt new practices to increase the ability of releasing more often, even immediately if required (Humble and Farley, 2010). To support this, DevOps emerged as a widespread concept to establish a high level of automation for accomplishing and managing software releases in shorter cycles. For rapid deployment cycles, the application components themselves need to feature certain characteristics, which resulted in the notion of *cloud-native applications* (Kratzke and Quint, 2017). The latter was introduced as an architectural approach to allow developing, deploying and running software solutions and their components. The motivation for this was to deliver software solutions more consistently, in an automated manner, while at the same time enabling horizontal scaling and integrating diverse technologies, different clouds, and legacy systems (Stine, 2015).

Several publications, authored by researchers from both academy and industry, define what a *cloud-native application* is, also in terms of the characteristics that it must feature for being considered such (Janssen, 2018; Fehling et al., 2014; Stine, 2015; Pahl et al., 2019; Vettor and Smith, 2019). At the same time, for ensuring a highly-automated deployment and configuration management, it is also crucial that the deployment technology itself offers support for processing cloud-native applications and their components. However, there is currently no reference work listing the features that a *deployment technology* should offer to support the deployment of arbitrary cloud-native applications, i. e., to feature what we shall call *cloud-native deploy-ability*.

In this paper, we make a first step in this direction. We indeed derive three essential features for deployment technologies based on the current cloud-native research and on the cloud-native application-centric attributes discussed therein. To do so, we first distill the main characteristics of cloud-native applications, as per their current perception in both white and gray reference literature. Due to the influence of microservices and their implementation based on containers, it matters that deployment technologies support emerging and de facto standard packaging formats and cloud service offerings. This means to support (natively, or via extensions) *all possible cloud service models* (e. g., FaaS or SaaS) on top of traditional IaaS offerings (Leymann et al., 2017) and to provide the opportunity to *target different cloud deployment models*, such a public, private, or hybrid clouds (Kratzke and Quint, 2017). Further, it emerges that reusable *deployment models* are needed to establish a repeatable end-to-end deployment automation and to specify the deployment of arbitrary applica-

tion components, their relations and desired configuration in a declarative manner, which is perceived as the most appropriate approach in practice (Wurster et al., 2019b; Herry et al., 2011).

It is finally also worth stressing that this paper aims at performing a first step towards classifying and assessing the support for deploying cloud-native applications provided by existing deployment technologies. Indeed, even if the presented features and characteristics in terms of supporting the deployment of cloud-native applications can already be used to compare existing deployment technologies, they are intended to set the basis for further research. We plan to expand and build on top of them, with the ultimate goal of providing a comprehensive framework to assess existing deployment technologies.

The rest of the paper is as follows. Sect. 2 distills currently recognized characteristics of cloud-native apps, which are taken in Sect. 3 to determine their influence on deployment technologies. Sect. 4 and Sect. 5 define and characterize cloud-native deployability, while Sect. 6 shows a deployment modeling example. Finally, Section 7 and Sect. 8 discuss related work and draw some concluding remarks.

2 CLOUD-NATIVE APPS: CHARACTERISTICS

Cloud-nativeness gained momentum over the last years, with the rise of the *Cloud Native Computing Foundation* (CNCF, 2019). The latter defines cloud-native technologies as the enablers for building and running scalable applications in modern, dynamic environments, such as public, private or hybrid clouds. Containers, (micro)service compositions, immutable deployment infrastructures and declarative APIs are concrete examples in this direction (Kratzke and Quint, 2017). They indeed enable the creation and enactment of loosely coupled systems (Hohpe and Woolf, 2004) that are resilient, manageable, and observable. If combined with robust automation, they allow engineers to enact changes while at the same time minimizing toil (CNCF, 2019).

Fehling et al. first isolated some of the characteristics of cloud-native applications by prescribing them to be *IDEAL* (Fehling et al., 2014), i. e., indicating that they should be (i) with *isolated state*, (ii) *distributed* in their nature, (iii) *elastic*, (iv) operated via *automated management systems*, and (v) made of *loosely coupled* components. By systematically integrating such characteristics with other emerging in published white literature, Kratzke and Quint (2017) managed to distill the properties characterizing a cloud-native application. They define it as a "*distributed, elastic and horizontally scalable system composed of (micro)services, which isolates states in a minimum of stateful components*" (Kratzke and Quint, 2017). Further, they define that the "*application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service platform*" (Kratzke and Quint, 2017). The findings by Kratzke and Quint (2017) were then confirmed by later published gray literature, e. g., the blog posts and whitepapers by Microsoft (Vettor and Smith, 2019), New Stack (Janakiram MSV, 2018), Pivotal (Pivotal Software, Inc., 2019), Red Hat (Red Hat, Inc., 2019), and Stackify (Janssen, 2018). Even if the list of cloud-native characteristics could be elaborated further, all above mentioned works agree with Kratzke and Quint (2017) in saying that the following are the main characteristics of cloud-native applications:

C1: Service-based architectures. Cloud-native applications are designed as suites of loosely-coupled (micro)services. Each service exists independently from the other services forming an application, hence allowing their independent development and operation (Kratzke and Quint, 2017). At the same time, a service typically interacts with other services in an application, which are discovered by exploiting features provided by the application runtime (Pivotal Software, Inc., 2019). This allows to compose services and form cloud-native applications.

C2: API-based interactions. Service-to-service communications in cloud-native application are API-based. The services forming an application indeed publish APIs to offer their functionalities, and they connect to and consume the APIs of other services in the application (Vettor and Smith, 2019). APIs are typically based on well-known standard protocols (e. g., REST over HTTP), and each component in a cloud-native application should be encapsulated by offering its API (Janssen, 2018; Vettor and Smith, 2019).

C3: State isolation. Cloud-native applications are designed with a clear separation among stateless and stateful services. Stateless services exist independently from stateful services, even if interacting with them, making them easy to scale in/out. Stateful services instead follow a different pattern for assuring higher availability and resiliency of persisted data (Fehling et al., 2014). This is done by typically exploiting natively scalable storage systems in the form of eventual consistent NoSQL databases (Kratzke and Quint, 2017).

C4: Self-contained service deployment. The services forming a cloud-native application are packaged in standardized, self-contained deployment units (Red Hat, Inc., 2019). Deployment units wrap services in virtual runtime environments containing everything services need to run, i. e., their source code and necessary runtime support (system tools, system libraries, etc.). This guarantees isolation among services running in different deployment units, and that they will always run the same, independently from the execution environment, whether it is on-premise or on third-party clouds, or whether it is a development, testing, or production environment (Janssen, 2018). Currently, beside PaaS and FaaS, containerization is a key enabler for self-contained deployment of the services in a cloud-native application (Kratzke and Quint, 2017; Pahl et al., 2019).

C5: Disposability. The actual instances of the services forming a cloud-native application are disposable (Vettor and Smith, 2019). This is a prerequisite for favoring both fast startups and scalability of services and graceful shutdown for leaving applications in correct state, which are both peculiar properties of cloud-native applications. Currently existing container-based technologies inherently satisfy this requirement (Pivotal Software, Inc., 2019; Pahl et al., 2019).

C6: Fault-resilience. Failures are first-class citizens in cloud application deployment and management (Brogi et al., 2018), hence requiring cloud-native applications to treat failures as first-class citizens as well. Cloud-native applications indeed assume that service instances can fail at any time and feature mechanisms ensuring fault-resilience (Kratzke and Quint, 2017; Soldani et al., 2018). The services forming an application are built for tolerating the failure of the other services they interact with, typically by exploiting cloud-native design patterns, e. g., circuit breakers (Vettor and Smith, 2019). At the same time, the platforms exploited for deploying and managing cloud-native applications are suitably configured to automatically recover failed service instances.

C7: Infrastructure abstraction. Cloud-native application abstract from underlying infrastructure and operating system dependencies, by operating at a higher abstraction level (Janssen, 2018; Pivotal Software, Inc., 2019). An enabler in this direction is the exploitation of above mentioned self-contained deployment units, which allow to distribute the services of an application over multiple, heterogeneous clouds (Kratzke and Quint, 2017). This is typically achieved by exploiting self-service deployment platforms, allowing to ship and scale deployment units on IaaS or PaaS clouds (Red Hat, Inc., 2019).

C8: Infrastructure as code. Cloud-native applications are highly automated, from their delivery and deployment to their management, scaling, and monitoring (Janssen, 2018; Vettor and Smith, 2019). Such automation is typically achieved using *infrastructure as code* (Pivotal Software, Inc., 2019), i. e., through machine-readable files allowing to specify the desired configuration for an application and its components (Morris, 2016).

C9: Policy-driven elasticity. Cloud-native applications are horizontally scalable, meaning that each of their services can be scaled in/out, by adapting the amount of replicas (Janssen, 2018). This is done by relying on self-service deployment platforms, which are configured through scaling policies, indicating how to dynamically (re-)allocate computing resources to services, to continuously satisfy the ongoing needs of an application (Pivotal Software, Inc., 2019; Red Hat, Inc., 2019).

C10: CI/CD compliance. Cloud-native applications are developed by embracing *continuous integration and continuous deployment/delivery* (CI/CD) DevOps paradigm (Kratzke and Quint, 2017). Each service of a cloud-native application is developed in a separate code base and possibly equipped with its own deployment pipeline (Vettor and Smith, 2019). This allows different services to be developed by different teams using different technologies, and to release service updates as soon as they are available, also throughout short and continuous delivery cycles (Janssen, 2018).

Deployment technologies can hence be considered to feature *cloud-native deploy-ability* if they natively support cloud-native applications in exhibiting the above-listed characteristics. A discussion concerning the influence on deployment technologies is presented in the following section.

3 ON DEPLOYING CLOUD-NATIVE APPLICATIONS

This section presents the first contribution of this work in the form of an analysis of required features of deployment technologies for automating the deployment of arbitrary cloud-native applications. The definition by Kratzke and Quint (2017) as well as the

characteristics C4 (standardized, self-contained service deployment) and C7 (infrastructure abstraction) imply that cloud-native applications are intended to run in public, private, or hybrid cloud environments offering self-service capabilities. Thus, deployment technologies must support the deployment of cloud-native applications to different cloud providers, environments, and platforms.

Further, characteristics C3 (state isolation), C4 (standardized, self-contained service deployment), and C5 (disposability) imply the fact that a deployment technology needs to provide the ability to deploy application components on top of IaaS, i. e., to containerized environments, PaaS offerings, and FaaS platforms. Even the use and instrumentation of existing SaaS offerings to implement application components need to be supported. Thus, deployment technologies must support the use of all cloud service models (XaaS) to deploy cloud-native applications, i. e., to deploy on IaaS, PaaS, FaaS, and SaaS.

To foster automation and to conform to C10 (CI/CD compliance), deployment technologies must provide interfaces that can be used in combination with existing technologies for enacting and automating deployment processes, e. g., Jenkins or Travis. Thus, deployment technologies must provide one or more options to integrate with external systems, e. g., by providing SDKs, command-line interfaces, or APIs, either local or remotely accessible (such as REST APIs over HTTP). For example, AWS CloudFormation provides several SDKs in different programming languages, an HTTP API, and a CLI to automate the interaction with AWS.

On top of that, to support C8 (infrastructure as code) and to establish a repeatable and consistent workflow to automate the release cycle, deployment technologies must provide the ability to express the application deployment in machine-readable formats, i. e., in so-called *deployment models*. They can be categorized into two types: (i) imperative models and (ii) declarative models (Endres et al., 2017). The main idea of imperative models is to describe a detailed, executable process specifying all necessary technical tasks to be executed, their implementations, and their order. In contrast, declarative models only describe the components to be deployed, their configurations, and the relations between them, but hardly provide executional details. Declarative models, hence, need to be interpreted by a deployment technology deriving the technical deployment instructions to reach the desired state. Therefore, C1 (service-based architectures) implies that deployment technologies must provide the ability to structure the application deployment into physical, functional, or logical units by defining the deployment specifics of application components inside a deployment model, while C9 (policy-driven elasticity) implies that the definition of such application components should be done in a declarative manner, as this is the most appropriate approach for application deployment and configuration management (Herry et al., 2011; Wurster et al., 2019b).

Moreover, the characteristics C3 (state isolation), C4 (standardized, self-contained service deployment), and C7 (infrastructure abstraction) imply that deployment technologies must offer the ability to express different types of application components in their deployment. For example, it must be possible to express the deployment of a virtual machine, the installation of a web server component on top of it, as well as the deployment of *containerized* applications and *functions* running on a FaaS platform. However, a deployment technology must not support all kinds of cloud service models from all kinds of cloud providers natively. The deployment technology itself could be extensible in two ways: (i) at the deployment modeling language level, where ontological types are used for extensibility and provided as reusable entities across different application deployments (Bergmayr et al., 2018), or (ii) at the deployment system level, where a plugin mechanism provides a clearly described interfaces to extend the set of deployable component types or host environments on which components can be deployed. A plugin, in this context, is a software component that adds a specific feature to an existing deployment technology without changing the actual source code of it.

Lastly, characteristic C2 (API-based interaction) implies the ability to express the notion that a specific application component connects to another one. For example, the deployment model of a deployment technology must support the definition of a respective relation between a web application and an arbitrary *backing service* expressing the fact that the web application connects to the service during runtime.

In summary, all characteristics (C1-C10) from Sect. 2 impact on what a deployment technologies should support, if looking at them from the perspective of automating the deployment of cloud-native applications. From this perspective, deployment technologies must support the automated deployment by using deployment models to deploy on different cloud providers and platforms and on all cloud service models. In the following section, we build on top of this analysis to derive and introduce three essential features for deployment technologies for automating the deployment of cloud-native applications, i. e., to feature *cloud-native deploy-ability*.

4 CLOUD-NATIVE DEPLOY-ABILITY

In Sect. 2 we presented application-centric characteristics, based on the current cloud-native research, that must be supported by applications to be considered as cloud-native. In Sect. 3 we analyzed what exactly it means for a deployment technology to support those characteristics. In this section, we build on top of the result of our analysis and discuss an initial set of features that must be supported by deployment technologies to deploy arbitrary cloud-native applications, i. e., (i) support for multiple cloud providers and platforms, (ii) support for all cloud service models (XaaS), (iii) usage of deployment models supporting arbitrary components. We hereafter present a first definition of *cloud-native deploy-ability* recapping features (i-iii), which we then explain in detail in Sects. 4.1, 4.2, and 4.3.

> **Cloud-native deploy-ability** describes the ability to deploy arbitrary application components, concerning all cloud service models, that can be vertically "hosted on" or horizontally "connected to" any other component or service hosted on any cloud provider, cloud platform, or hybrid environment. This includes supporting the processing of declarative deployment models given in machine-readable formats fostering automation.

4.1 Support for Multiple Cloud Providers and Platforms

From the analysis in Sect. 3 we can derive that a deployment technology must support the deployment of application components to multiple cloud providers and platforms to comply with C4 (standardized, self-contained service deployment) and C7 (infrastructure abstraction). This includes the deployment to public cloud offerings such as Amazon Web Services (AWS) or Microsoft Azure. Further, this also requires the capability to deploy applications to self-hosted platforms such as OpenStack. Even more, deployment technologies need to support hybrid cloud deployments, i. e., the distribution of application components to different cloud providers. For example, by executing an automated deployment, it must be possible to deploy certain application components to a public cloud provider (e. g., AWS) and others to a self-hosted platform (e. g., OpenStack). This increases flexibility and enables the possibility to target different environments for different use cases in the application development lifecycle, e. g., to deploy differently for *development*, *testing*, and *production*.

4.2 Support for all Cloud Service Models (XaaS)

The analysis in 3 showed that providing the ability to deploy on IaaS offerings may not be enough to deal with cloud-native applications (Leymann et al., 2017). To support the *disposability* (C5), *self-contained* (C4), and *state isolation* (C3) aspects of cloud-native apps, deployment technologies must be able to target containerized environments as well as PaaS offerings. Further, lately the *serverless computing* paradigm has gained momentum (CNCF, 2018a). The term serverless is often linked with the arisen *Function as a Service* (FaaS) cloud delivery model. The idea of FaaS is to develop and deploy custom server-side logic in the form of ephemeral and stateless functions employing an *event-driven programming model* (CNCF, 2018a). With the use of serverless and FaaS, developers increase *infrastructure abstraction* as the management and scaling of required computing resources remain fully the responsibility of cloud providers. This is why FaaS can be considered as an enabler for cloud-native application components and, therefore, must be supported by deployment technologies (Gannon et al., 2017; Roberts, 2016). Moreover, to exploit the full stack of cloud service models, deployment technologies have to enable the use and instrumentation of existing SaaS offerings to implement application components. Thus, deployment technologies must support the use of all cloud service models to deploy cloud-native applications.

4.3 Usage of Deployment Models Supporting Arbitrary Components

Automation plays a major role in developing and running cloud-native applications. Further, the emergence of *containers* to implement microservices and new cloud service models such as FaaS, requires that a deployment technology can be used to express the deployment of such applications components using their deployment model mechanics. This requires the usage of *deployment models* and the ability to define the deployment of arbitrary applications components in a declarative manner, which in turn complies with the presented characteristics C2 (API-based interaction), C3 (state isolation), C4 (standardized, self-contained service deployment), C7 (infrastructure abstraction), and C8 (infrastructure as code).

This means for deployment technologies to provide the ability to express the application deployment in a declarative model by defining arbitrary components based on arbitrary component types and respective relations among them. Notably, component types are either natively provided, or the deployment technology is extensible (cf. Sect. 3) such that custom component types can be introduced. As depicted in Fig. 1, this means that it must be possible to express constellations of components and relations defining that a component `A` is "hosted on" or "contained in" a component `B`, such that component `A` is the *hosted component* and component `B` is the *hosting component*. For example, a Tomcat web server is hosted on a Ubuntu virtual machine. Further, deployment technologies must be able to express constellations of components and relations specifying that a component `X` "connects to" or "invokes" a component `Y`, such that component `X` is the *connecting component* and component `Y` is the *connected component*. For example, a web application needs to establish a connection to a managed messaging service.

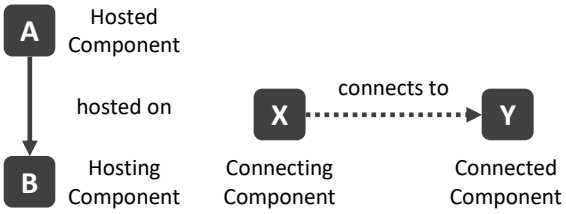

Figure 1: Deployment technologies must be able to deploy components hosted on other components (left) and to deploy components connecting to other components (right).

5 DEPLOYMENT MODELING SUPPORT: FORMALIZATION

In the last section we identified that an important aspect is to express arbitrary constellations of components and relations in deployment models. In terms of deployment model expressiveness, we can characterize cloud-native deploy-ability in terms of two separate sub-characteristics, i. e., *vertical* and *horizontal* deploy-ability. Therefore, as a second contribution of this work, we hereafter provide a formal definition of such characteristics.

Wurster et al. (2019b) showed that the top-most deployment technologies employing declarative models share similar modeling constructs. All of them provide the notion of *components* to describe logical units, *relations* to specify dependencies between components, and *component types* and *relation types* to convey reusable semantics, which was published as the Essential Deployment Metamodel (EDMM). It has been derived by conducting a systematic review of the 13 top-most deployment automation technologies and describes the essential modeling elements supported by declarative deployment models used in practice (Wurster et al., 2019b). Figure 2 shows a simplified version of EDMM describing components, used to form an application deployment model, as well as the component types, allowing to distinguish them and giving them semantics.

According to EDMM, a *Component* is a physical, functional, or logical unit of an application, while a *Component Type* is a reusable entity that specifies the semantics of a component that has this type assigned. For example, a component representing the deployment of a Tomcat server may be of type *Tomcat* while a component representing the provisioning of a virtual machine (VM) may be of type *Compute*. While a component represents the deployment for a specific application, the component type can be used in different deployment models. Components often depend on other components, which is specified by relations between components. A *Relation* is a directed physical, functional, or logical dependency between exactly two components, while a *Relation Type* is a reusable entity that specifies the semantics of a relation that has this type assigned. For example, a relation between a Tomcat server and its hosting compute component may of type *hosted on*, while a relation expressing that a component connects to another component may of type *connects to*.

Formally, an EDMM deployment model is a directed graph, which nodes represent application components, and which edges represent inter-component relations. Let $\mathcal{M}$ be the set of all *valid* EDMM deployment models (i. e., all deployment models having syntactically and semantically correct constellations of components and relations) then a deployment model $m \in \mathcal{M}$ is defined by the following tuple[1]:

$$m = \langle C_m, R_m, CT_m, RT_m, \mathsf{type}_m, \mathsf{supertype}_m \rangle$$

The elements of EDMM are defined as follows:

- C_m is the set of *Components* in m, whereby each $c_i \in C_m$ represents a component of the application to be deployed.
- $R_m \subseteq C_m \times C_m$ is the set of *Relations* in m, whereby each $r_i = (c_s, c_t) \in R_m$ models a relationship between two components, with c_s being the source of the relationship and c_t being its target.

[1]The tuple defines a deployment model according to the simplified EDMM considered in this paper. Full EDMM definitions have been provided by Wurster et al. (2019b).

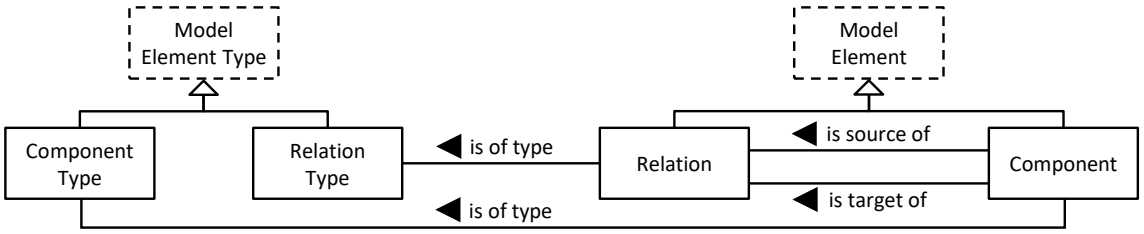

Figure 2: Simplified Essential Deployment Metamodel (EDMM).

Table 1: Formal definitions of *vertical* and *horizontal* deployability based on the simplified EDMM. Therein, *hostedOn* is a generic relationship type denoting the hosting of a component in another, while *connectsTo* is a generic relationship type denoting the connection of a component to another (cf. Fig. 1). Additionally, $\mathsf{canDeploy}(\mathsf{t},\mathsf{m})$ is a predicate used to denote that a given (valid) deployment model $m \in \mathcal{M}$ can be deployed by deployment technology $t \in \mathcal{T}$.

$\mathsf{arbitraryVerticalDeployAbility}(t) \iff$ $\forall ct_1, ct_2 \in CT \,.\, \exists m \in \mathcal{M}$: $(\exists c_1, c_2 \in C_m, r = \langle c_1, c_2 \rangle \in R_m \,.\, \mathsf{type}(c_1) = ct_1 \wedge \mathsf{type}(c_2) = ct_2 \wedge \mathit{hostedOn} \in \mathsf{supertypes}(r)) \wedge \mathsf{canDeploy}(t, m)$
$\mathsf{arbitraryHorizontalDeployAbility}(t) \iff$ $\forall ct_1, ct_2 \in CT \,.\, \exists m \in \mathcal{M}$: $(\exists c_1, c_2 \in C_m, r = \langle c_1, c_2 \rangle \in R_m \,.\, \mathsf{type}(c_1) = ct_1 \wedge \mathsf{type}(c_2) = ct_2 \wedge \mathit{connectsTo} \in \mathsf{supertypes}(r)) \wedge \mathsf{canDeploy}(t, m)$

- CT_m is the set of *Component Types* in m, whereby each $ct_i \in CT_m$ describes the semantics for the components that have this type assigned.
- RT_m is the set of *Relation Types* in m, whereby each $rt_i \in RT_m$ describes the semantics for the Relations that have this Relation Type assigned.
- type_m is a map that assigns all *Model Elements* in m to their respective *Model Element Type*. Letting $ME_m := C_m \cup R_m$ be the union set of all *Model Elements* of m, and $MET_m := CT_m \cup RT_m$ be the unioin set of all *Model Element Types* of m, type_m is defined as follows:

$$\mathsf{type}_m : ME_m \to MET_m$$

- $\mathsf{supertype}_m$ is a partial function mapping each *Model Element Type* to its respective supertype. It associates a $met_i \in MET_m$ with a $met_j \in MET_m$ where $i \neq j$, i. e., that met_j is the supertype of met_i:

$$\mathsf{supertype}_m : MET_m \to MET_m$$

For simplifying notation, we hereafter also use the partial function $\mathsf{supertypes}_m$ to map a *Model Element Type* $met_i \in MET_m$ to the set containing all supertypes. The latter can be easily computed by computing the transitive closure of supertype, i. e., $\mathsf{supertypes} = \mathsf{supertype}^+$:

$$\mathsf{supertypes}_m : MET_m \to 2^{MET_m}$$

Following this, we define two requirements for modeling *arbitrary component-relation-constellations* as a feature for deployment technologies to support deploying arbitrary cloud-native applications:

Arbitrary Vertical Deploy-Ability. The ability to deploy arbitrary types of hosted components on arbitrary types of hosting components. This is depicted in Fig. 1 (left-hand side) and formalized in Table 1 (by predicate $\mathsf{arbitraryVerticalDeployAbility}$).

Arbitrary Horizontal Deploy-Ability. The ability to deploy arbitrary types of components connecting to other types of components. This is depicted in Fig. 1 (right-hand side) and formalized in Table 1 (by predicate $\mathsf{arbitraryHorizontalDeployAbility}$).

However, deployment technologies may be restricted in their ability to use arbitrary constellations of components and relations. AWS CloudFormation (Amazon Web Services, Inc., 2018), for example, has a limited set of component types as this deployment technology is only intended to be used with the AWS cloud services. Further, Kubernetes (CNCF, 2018b) is considered to be multi-cloud deployable (many cloud offer managed clusters), but is limited in component types as only containers can be used. If such technologies comply with the derived feature requirements, but are restricted in component and relation types, we speak of *single-environment* cloud-native deployability. Notably, we speak of *multi-environment* cloud-native deploy-ability if a deployment technology complies with the definition above. For example, TOSCA (OASIS, 2019) and Terraform (HashiCorp, 2018) are considered as such, since both support the usage of arbitrary component types, either natively or by providing respective extension mechanisms.

6 DEPLOYMENT MODELING EXAMPLE

We hereafter introduce a simple yet effective example to show a deployment model that follows our formalized description from Sect. 5 based on a concrete cloud-native deployment scenario. Figure 3 depicts the scenario and shows a Java application named "Pet Clinic", implemented using the Java framework *Spring Boot*. In the depicted scenario, the web application is hosted on *AWS Beanstalk*, the platform as a service (PaaS) offering by Amazon Web Services (AWS). Further, the application connects to a *MySQL* database to store its data. The database in this cloud-native scenario is hosted on *Amazon Aurora*, which is a fully managed MySQL database as a service (DBaaS) offering by AWS. By using EDMM's metamodel entities, we can express the structure of such a cloud-native application in a technology-agnostic way. For example, the "Pet Clinic" application is a component, which also references a component type (Web App) to define further semantics. Further, the "Database" component defines several properties that can be used to configure the deployment. Moreover, the Java application and the database component have an artifact attached which is, for example, a packaged WAR file in case of the Java application and a SQL file containing the actual database schema and initial data of the database component.

Even this simple example shows concretely the requirement that deployment technologies must be able to deploy arbitrary component types on top of other ones, e. g., that the Java web application is hosted on AWS Beanstalk. Further, it highlights that it is also required to express that arbitrary components connect to other types of components, e. g., that the web application connects to a database at runtime. Hence, the deployment technology has to execute the deployment of the components in a certain order, e. g., the database stack on the right-hand side of Fig. 3 must be deployed before the application stack on the left-hand side. The order in EDMM terminology is defined by defining typed relations ("HostedOn" or "ConnectsTo" relations in Fig. 3) between two components. Moreover, in the case of so-called *connects to* relations, it implies that a deployment technology must provide the possibility to inject endpoint and credential information of related components. In practice, cloud-native applications often use environment variables to configure the application's logic. For example, the information required to connect to the modeled database is made available at runtime by injecting the respective deployment model properties into the application's environment.

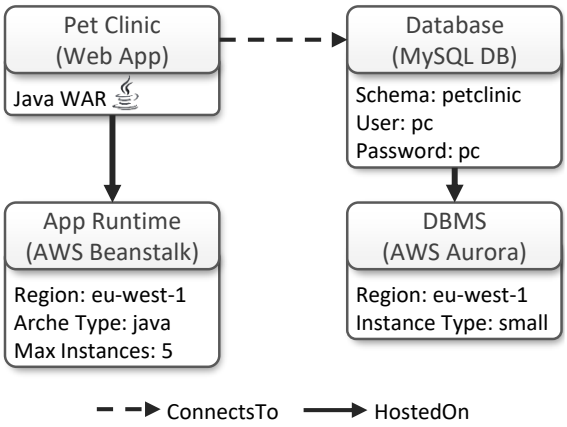

Figure 3: Simple cloud-native application expressed as EDMM-based deployment model[2].

The concrete deployment model, which can be executed using a certain deployment technology can be created manually by an application developer or operations engineer. Using Ansible, a *Playbook* can be developed containing the definitions to deploy the application: first, the deployment of the database component hosted on AWS Aurora, then the Java application hosted on AWS Beanstalk. Additionally, the Playbook also defines that the database connection credentials are injected using environment variables to the AWS Beanstalk runtime such that the Java application can be started accordingly. Similarly, this can be also realized by using AWS CloudFormation, the *infrastructure as code* (IaC) tooling provided by AWS. In a declarative manner, a CloudFormation template allows you to define and provision the needed AWS resources including their configuration and wiring.

However, EDMM as a normalized metamodel provides the basis for technology-independent deployment modeling. Also, recent work showed that it can be used to facilitate the automated transformation in different concrete deployment technologies Wurster et al. (2019a). Using EDMM tooling, a user can model the deployment of an application graphically by specifying the components to be deployed, their configurations, implementations, and relations to other components. By applying transformation rules to such a model, depending on the selected target deployment technology, the abstract EDMM model can be transformed into a concrete deployment technology. The output of the EDMM Transformation Framework is an executable, technology-specific deployment model, which can be executed using the selected technology. The tooling is open-source and respective examples are available online on GitHub[3].

[2] http://bit.ly/39OYPMx
[3] https://github.com/UST-EDMM

7 RELATED WORK

Several existing research efforts provide statements regarding *cloud-nativeness* of software solutions, with that by Kratzke and Quint (2017) perhaps being one of the most prominent efforts in this direction. By studying published research (i. e., Fehling et al. 2014; Kratzke and Peinl 2016; Leymann et al. 2017; Stine 2015; just to mention some) and existing trends in engineering such applications, they defined the term "cloud-native application" as a distributed, elastic, and horizontally scalable system composed of services and operated on self-service platforms, while the services itself are designed as self-contained deployment units according to cloud design patterns.

Other noteworthy studies defining *cloud-nativeness* of applications come from practitioners daily working with them, e. g., blog posts and whitepapers published by renowned IT players, practitioners and companies (Vettor and Smith, 2019; Red Hat, Inc., 2019; Pivotal Software, Inc., 2019; Janakiram MSV, 2018; Janssen, 2018). Also, the Cloud Native Computing Foundation was founded in 2015 to help aligning the industry and coordinate its evolution, with the support of large companies (i. e., Google, IBM). They shaped the notion of *cloud-native applications* over the last years (CNCF, 2019) and defined it as an enabler for building and running scalable applications in modern environments.

The study by Kratzke and Quint (2017) and the industrial effort mentioned above however focus on defining what a cloud-native *application* is. In our study, we instead plan to pursue a different characterization, i. e., knowing how cloud-native application are characterized, can we understand whether an existing *deployment technology* can actually support the deployment of arbitrary cloud-native applications?

On the deployment technology side, there are several publications available comparing and classifying tools and platforms regarding their set of features and their intended use (Masek et al., 2018; Wettinger et al., 2016; Weerasiri et al., 2017). Further, Wurster et al. (2019b) conducted a systematic review of the 13 top-most deployment automation technologies and introduced the categorization *general-purpose*, *provider-specific*, and *platform-specific* of such. The categorization was done to find commonalities between deployment technologies to derive a metamodel for creating technology-independent deployment models, while those being transformable into concrete technologies (Wurster et al., 2019b,a).

However, the recent publications either considering the engineering part of the application itself or classifying deployment technologies regarding their features or use cases. To the best of our knowledge, no published work provided means to analyze deployment technologies regarding their ability to deploy arbitrary cloud-native applications. That is the main reason motivating our work, in which we aim at analyzing what does it mean to deploy arbitrary cloud-native applications and what kind of features a respective deployment technology must provide.

8 CONCLUSIONS

Cloud-nativeness emerged as a paradigm and enabler for building and running scalable applications in modern cloud environments (Kratzke and Quint, 2017). Beyond that, it is equally important that such built applications can be effectively deployed. In this work, we presented three features a deployment technology requires to deploy arbitrary cloud-native applications. We derived these attributes by analyzing the current research around what cloud-native means. We then exploited such characteristics to provide a first definition of *cloud-native deploy-ability*.

It is worth highlighting that both the presented features and the definition of *cloud-native deploy-ability* aims at performing a first step towards classifying and assessing the support for deploying cloud-native applications provided by existing deployment technologies. We plan to refine and extend them in future work, to use them in a comprehensive review and classification framework to assess existing deployment technologies concerning their ability to deploy arbitrary cloud-naive applications. In general, other dimensions should be considered to fully classify deployment technologies, e. g., characteristics of the employed modeling language or the extensibility of the deployment technology itself. Therefore, as immediate future, we emerge our contribution to a complete review framework capable of evaluating and classifying deployment technologies, to ultimately provide a decision support system for the selection of the right deployment technology.

Further, in future work we want to address influencing factors such as *service level agreements* (SLAs) or scalability attributes of components. For example, based on EDMM we are able to annotate SLAs for certain components in a technology-agnostic way and provide additional tooling to check the conformance of those SLAs for different technologies and cloud environments.

ACKNOWLEDGEMENTS. This work is partially funded by the EU project *RADON* (825040) and the projects *AMaCA* (POR-FSE) and *DECLware* (University of Pisa, PRA_2018_66).

REFERENCES

Amazon Web Services, Inc. (2018). AWS CloudFormation Official Site. https://aws.amazon.com/de/cloudformation.

Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., and Kappel, G. (2018). A Systematic Review of Cloud Modeling Languages. *ACM Computing Surveys (CSUR)*, 51(1):1–38.

Brogi, A., Canciani, A., and Soldani, J. (2018). Fault-aware management protocols for multi-component applications. *Journal of Systems and Software*, 139:189 – 210.

CNCF (2018a). CNCF Serverless Whitepaper v1.0. https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview.

CNCF (2018b). Kubernetes Official Site. https://kubernetes.io.

CNCF (2019). CNCF Cloud Native Definition v1.0. https://github.com/cncf/toc/blob/master/DEFINITION.md.

Endres, C., Breitenbücher, U., Falkenthal, M., Kopp, O., Leymann, F., and Wettinger, J. (2017). Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS)*, pages 22–27. Xpert Publishing Services.

Fehling, C., Leymann, F., Retter, R., Schupeck, W., and Arbitter, P. (2014). *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Publishing Company, Inc.

Gannon, D., Barga, R., and Sundaresan, N. (2017). Cloud-Native Applications. *IEEE Cloud Computing*, 4(5):16–21.

HashiCorp (2018). Terraform Official Site. https://www.terraform.io.

Herry, H., Anderson, P., and Wickler, G. (2011). Automated Planning for Configuration Changes. In *Proceedings of the 25th International Conference on Large Installation System Administration (LISA 2011)*, pages 57–68. USENIX.

Hohpe, G. and Woolf, B. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley.

Janakiram MSV (2018). 10 Key Attributes of Cloud-Native Applications. The New Stack, https://thenewstack.io/10-key-attributes-of-cloud-native-applications.

Janssen, T. (2018). The Path to Cloud-Native Applications. Stackify, https://stackify.com/cloud-native.

Kratzke, N. and Peinl, R. (2016). ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects. In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 1–10.

Kratzke, N. and Quint, P.-C. (2017). Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study. *Journal of Systems and Software*, 126:1–16.

Leymann, F., Breitenbücher, U., Wagner, S., and Wettinger, J. (2017). Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation. *Cloud Computing and Services Science*, pages 16–40.

Masek, P., Stusek, M., Krejci, J., Zeman, K., Pokorny, J., and Kudlacek, M. (2018). Unleashing Full Potential of Ansible Framework: University Labs Administration. In *2018 22nd Conference of Open Innovations Association (FRUCT)*.

Morris, K. (2016). *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media, Inc., 1st edition.

OASIS (2019). TOSCA Simple Profile in YAML Version 1.2.

Pahl, C., Brogi, A., Soldani, J., and Jamshidi, P. (2019). Cloud Container Technologies: A State-of-the-Art Review. *IEEE Transactions on Cloud Computing*, 7(3):677–692.

Pivotal Software, Inc. (2019). What are Cloud-Native Applications? https://pivotal.io/cloud-native.

Red Hat, Inc. (2019). The Path to Cloud-Native Applications. https://www.redhat.com/en/resources/path-to-cloud-native-applications-ebook.

Roberts, M. (2016). Serverless Architectures. http://martinfowler.com/articles/serverless.html.

Soldani, J., Tamburri, D. A., and Van Den Heuvel, W.-J. (2018). The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, 146:215 – 232.

Stine, M. (2015). *Migrating to Cloud-Native Application Architectures*. O'Reilly Media, Inc.

Vettor, R. and Smith, S. (2019). *Architecting Cloud-Native .NET Apps for Azure*. Microsoft. https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native.

Weerasiri, D., Barukh, M. C., Benatallah, B., Sheng, Q. Z., and Ranjan, R. (2017). A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Computer Surveys*, 50(2).

Wettinger, J., Breitenbücher, U., Kopp, O., and Leymann, F. (2016). Streamlining DevOps Automation for Cloud Applications using TOSCA as Standardized Metamodel. *Future Generation Computer Systems*, 56:317–332.

Wurster, M., Breitenbücher, U., Brogi, A., Falazi, G., Harzenetter, L., Leymann, F., Soldani, J., and Yussupov, V. (2019a). The EDMM Modeling and Transformation System. In *Service-Oriented Computing – ICSOC 2019 Workshops*. Springer.

Wurster, M., Breitenbücher, U., Falkenthal, M., Krieger, C., Leymann, F., Saatkamp, K., and Soldani, J. (2019b). The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *SICS Software-Intensive Cyber-Physical Systems*.