Self-Contained Service Deployment Packages

Michael Zimmermann, Uwe Breitenbücher, Lukas Harzenetter, Frank Leymann
and Vladimir Yussupov
Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstrasse 38, 70569 Stuttgart, Germany

Keywords: Deployment Model, Deployment Automation, Provisioning, Self-Contained, Dependency Resolving, TOSCA.

Abstract: Complex applications are typically composed of multiple components. In order to install these components all their dependencies need to be satisfied. Typically these dependencies are resolved, downloaded, and installed during the deployment time and in the target environment, e.g., using package manager of the operating system. However, under some circumstances this approach is not applicable, e.g., if the access to the Internet is limited or non-existing at all. For instance, Industry 4.0 environments often have no Internet access for security reasons. Thus, in these cases, deployment packages without external dependencies are required that already contain everything required to deploy the software. In this paper, we present an approach enabling the transformation of non-self-contained deployment packages into self-contained deployment packages. Furthermore, we present a method for developing self-contained deployment packages systematically. The practical feasibility is validated by a prototypical implementation following our proposed system architecture. Moreover, our prototype is evaluated by provisioning a LAMP stack using the open-source ecosystem OpenTOSCA.

1 INTRODUCTION

Cloud computing is of vital importance for the realization of modern IT systems focusing on automated deployment and management (Leymann, 2009). Cloud properties, for instance, scalability, pay-on-demand pricing, or self-service as well as new paradigms, such as edge or fog computing (Mahmud et al., 2018) enable developers to build flexible and automated cloud applications (Leymann, 2009). A wide range of domains can benefit from these new opportunities, for example, mobility (Guo et al., 2017), health care (Haque et al., 2014), energy management (Shrouf and Miragliotta, 2015), and scientific computing in general (Iosup et al., 2011). However, installing, configuring, and running complex software on a remote IT infrastructure is a challenging task that requires detailed expertise (Breitenbücher et al., 2013). First, the middleware as well as all dependencies required by the software need to be installed. Secondly, the software itself needs to be deployed. And finally, the software and its components need to be configured as well as connected with each other. Dependent of the involved software, these tasks can get complex, are time-consuming and error-prone, and, therefore, are not efficient if done manually (Eilam et al., 2006; Breitenbücher et al., 2013).

The deployment as well as the management of applications can be automated using various deployment technologies (Wurster et al., 2019): (i) provider-specific systems, (ii) provider-independent, but platform-specific technologies, (iii) general-purpose technologies, or furthermore, (iv) provider- and technology-agnostic standards, such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) (OASIS, 2013a; OASIS, 2013b).

Typically, regardless of whether the deployment of the application is executed manually or automatically, external dependencies have to be resolved, downloaded, and installed during the deployment of the application within the respective environment (Wettinger et al., 2014). For example, often the Linux package handling utility *apt-get* is used in scripts to install required dependencies, e.g., *apt-get install python*. This enables the creation of light-weighted deployment packages, since external dependencies are resolved and downloaded during the deployment phase and don't need to be added to it in advance. Besides minimizing the package size, this also allows to only use scripts. Two different types of external dependencies can be distinguished: (i) dependencies that are required in order to deploy and manage the application and (ii) dependencies of the application itself, required for properly executing it.

Zimmermann, M., Breitenbücher, U., Harzenetter, L., Leymann, F. and Yussupov, V.
Self-contained Service Deployment Packages.
DOI: 10.5220/0009414903710381
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 371-381
ISBN: 978-989-758-424-4

However, resolving and downloading external dependencies can slow down the deployment significantly and furthermore, when access to the Internet is limited, unstable, or non-existing at all, this can prevent the deployment of the application. Moreover, the download from external sources could compromise the security of applications. For example, a use case is in the area of Industry 4.0, when software should be provisioned in remote environments with no or only restricted Internet access for security reasons (Tsuchiya et al., 2018). Another use case is in the field of eScience, where dependencies need to be packaged together with the research software in order to archive them and thus, enable the reusability of the software, because the external dependencies might get stale over time (Zimmermann et al., 2018a). Thus, in such cases, fully self-contained deployment packages are required, i.e., archives without external dependencies, containing everything required for the deployment and management of the application.

In this paper, we tackle these issues. We present a system architecture supporting the automated transformation of a non-self-contained deployment package into a self-contained deployment package. Moreover, we present a method, utilizing our provided system architecture, enabling the systematic developing of self-contained deployment packages in order to ease the development of them. Our approach is useful for both cloud and IoT applications, since for both, use cases exist which benefit from our approach. To validate the practical feasibility of the approach, we present a prototypical implementation of a *Self-Containment Packager Framework* based on the TOSCA standard. TOSCA enables the modeling and automated execution of the deployment and management of cloud applications. Furthermore, we evaluate our approach by comparing the time required for provisioning a non-self-contained TOSCA-based deployment package with the time required for the transformation and the provisioning of a self-contained TOSCA-based deployment package. For provisioning, we are using the open-source tool OpenTOSCA.

The remainder of this paper is organized as follows. In Section 2, background and fundamentals of this work are introduced, e.g., deployment models. Moreover, this section motivates our approach by presenting existing problems and limitations we tackle in this work. Section 3 introduces our system architecture of a Self-Containment Packager Framework and presents our method for developing self-contained deployment packages. In Section 4, we present our prototypical implementation based on the TOSCA standard. In Section 5, related work is discussed. Section 6 concludes this paper and discusses future work.

2 FUNDAMENTALS, BACKGROUND & MOTIVATION

Since our approach and our prototype is based on deployment models, in the following subsection, we first introduce some basic information about them. Furthermore, we illustrate the problems of creating deployment packages enabling the deployment and management of cloud and IoT applications in remote IT environments, that take place when utilizing them, by means of three different use cases in the area of eScience, automotive, and Industry 4.0. For example, deploying an application into an environment without Internet access or preserving dependencies. Moreover, we present an overview of state-of-the-art approaches for creating such deployment packages for cloud and IoT applications. A deployment package, in our case, specifies the deployment of an application and can contain arbitrary artifacts and files for achieving that, for example, Bash scripts, Chef Cookbooks, or Dockerfiles, depending on the used deployment technology as well as deployment language.

2.1 Deployment Models & Topologies

The manual deployment of services consisting of multiple components is complex, hard to repeat, and error-prone (Eilam et al., 2006; Breitenbücher et al., 2013). Therefore, the automation of application deployment and management over its entire lifecycle, e.g., install, start, stop, or update is essential. By using maintainable and reusable deployment models, describing the software components as well as the infrastructure components of an application, an automated and reliably repeatable deployment solution can be created. Various deployment automation systems are available, supporting the model-based and automated deployment of applications (Bergmayr et al., 2018). Deployment models can be imperative or declarative (Endres et al., 2017). Imperative deployment models describe all the single steps required for the deployment of an application. Declarative deployment models, on the other hand, describe the desired state of an application that shall be deployed. These models, called topologies, are graph-based and describe the structure of an application consisting of its components, their relations to each other, and properties. Declarative deployment models are widely accepted in industry and research, as the most appropriate approach for the automated deployment of applications and configuration management (Herry et al., 2011). Various technologies are following this approach, for example, Kubernetes, Puppet, and Chef.

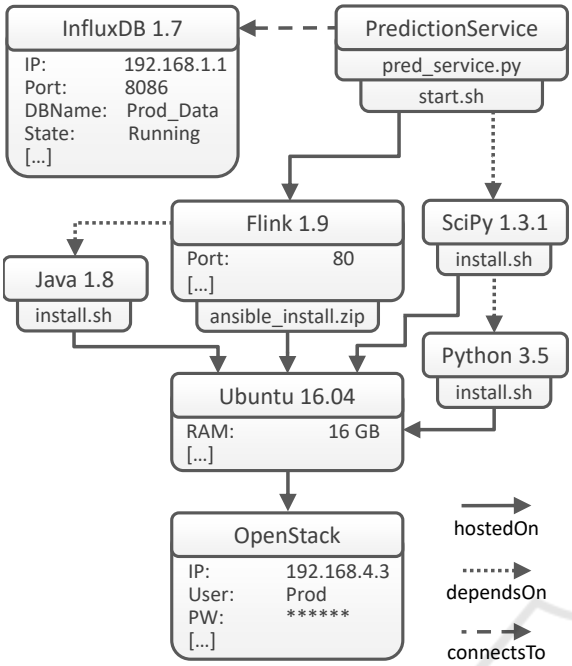

Figure 1: Exemplary declarative deployment model.

In Fig. 1 an exemplary declarative deployment model is depicted consisting of various components, such as Ubuntu, Flink, and InfluxDB. The components are connected using relations, e.g., hostedOn or connectsTo. For instance, Flink is hosted on Ubuntu, which is provided by OpenStack. Properties are used to provide information, such as the endpoint and credentials of OpenStack or to specify that the InfluxDB is already running. The figure also shows how required dependencies, such as the SciPy library for scientific computing, can be specified within a deployment model. Different kinds of artifacts can be attached to nodes. For installing the python runtime, a Bash script (install.sh) is provided, which, for example, could execute the command *apt-get install python* in order to fetch and install the required files. The file pred_service.py is part of the PredictionService node and represents the business logic of this component.

2.2 Use Cases & Problems

Typically, when deploying an application in a remote IT environment, required components and dependencies are resolved, downloaded, and installed on the fly during the deployment of the application in the respective environment – regardless of whether the deployment is executed manually or automatically by using technologies, such as configuration management tools (Wettinger et al., 2014). Therefore, a deployment package typically only describes the steps required to deploy an application, for example, which specific files need to be downloaded and installed, e.g., *apt-get install python* to install a python runtime, or how the components need to be configured in order to work properly. This allows to have lightweighted deployment packages, since external dependencies are only resolved and fetched during the deployment phase of the application and don't have to be added to the deployment packages in advance. Moreover, this also eases the maintenance of such deployment packages. For example, if a new version of a required component is released, in the deployment description only the reference of this component needs to be adapted in order to reflect the new version, but no files need to be downloaded and replaced in the deployment package. However, there are some use cases where this approach of resolving and downloading required external dependencies during the deployment time results in serious problems, which can prevent the successful deployment of the application.

One use case is in the field of eScience, with the problem of reusability, reproducibility, and repeatability of scientific software (Mesnard and Barba, 2016; Fehr et al., 2016). Often research software is only implemented for a specific experiment by the researcher himself and is not maintained and managed like a professionally developed software. If some time later the software shall be reused in order to reproduce the research results, the experiment needs to be executed in the same environment. This not only includes using the same version of the software itself, but also using the identical versions of dependencies, as using a different library, for example, could lead to a different result. Furthermore, it is possible that some dependencies will cease to be available over time. Thus, in such use cases, the required artifacts should be bundled together with the deployment description in order to provide a self-contained deployment package.

In the area of automotive, the resolving of external dependencies is also important. For example, if a new update should be installed on a component of a car while it is operated, there are situations in which an update is more sensible than in another. For instance, it is not useful to update the start-stop system of a car when it is located within a city with a lot of traffic lights or on a road with a traffic jam, because these are situations in which the start-stop system should work properly. Thus, a better time to update the start-stop system might be, when the car is operated on a highway outside of a city without traffic. Or regarding the car radio, it makes sense to update its software when it has no radio reception anyway, for example, when the car is inside a tunnel. However, although these are situations in which an update would be sen-

sible from a contextual point of view, there might be circumstances that prevent updates in these situations. For instance, in a tunnel or in a rural countryside without mobile Internet connection, an over-the-air update is technically not possible. Thus, the update and required dependencies should be resolved and fetched when a stable Internet connection is available and only installed when the situation allows it.

Another use case is in the field of Industry 4.0, where, e.g., analysis software should be shipped and deployed into manufacturing environments to optimize production lines or for enabling predictive maintenance (Zimmermann et al., 2017). For security reasons, these environments often have no Internet access. Thus, in such environments it is not possible to deploy software by using a technology that needs Internet access in order to download external artifacts that are required for installing the application as well as for the properly execution of the application.

To sum up, there are use cases in which no Internet access is available and thus, no dependencies can be downloaded and installed during the deployment. Also, external dependencies can cease to be available over time, not only hindering the download, but also the successful execution of the entire deployment. Therefore, in this work we present an approach for building self-contained deployment packages.

2.3 State of the Art Approaches

In the following, we want to give and overview of state-of-the-art approaches and existing technologies for the deployment of cloud and IoT applications in remote IT environments. Furthermore, we want to discuss their applicability for the three previously described use cases as well as their capabilities for creating completely self-contained deployment packages.

Of course, the simplest approach is to manually download the required artifacts and install and configure the application by hand. However, dependent of the complexity of the software, these tasks typically requires immense expertise, are time-consuming as well as error-prone, and, therefore, are not efficient if done manually (Breitenbücher et al., 2013). Another approach is to bundle the entire application together with all required dependencies into a virtual machine image or a container using container technologies such as Docker. While this resolves the problem of creating a self-contained deployment package, this approach comes with other problems. For example, if packaged as a monolithic image or container, an application may not benefit from cloud properties such as scaling (Leymann et al., 2017). Also, if the components of the application are split, for example, into single Docker containers, the management and orchestration of these containers is a new challenge and requires the use of additional technologies such as Docker Swarm or Kubernetes. However, these technologies are categorized as platform-specific deployment technologies (Wurster et al., 2019), since they are restricted regarding the use of specific platform bundles for realizing deployable components, e.g., Kubernetes only supports container images. Also, these technologies not only have some constraints regarding the orchestration and wiring of different kinds of application components (Zimmermann et al., 2018b), but also regarding the deployment and management of components on bare metal, which can be important, especially when IoT devices are involved.

Configuration management technologies, such as Chef, Ansible, or Puppet can also be used to automate the deployment of applications in remote IT environments. These kind of deployment technologies are categorized as general-purpose technologies (Wurster et al., 2019). They use scripts to define the required deployment steps in order to reach the desired state and to execute the provisioning and configuration of the application. However, since their focus is not on the creation of deployment packages that can be shipped to the target environment as a whole, but on configuration files describing the deployment steps or the desired system state, these technologies typically require an Internet connection in order to download and install the defined components and dependencies in the target environment. Another approach for deploying and managing cloud and IoT applications is the usage of provider- and technology-agnostic standards, such as TOSCA. TOSCA enables to describe the structure of an application, the dependencies, and required infrastructure resources in a portable manner. Furthermore, TOSCA allows the integration of all kinds of technologies, such as Bash scripts, Docker, or Ansible, and thus, represents a technology independent and interoperable deployment approach. Moreover, with the Cloud Service Archive (CSAR), TOSCA specifies a format, for packaging all required files for provisioning and managing the described application. However, typically these archives contain scripts in order to install external dependencies, e.g., by using the Linux package handling utility *apt-get*.

How a non-self-contained deployment package with external dependencies can be transformed into a self-contained deployment package is shown in this work. Our approach enables to use the same deployment model for creating a light-weighted deployment package and for creating a bigger, but completely self-contained deployment package. Therefore, the topology of the application can be modeled uniformly.

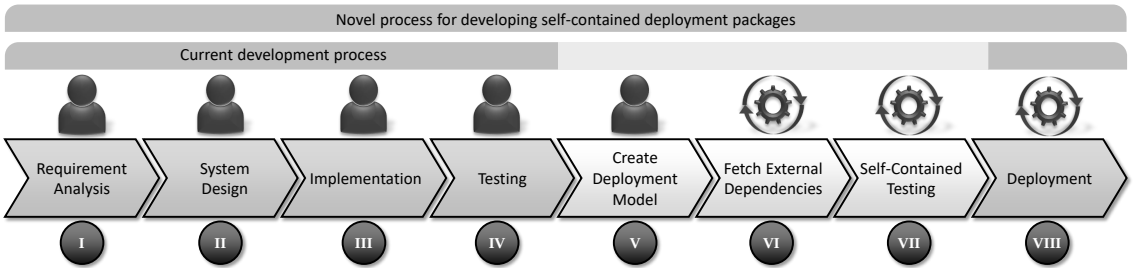

Figure 2: Our method for developing self-contained deployment packages.

3 APPROACH FOR CREATING SELF-CONTAINED DEPLOYMENT PACKAGES

In this section, we present our approach for creating self-contained deployment packages. First, we give a brief overview of our proposed approach. After that, we describe our method for systematically developing completely self-contained deployment packages, and moreover, present the system architecture of our proposed *Self-Containment Packager Framework*.

3.1 Overview of the Approach

To recap, typically deployment automation technologies have external dependencies that are downloaded and installed during the deployment time of the application. For example, the Advanced Packaging Tool (APT) is often used for resolving and installing required software components on Linux distributions, for instance, *apt-get install python*, in order to install a python runtime. Moreover, with the emergence of container technologies such as Docker or Kubernetes, also Dockerfiles describing the container to be built, are often used within deployment models in order to containerize single components or entire applications.

Therefore, the goal of our approach is to enable the automatic transformation of a non-self-contained deployment package in a self-contained deployment package, and thus, support the creation of deployment packages in a uniform way. To realize the transformation, all artifacts need to be analyzed in order to find external dependencies. Furthermore, these external dependencies need to be resolved, downloaded, and packaged into the deployment package. Moreover, references in the deployment model need to be adapted in order to reflect the made changes. Therefore, by following our method, our approach enables the developer to create both variants of deployment packages – light-weighted, but non-self-contained deployment packages as well as a fully self-contained deployment packages – without any additional effort.

3.2 Method for Developing Self-Contained Deployment Packages

In this subsection, we present our new method for systematically developing self-contained deployment packages. We explain the general advantages of our proposed method and show how typical non-self-contained deployment packages can be automatically transformed in self-contained deployment packages.

Our method to develop a self-contained deployment package is oriented on the waterfall software engineering process and extends it by additional steps. The method is illustrated in Fig. 2. The first step (I) is to capture all requirements of the system to be developed. Furthermore, the purpose and the provided functionality of the application to be packaged is defined in this step. The second step (II) is to prepare a system design based on the previously defined requirements specification. So, in this step, for example, the required components as well as the required infrastructure resources are defined. The third step (III) is the implementation of required components, for example, a service for analyzing manufacturing data. For example, this could be a .jar which needs to be deployed on some data processing framework, such as Apache Flink. Moreover, this Apache Flink component could be realized by using a Dockerfile. In the forth step (IV), the previously implemented components are tested. Here, the functionality of the components can be tested separately as unit tests as well as the entire system as a whole. The new fifth step (V) is the creation of a deployment model of the entire system. The deployment model represents the configuration and relations of the employed nodes at run-time and enables the automatically deployment of the described application. Furthermore, required management operations, e.g., to start virtual machines or install a component, are defined and implemented in this step. The sixth step (VI) is the fetching of external dependencies. Thus, in this step the single ar-

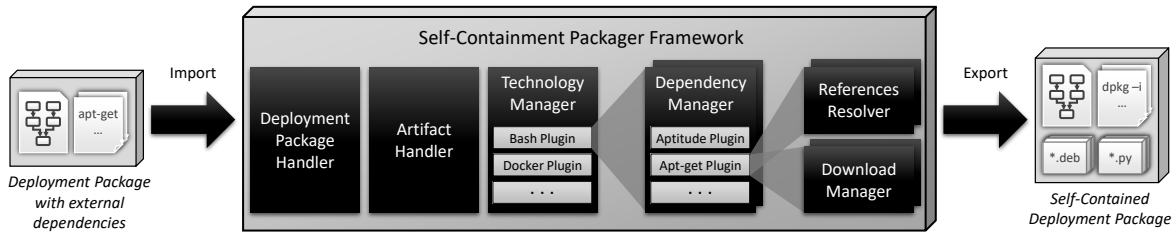

Figure 3: Simplified system architecture of the *Self-Containment Packager Framework* that enables the transformation of non-self-contained deployment packages into self-contained deployment packages.

tifacts are analyzed by a Self-Containment Packager Framework and if external dependencies are found, they are resolved and downloaded. Moreover, the downloaded files are packaged into the deployment package and the respective artifacts are adapted in order to utilize them. Since various technologies can be used to create these artifacts, this is highly dependent of the concrete implementation of each artifact. For example, if such an artifact is implemented as a Bash script using the command *apt-get install python* to install python on an Ubuntu virtual machine, this command will be replaced with *dpkg -i <*.deb-files>* in order to use the downloaded software packages for installing the python runtime. If a Dockerfile is used, first the Docker image described by this Dockerfile needs to be build, then this image must be exported as a file and put into the deployment package. Thus, in this step, a non-self-contained deployment package is transformed into a self-contained deployment package, which can be executed in an environment without Internet access. Details are presented in the following subsection. In the seventh step (VII), the transformation of the deployment package can be automatically tested. Therefore, the now self-contained deployment package is automatically executed in a local environment with limited network access. If the deployment of the described and modeled application finishes without failures, the transformation was successful. The eight step (VIII) is the automatically deployment of the packaged application in the target environment, by using a compliant deployment tool or the release of the created package into a marketplace.

From our three additionally added steps (V - VII), only step five, the creation of the deployment model, needs to be done manually. However, this task typically needs to be done manually, nevertheless, if a non-self-contained deployment package should be created or the self-contained counterpart. Steps six and seven can be automated by using a Self-Containment Packager Framework for the transformation and a compliant deployment tool for the deployment. Therefore, for the developer, no extra effort is needed for developing the self-contained variant in comparison with the non-self-contained variant.

3.3 System Architecture of the Self-Containment Packager Framework

In this subsection, we present a system architecture supporting the automatic transformation of non-self-contained deployment packages into self-contained deployment packages. Our proposed system architecture, supporting the presented method, is shown in Fig. 3. The main components of the Self-Containment Packager Framework are: (i) the *Deployment Package Handler*, (ii) the *Artifact Handler*, (iii) the *Technology Manager*, (iv) several *Dependency Manager*, (v) *References Resolver*, and (vi) *Download Manager*. All these mentioned components as well as the overall procedure to transform a deployment package with external dependencies into a self-contained deployment package are explained in the following.

Typically, deployment packages are archives, e.g., ZIP archives, containing the description of an application as well as further required executable artifacts. Thus, a deployment package with external dependencies, that should be transformed into a self-contained deployment package, first needs to be unpacked and prepared for processing the content. Moreover, at the end of the entire transformation process, when all external references are resolved, the adapted content as well as the downloaded files must be packaged again appropriately. Therefore, the *Deployment Package Handler* provides methods for unzipping and zipping of deployment packages, accessing the contained files, and maintaining the consistency of the package.

In order to be able to resolve external dependencies preventing the provisioning of an application in an environment without Internet access, all artifacts contained in a deployment package need to be checked for specified external references. Thus, the *Artifact Handler* searches the deployment package for all kinds of contained executable artifacts, such as, Bash scripts or Dockerfiles, potentially specifying external dependencies. All found artifacts are forwarded to the *Technology Manager* for further processing.

Since, the Self-Containment Packager Framework should be able to handle different technologies, e.g., configuration management tools and script languages, the framework is designed to be easily extensible, regarding its capabilities to support various languages and tools. Therefore, the *Technology Manager* component provides a plugin system for handling the technology specific plugins, supporting, for instance, Ansible playbooks, Bash scripts, or Dockerfiles. Each artifact is analyzed by a plugin responsible for this specific artifact type, checking the artifact for specified external dependencies and how they are defined. Dependent on that, the artifacts are forwarded for further processing to the responsible plugins, which are managed by the *Dependency Manager* component.

Since these mentioned technologies can support different kinds of technologies again, for example, package manager for specifying external dependencies in case of Bash scripts, each specific language plugin has its own *Dependency Manager*. To enable a high extensibility, the *Dependency Manager* provides a plugin system as well, for handling the specific plugins supporting different kinds of technologies. For example, in case of Bash scripts, typically one of the both package manager Aptitude or apt-get is used.

To finally resolve the specified external dependencies and download all required files, each Dependency Manager plugin holds its own *References Resolver* and *Download Manager*. The References Resolver is responsible for resolving the explicitly specified external dependencies and further required implicit dependencies. For example, the command *apt-get install python* will not only install python, but also some further packages required in order to install python. Thus, the References Resolver analyzes every dependency which is explicitly specified within an artifact and, if necessary, determines the dependency tree for each detected dependency. Since this step is highly technology specific, there is no generic references resolver available and this component needs to be implemented for each single technology that should be supported. For instance, in case of required software packages on an Ubuntu virtual machine, the tool apt-get provides the functionality to determine the dependency tree of a software that should be installed. Afterwards, the dependency tree is forwarded to the Download Manager, which is responsible for downloading the determined dependencies. The Download Manager component is reusable within the framework, since different deployment technologies can use the same method for specifying external dependencies. For example, the utility Wget to download files from the Web is supported by various deployment technologies, such as Ansible or Bash scripts.

4 VALIDATION & EVALUATION

In this section, we provide details about our prototypical implementation of the *Self-Containment Packager Framework*. In order to evaluate our prototype, we also compare the time required for the provisioning of both, a self-contained deployment package and a non-self-contained deployment package describing the provisioning of a LAMP stack. Moreover, we evaluate the time required for transforming a non-self-contained deployment package into a self-contained deployment package and compare the size of both.

4.1 Validation

For validating the practical feasibility of our approach, we use the deployment modeling language TOSCA for the following reasons: (i) it provides a vendor- and technology-agnostic modeling language, (ii) it is ontologically extensible (Bergmayr et al., 2018), and (iii) it is fully compliant with the Essential Deployment Metamodel (EDMM) (Wurster et al., 2019). In the course of a systematic review, the essential parts supported by declarative deployment automation technologies were derived and showed how they can be mapped to EDMM. Therefore, EDMM provides a technology-independent baseline for deployment automation research and a common understanding of declarative deployment models (Wurster et al., 2019). For testing the resulting deployment packages of our prototype, we use the OpenTOSCA ecosystem (Breitenbücher et al., 2016), a standard-compliant open-source toolchain, used for provisioning and managing of TOSCA-based deployment packages, called *Cloud Service Archives (CSARs)*[1]. In particular we use *Winery*[2] (Kopp et al., 2013), an application enabling to model TOSCA-based topologies graphically and *OpenTOSCA Container*[3] for deploying the modeled and packaged applications.

Our TOSCA-based prototype[4] of the Self-Containment Packager Framework is implemented using Java version 1.8. It is capable of resolving external dependencies specified using Apt-get and Aptitude for Bash scripts, Apt-get for Ansible playbooks, as well as the creation of Docker images containing all dependencies based on a Dockerfile. For example, in order to find artifacts implemented as a script, the framework iterates over all artifacts contained in a CSAR and checks if their file endings matches

[1] Additional information about the ecosystem and documentation can be found at http://opentosca.org

[2] https://github.com/OpenTOSCA/winery

[3] https://github.com/OpenTOSCA/container

[4] https://github.com/zimmerml/TOSCA_packager

"*.sh" or "*.bash". Next, the found scripts will be searched for defined "apt-get install" commands, since these commands indicate that the CSAR is non-self-contained and has specified external dependencies. If so, the respective file will be forwarded to the "apt-get" plugin for further processing. There, the "apt-get install" command is analyzed and checked which installation files are required for installing the defined component. These installation files, for example, Debian packages (*.deb), are downloaded and packaged into the CSAR. However, since each dependency can have its own dependencies again, defined external dependencies are resolved recursively. For creating Docker images based on Dockerfiles, our prototype utilizes the Docker CLI. Therefore, Docker needs to be installed on the same machine as our prototype is used on. After all external dependencies are resolved, downloaded, and packaged into the CSAR, this now resulting self-contained CSAR can be tested automatically (Wurster et al., 2018) by using it as input of the OpenTOSCA Container in order to deploy it into an environment without Internet access.

4.2 Evaluation

For evaluating our prototype we created a CSAR for provisioning a LAMP stack on our in-house hypervisor vSphere (*CSAR A*). The corresponding TOSCA-based deployment model is illustrated in Fig. 4. The CSAR contains two artifacts implemented as Web Services ("HypMngmt.war" and "VMMngmt.war") for creating a virtual machine using the web service API of the hypervisor vSphere as well as to upload files and run commands on this virtual machine using SFTP and SSH protocol. Furthermore, the CSAR contains three artifacts implemented as Bash scripts (all named "Install.sh") for installing Apache, MySQL, and PHP. Technically, the "transferFile" operation of the Ubuntu node is used in order to upload these three scripts to the virtual machine. Furthermore, the "runScript" operation is used in order to invoke them directly on the virtual machine. Internally, the scripts are using the "apt-get install" command in order to install Apache, MySQL, and PHP. Thus, the described CSAR is non-self-contained, since the components and their dependencies need to be downloaded first on the virtual machine in order to be installed. For evaluating our framework with a second CSAR, we additionally created a CSAR with a contained Dockerfile (*CSAR B*). As Dockerfile, we used an image also realizing a LAMP stack[5]. How the transformation effects the provisioning time as well as the size of the CSARs, is presented in Table 1.

[5]https://hub.docker.com/r/tutum/lamp

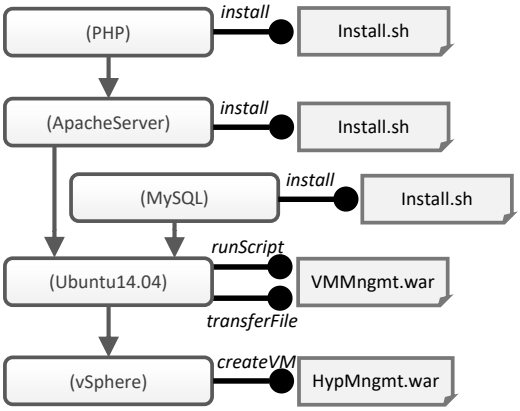

Figure 4: LAMP stack modeled as declarative deployment model used for evaluating our approach.

The transformation of *CSAR A* into a self-contained deployment package using our prototype took in average 55 seconds. Basically, this is the time required for downloading the components and its dependencies as well as to adapt the CSAR accordingly. Furthermore, since all required installation files and dependencies are added to the CSAR, the overall size of the CSAR grew from 28 MB to 143 MB. The provisioning of the LAMP stack using the non-self-contained CSAR took in average 164 seconds, while the provisioning using the self-contained CSAR took in average 160 seconds. In average 51 seconds were required for creating the virtual machine. Thus, the pure installation took 113 seconds in case of the non-self-contained CSAR and 109 seconds in case of the self-contained CSAR. While for the non-self-contained CSAR all installation files and dependencies need to be downloaded from the Internet, for the self-contained CSAR, these files need to be uploaded to the virtual machine. Thus, the overall provisioning time remained almost equal for both CSAR variants.

For *CSAR B*, the size grew from 72 MB to 212 MB and the transformation took 68 seconds in average. In this test, the virtual machine was already running and a Docker engine was installed on it. The provisioning of the CSAR containing the Dockerfile took 48 seconds in average, while the self-contained CSAR containing the docker image took 60 seconds in average.

Table 1: Evaluation results.

CSAR	Size	Transformation Time	Provisioning Time
CSAR A^{n}	28 MB	55 s	113 s
CSAR A^{s}	143 MB		109 s
CSAR B^{n}	72 MB	68 s	48 s
CSAR B^{s}	212 MB		60 s

^{n} non-self-contained; ^{s} self-contained

5 RELATED WORK

Discovery and management of software dependencies have been in the focus of numerous research papers. One particular direction we are interested in is how to achieve the portability of software by analyzing and materializing its dependencies. Thus, in this section, we complete our discussion about related work, which we already discussed partially in Section 2.

Keller et al. (Keller et al., 2000) present a classification of dependencies and propose a technique of how the management information can be collected for IP-based services and applications. More specifically, the authors describe the derivation of applications characteristics and interdependencies by analyzing the system information repositories, such as Windows Registry or Linux Red Hat Package Manager. The main idea is to make use of the information available in system repositories by constructing functional service dependency models which can be used for automatic dependency generation. In the context of self-contained deployment packages, the idea to leverage from system information repositories might be useful in case the archive's constituents installation relies only on the usage of operating systems' package managers. However, the package manager installation command is not the only possible way to specify how a required deployment artifact can be deployed.

Mugler et al. (Mugler et al., 2005) discuss the usage of Open Source Cluster Application Resources (OSCAR) toolkit and its meta-packaging system for distribution and installation of software for clusters. So-called OSCAR packages might consist of binary software, e.g., RPM packages, plus additional configurations, tests, and documentation. The authors distinguish between native package systems like RPM and meta-packaging system which operates on a higher level to avoid coupling with a certain Linux distribution by including native package system files into the OSCAR packages. In the context of our work, the idea to include various representations of a particular software artifact is useful in order to support a broad spectrum of target platforms. However, this particular approach is not suitable for automatically transforming non-self-contained deployment packages into completely self-contained deployment packages, since the proposed OSCAR packages need to be created in advance as well as manually.

Guo and Engler (Guo and Engler, 2011) introduce a system called CDE which provides means to achieve software portability by packaging the code, data, and environment which are necessary for execution on x86 Linux machines. This approach is suitable for the cases where an application is available for execution and is bound to Linux and a x86 architecture. While this approach is based on building virtual machine images, in contrast, our approach enables the creation of technology-agnostic deployment packages, supporting, e.g., cloud-native applications.

Etchevers et al. (Etchevers et al., 2011) present a process for modeling and deploying distributed applications in the cloud. The process starts with modeling a target application using an extended version of Open Virtualization Format, a standard which provides means to describe software based on virtual systems. Again, this approach is mainly based on virtual machines. However, bundling applications into a virtual machine image is not enough for creating real cloud-native applications (Leymann et al., 2016).

Fischer et al. (Fischer et al., 2012) describe the process of complex application stacks configuration, installation, and management using a system called Engage. The main idea is similar to configuration management systems like Chef or Puppet, with additional enhancements. Based on the provided partial installation specifications, Engage can generate installation specifications which then will be used for the deployment process. However, required packages are downloaded during the deployment process, whereas, our goal is to materialize external dependencies already beforehand of the deployment time.

Meng and Thain (Meng and Thain, 2015) demonstrate how sophisticated execution environments can be specified and materialized by using a tool called Umbrella. The goal is to let user run a task via Umbrella by providing all the required information including an execution environment specification. After the task submission, Umbrella makes a decision regarding an execution engine, e.g., Docker or Amazon EC2 suitable for running the task. Software dependencies necessary for task's execution are obtained by the system. However, again this approach utilizes virtual machine images and containers in order to create self-contained execution environments, and ignores other deployment technologies, for example, Ansible or Chef. Furthermore, required files are downloaded during the deployment time and not in advance.

Di Cosmo et al. (Di Cosmo et al., 2015) introduce a toolchain called Aeolus Blender, which provides means to automatically deploy cloud applications in OpenStack. From the user's perspective, the process starts with providing initial application-related information which triggers the computation of the complete application's installation architecture. In case some configuration information is missing the user will be asked to provide it. Afterwards, the resulting configuration is used for the deployment of the application in OpenStack. However, Blender is limited to

OpenStack environments, and moreover, dependencies are resolved during the deployment process, thus, self-contained deployment packages are not created.

Further related research work in the area of TOSCA is available from Brogi et al. (Brogi et al., 2018a; Brogi et al., 2018b) and Kehrer and Blochinger (Kehrer and Blochinger, 2018). In their approaches, they try to synergically combine TOSCA and Docker together, in order to enable automated deployment and orchestration support for multi-component applications consisting of container artifacts. However, while they are using the CSAR format, their proposed approaches do not consider the creation of self-contained deployment packages, in contrast they use repositories, such as Docker Hub or GitHub, to retrieve artifacts when they are required.

6 CONCLUSION

In this paper, we presented an approach for transforming non-self-contained deployment packages into self-contained deployment packages. With our approach, we enable to automatically deploy applications into an environment without Internet access, like for example, into manufacturing environments which for data security and privacy reasons often have no Internet connection. Furthermore, our approach enables the preserving of required software components and dependencies, for instance, for research software used in eScience. Therefore, we presented a system architecture of a Self-Containment Packager Framework enabling this transformation by searching for artifacts specifying external dependencies. Furthermore, these dependencies are resolved, downloaded, as well as packaged into the final deployment package. Moreover, we introduced a method describing the systematically developing of such self-contained deployment packages. The presented approach is validated by a prototypical TOSCA-based implementation and evaluated, by comparing the time required for provisioning two exemplary non-self-contained CSARs with its self-contained counterparts. Moreover, we compared the size of the different TOSCA archive variants.

We plan to extend our approach to also cope with other artifacts that can be contained in a deployment package, e.g., process models, i.e., workflows, describing the steps to manage an application, e.g., to scale or update it. For instance, instead of downloading external dependencies by artifacts, these process models could be used for that. Therefore, in future work we also want to check these process models, for example, for outgoing requests to external resources.

ACKNOWLEDGEMENTS

This work was partially funded by the BMWi project *IC4F* (01MA17008G), the DFG project *SustainLife* (641730), and the European Union's Horizon 2020 research and innovation project RADON (825040).

REFERENCES

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.

Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., and Wettinger, J. (2013). Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. In *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, pages 130–148. Springer.

Breitenbücher, U., Endres, C., Képes, K., Kopp, O., Leymann, F., Wagner, S., Wettinger, J., and Zimmermann, M. (2016). The OpenTOSCA Ecosystem - Concepts & Tools. *European Space project on Smart Systems, Big Data, Future Internet -Towards Serving the Grand Societal Challenges -Volume 1: EPS Rome*, pages 112–130.

Brogi, A., Neri, D., Rinaldi, L., and Soldani, J. (2018a). Orchestrating incomplete tosca applications with docker. *Science of Computer Programming*, 166:194–213.

Brogi, A., Rinaldi, L., and Soldani, J. (2018b). TosKer: A synergy between TOSCA and Docker for orchestrating multicomponent applications. *Software: Practice and Experience*, 48(11):2061–2079.

Di Cosmo, R., Eiche, A., Mauro, J., Zacchiroli, S., Zavattaro, G., and Zwolakowski, J. (2015). Automatic Deployment of Services in the Cloud with Aeolus Blender. In *Service-Oriented Computing*. Springer.

Eilam, T., Kalantar, M., Konstantinou, A., Pacifici, G., Pershing, J., and Agrawal, A. (2006). Managing the Configuration Complexity of Distributed Applications in Internet Data Centers. *Communications Magazine*, 44(3):166–177.

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*, pages 22–27. Xpert Publishing Services.

Etchevers, X., Coupaye, T., Boyer, F., and De Palma, N. (2011). Self-configuration of distributed applications in the cloud. In *2011 IEEE International Conference on Cloud Computing*, pages 668–675. IEEE.

Fehr, J., Heiland, J., Himpe, C., and Saak, J. (2016). Best Practices for Replicability, Reproducibility and Reusability of Computer-Based Experiments Exemplified by Model Reduction Software. *AIMS Mathematics*, 1(3):261–281.

Fischer, J., Majumdar, R., and Esmaeilsabzali, S. (2012). Engage: A Deployment Management System. In *ACM SIGPLAN Notices*, pages 263–274. ACM.

Guo, P. J. and Engler, D. R. (2011). CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *Proceedings of the 2011 USENIX Annual Technical Conference*, page 247–252. USENIX Association.

Guo, Y., Hu, X., Hu, B., Cheng, J., Zhou, M., and Kwok, R. Y. K. (2017). Mobile Cyber Physical Systems: Current Challenges and Future Networking Applications. *IEEE Access*, 6:12360–12368.

Haque, S. A., Aziz, S. M., and Rahman, M. (2014). Review of Cyber-Physical System in Healthcare. *International Journal of Distributed Sensor Networks*, 10(4):217415.

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*, pages 57–68. USENIX.

Iosup, A., Ostermann, S., Yigitbasi, M. N., Prodan, R., Fahringer, T., and Epema, D. (2011). Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945.

Kehrer, S. and Blochinger, W. (2018). TOSCA-based Container Orchestration on Mesos. *Computer Science - Research and Development*, 33(3):305–316.

Keller, A., Blumenthal, U., and Kar, G. (2000). Classification and Computation of Dependencies for Distributed Management. In *Proceedings of the Fifth IEEE Symposium on Computers and Communications*, pages 78–83. IEEE.

Kopp, O., Binz, T., Breitenbücher, U., and Leymann, F. (2013). Winery – A Modeling Tool for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*, pages 700–704. Springer.

Leymann, F. (2009). Cloud Computing: The Next Revolution in IT. In *Proceedings of the 52th Photogrammetric Week*, pages 3–12. Wichmann Verlag.

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

Leymann, F., Fehling, C., Wagner, S., and Wettinger, J. (2016). Native Cloud Applications: Why Virtual Machines, Images and Containers Miss the Point! In *Proceedings of the 6th International Conference on Cloud Computing and Service Science*, pages 7–15. SciTePress.

Mahmud, R., Kotagiri, R., and Buyya, R. (2018). *Fog Computing: A Taxonomy, Survey and Future Directions*, pages 103–130. Springer.

Meng, H. and Thain, D. (2015). Umbrella: A Portable Environment Creator for Reproducible Computing on Clusters, Clouds, and Grids. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, pages 23–30. ACM.

Mesnard, O. and Barba, L. A. (2016). Reproducible and Replicable Computational Fluid Dynamics: It's Harder Than You Think. *Computing in Science Engineering*, 19(4):44–55.

Mugler, J., Naughton, T., and Scott, S. L. (2005). OSCAR Meta-Package System. In *19th International Symposium on High Performance Computing Systems and Applications*, pages 353–360. IEEE.

OASIS (2013a). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

OASIS (2013b). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

Shrouf, F. and Miragliotta, G. (2015). Energy management based on Internet of Things: Practices and framework for adoption in production management. *Journal of Cleaner Production*, 100:235–246.

Tsuchiya, A., Fraile, F., Koshijima, I., Órtiz, A., and Poler, R. (2018). Software defined networking firewall for industry 4.0 manufacturing systems. *Journal of Industrial Engineering and Management*, 11(2):318–333.

Wettinger, J., Breitenbücher, U., and Leymann, F. (2014). Compensation-based vs. Convergent Deployment Automation for Services Operated in the Cloud. In *Proceedings of the 12th International Conference on Service-Oriented Computing*, pages 336–350. Springer.

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

Wurster, M., Kopp, U. B. O., and Leymann, F. (2018). Modeling and Automated Execution of Application Deployment Tests. In *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference*, pages 171–180. IEEE Computer Society.

Zimmermann, M., Breitenbücher, U., Falkenthal, M., Leymann, F., and Saatkamp, K. (2017). Standards-based Function Shipping – How to use TOSCA for Shipping and Executing Data Analytics Software in Remote Manufacturing Environments. In *Proceedings of the 21st International Enterprise Distributed Object Computing Conference*, pages 50–60. IEEE Computer Society.

Zimmermann, M., Breitenbücher, U., Guth, J., Hermann, S., Leymann, F., and Saatkamp, K. (2018a). Towards Deployable Research Object Archives Based on TOSCA. In *Papers from the 12th Advanced Summer School on Service-Oriented Computing*, pages 31–42. IBM Research Division.

Zimmermann, M., Breitenbücher, U., and Leymann, F. (2018b). A Method and Programming Model for Developing Interacting Cloud Applications Based on the TOSCA Standard. In *Enterprise Information Systems*, volume 321 of *Lecture Notes in Business Information Processing*, pages 265–290. Springer.