2015 IEEE International Conference on Mobile Services

Smart Fabric – An Infrastructure-Agnostic Artifact Topology Deployment Framework

Johannes M. Schleicher, Michael Vögler, Christian Inzinger and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, 1040 Vienna, Austria
{lastname}@dsg.tuwien.ac.at

***Abstract*—The cloud computing paradigm enables the development of applications that can elastically react to changes in their environment by autonomously provisioning and releasing infrastructure resources. However, current applications need to be specifically tailored to a concrete cloud provider infrastructure, leading to vendor lock-in. Migrating applications to the cloud or between cloud providers is challenging due to differences in deployment directives, available services, and programming interfaces. Existing infrastructure as code approaches closely tie application artifacts to their deployment directives and do not allow for a clear separation of application artifacts from deployment infrastructure. In this paper, we present Smart Fabric, a methodology and accompanying toolset for infrastructure-agnostic deployment of application artifact topologies based on a constraint-based, declarative specification of the required deployment infrastructure. Our framework allows for seamless migration of application topologies between deployment targets and enables independent, parallel evolution of both, applications and underlying infrastructure. We discuss the feasibility of the proposed methodology and prototype implementation using representative applications from the Internet of Things and smart city domains.**

I. Introduction

The recent emergence of the cloud computing paradigm [1] allows stakeholders to leverage a utility-oriented, on-demand approach to create applications that elastically respond to changes in request load, do not depend on dedicated operations teams on site, and can be managed and evolved without upfront infrastructure investments. Despite the apparent benefits, companies and government agencies are still hesitant to migrate business-critical applications to the cloud [2], mainly due to concerns related to vendor lock-in [3], [4] and service availability. While cloud services are designed to provide abstractions that shield stakeholders from the underlying physical infrastructure, applications must nevertheless be specifically tailored to concrete cloud providers to make use of the offered services.

This strong dependence on specific cloud providers is problematic for several reasons. While cloud providers now usually offer service level agreements[1] (SLAs) that provide certain guarantees for service availability to customers, the terms governing customers' use of the offered services can still be unilaterally changed by providers. Changes in offered services or pricing can occur at any time (e.g., retiring offerings, changing pricing structures, introducing new offerings that better suit customers' requirements) and customers may need to migrate their applications to different services or providers as a result. Current approaches for software engineering and lifecycle management do not sufficiently support the independent evolution of infrastructure alongside an application. While approaches like DevOps [5] and Infrastructure as Code [6] simplify application provisioning by integrating deployment directives into the development process, infrastructure evolution is not currently considered.

We argue that deployment infrastructure and application development must be clearly separated to allow for seamless, independent evolution of application components as well as the underlying infrastructure. In this paper, we present Smart Fabric, a methodology and toolset for infrastructure-agnostic deployment of artifact topologies based on a declarative, constraint-based specification of the required deployment infrastructure. Smart Fabric extends the previously introduced MADCAT [7] methodology with an abstraction layer that cleanly separates application artifacts from the concrete deployment infrastructure, along with mechanisms to seamlessly migrate application topologies between deployment targets. We illustrate the feasibility of the proposed framework and prototype using representative applications from the Smart City and Internet of Things (IoT) domain.

The remainder of this paper is structured as follows. In Section II we discuss the foundational system model underlying our approach. Section III presents the Smart Fabric framework for infrastructure-agnostic deployment of artifact topologies, followed by a detailed discussion and validation of the framework properties in Section IV. Related research is discussed in Section V and we conclude the paper in Section VI along with an outlook for ongoing and future research.

II. System Model

In order to enable an infrastructure independent artifact deployment framework we introduce an abstraction to model and describe the relevant entities in our domain. As foundation for the abstraction we use the MADCAT [7] methodology, as it introduces several abstract concepts suitable for describing application topologies independent of their deployment targets. Specifically, these are *Technical Units (TU)* to describe applications and their components, as well as *Deployment Units (DU)* to describe how to deploy them on cloud infrastructure. This however is not sufficient for complex, large-scale applications, such as in the Smart City domain, since we need to cover a wider array of different infrastructures (e.g., legacy systems on premises, edge devices, etc.) and deployment types (e.g., scaling across infrastructure boundaries).

[1] e.g., https://cloud.google.com/compute/sla, http://aws.amazon.com/ec2/sla/, http://azure.microsoft.com/en-us/support/legal/sla/

978-1-4673-7284-8/15 $31.00 © 2015 IEEE
DOI 10.1109/MS.2015.52

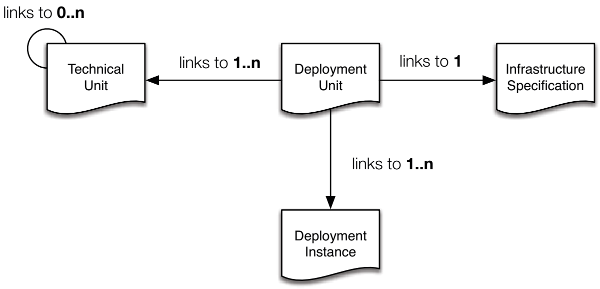

Fig. 1: Relations between TU, DU, IS and DI

To address this limitation we extend the existing concepts with *Infrastructure Specifications (IS)* to capture infrastructure heterogeneity and *Deployment Instances (DI)* to represent the current state of an application deployment topology during runtime. Additionally, we provide an implementation of the abstract concepts along with the proposed extensions of the MADCAT methodology for the Smart Fabric framework. We choose JSON-LD[2] as data format for our concept description as a simple, both human and machine readable, pragmatic, and extensible representation that also allows us to interlink the relevant concepts with each other. Fig. 1 outlines the relations of the newly introduced *IS* and *DI* to MADCAT concepts *TU* and *DU*. In the following, we discuss the introduced concepts in more detail.

A. Technical Unit

TUs describe applications as well as application components and focus on the technical aspects of these artifacts. Listing 1 shows an example of such a *TU* for a citizen information system based on the Ruby on Rails framework. As JSON-LD document, a *TU* can start with a `context` to set up common namespaces. This is followed by a `type` and `name` to identify the unit. The `artifact-uri` points to the necessary resource to build and execute the artifact, which in turn is described in the `build` and `execution` sections. Both sections are composed of step descriptions as a flexible and extensible way of describing build and execution processes that do not depend on any specific technology. Each `step` is numbered to provide ordering, specifies a `tool` mandatory to perform the step (later used in dependency resolution), as well as a command (`cmd`) to be executed. The `verification` section follows the same step format and serves to verify a successful build, deployment and execution of an artifact. By default, verification steps follow the UNIX philosophy, considering commands that exit with a result code of 0 to be successful, whereas other result codes are interpreted as errors. It further augments the previously introduced step elements with an `expected-results` element that allows each step to specify an expected result in order to verify the successful execution. This allows the verification to be as flexible as possible, covering traditional integration tests, custom scripts, or service invocations. Additionally, *TUs* provide a `configuration` element with an extensible key-value format that allows to provide additional configuration information. Furthermore, a *TU* contains a `dependency` section to specify dependent artifacts (such as other application components or required data stores), as well as a `metainformation` section to describe additional relevant aspects like used `framework`, required `runtime`, as well as basic system requirements, such as minimum amount of `memory` or desired `cpu` capacity. Additionally, *TUs* as well as all other elements of the system model allow the use of variables that are evaluated by the Smart Fabric framework. These variables are designated with the '`@`' prefix.

Listing 1: Technical Unit - Structure

```
{
  "@context": "http://smartfabric.dsg.tuwien.ac.at",
  "@type": "TechnicalUnit",
  "name": "CitizenInformationSystem",
  "artifact-uri": "...",
  "language": "ruby",
  "build": {
    "assembly": "/citizeninformationsystem",
    "steps":[{"step":1,"tool":"bundler","cmd":"
        bundle install"},{"step":2,"tool":"rake","
        cmd":"rake db:migrate"}, {"step":3,"tool":"
        rake","cmd":"rake db:seed"}]
  },
  "execute":[{"step":1,"tool":"rails",
    "cmd":"rails s"}],
  "verification": {
    "steps":[{"step":1,"tool":"curl","cmd":
      "curl -i @destination_url/status",
      "expected-result":"HTTP 200 OK"}]
  }
  "metainformation": {
    "type": "standalone",
    "framework": "Rails 4.0",
    "runtime":"ruby 2.0, rails 4.0"
  },
  "dependencies": {
    "datastore": {
      "type":"relational",
      "interface": "sql"
    }
  }
}
```

B. Infrastructure Specification

ISs are used to describe the capabilities of different infrastructure resources. This ranges from legacy systems that are running on premises on cloud infrastructures including Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) providers. The *IS* contains a `name` and `version` identifying specific infrastructures with the support for different versions of infrastructure stacks as well as a `kind` to discriminate between infrastructure concepts like bare metal systems, IaaS, and PaaS. This is followed by a `server` entry, which is used to describe the computing capabilities of an infrastructure. For instance, in the case of a classical server system this section contains the bare metal specifications of the machine itself, in the case of Amazon the available compute instances and their respective properties. Processing capabilities are specified using the `compute_units` array, where each entry specifies a `name` to identify the entity as well as a `capacity` element. Similarly, other available services, such as `storage`, `network`, or `databases` can be specified with each of them

[2] http://json-ld.org/

including name, type, version, interface, as well as the previously introduced step notation to describe how to use or access them. Listing 2 shows an example IS for an excerpt of the Amazon AWS stack.

Listing 2: Infrastructure Specification - Structure

```
{
  "@context": "http://smartfabric.dsg.tuwien.ac.at",
  "@type": "InfrastructureSpecification",
  "name": "Amazon AWS",
  "version": "1.0",
  "kind":"cloudprovider",
  "server": {
      "metainformation":{
        "cpu":"Intel Xeon E5"
      },
      "compute_units":[{"name":"t2.micro","capacity"
          :{"cpu":"1","memory":"1 GB","storage":"
          @EBS"}, ..., {"name":"m3.2xlarge","
          capacity":{"cpu":"8","memory":"15 GB","
          storage":"80 GB"}}]
  },
  "storage": {...},
  "network": {...},
  "databases":[{"name": "RDS db.m3.medium","type": "
      relational","interface": "sql/mysql"},...],
  "services":[...]
}
```

C. Deployment Unit

DUs provide a mean to describe how to deploy a *TU* on a specific *IS*. They link one or more *TUs* to exactly one specific *IS* by referencing their names. Additionally they allow to define constraints that need to be met in terms of hardware and software. Hardware constraints, for example, cover minimal machine requirements in terms of CPU and Memory, or, in the case of IaaS or PaaS providers, the minimal provided compute units. Software constraints allow to specify a priori requirements on the *IS* in order to be able to actually deploy the *TU* on it. This includes programming languages, as well as runtime environments or framework requirements. The last element of the *DU* is the previously introduced flexible step definition that outlines the steps that are necessary to deploy the set of *TUs* to the specified *IS*.

Listing 4 shows an example of such a *DU* for the deployment of the *TU* Citizen Information System on the *IS* dedicatedserver.

Listing 3: Deployment Unit - Structure

```
{
  "@context": "http://smartfabric.dsg.tuwien.ac.at",
  "@type": "DeploymentUnit",
  "name": "CitizenInformationSystem/DedicatedServer"
      ,

  "technicalUnits":[{
    "name":"citizeninformationsystem",
    "id":"citizeninformationsystem.tu.json"}],
  "infrastructureSpecification":{
    "name":"dedicatedServer"
  },

  "constraints": [{"hardware":[{"memory": "4GB"}]}],
  "steps": [{"step": 1,"tool": "git","cmd": "git
      clone @destination/@citizeninformationsystem.
      artifcat-uri"},{"step": 2,     "tool": "bash",
      "cmd": "cd @destination"},{"step": 3,"cmd": "
      @citizeninformationsystem.@build"},{"step": 4,
      "cmd":"@citizeninformationsystem.@execute"},]
}
```

D. Deployment Instance

A *DI* represents a specific deployment of a *TU* to an *IS* based on a *DU* taking into account all defined hardware and software constraints. There can be multiple *DIs* for a *DU* representing different specific deployments for example covering development, staging, or production deployments of an application on specific infrastructure. A *DI* specifies context, type, and name, as well as a deploymentUnit to reference the corresponding DU. This is followed by a machine element that stores data about the specific machine or machines that are currently used to execute the deployment. The application element that contains runtime information about the *TU* stores name and version of the application component, as well as an environment element that contains relevant environment information resolved by the framework. Finally, the global element allows to store additional information about the specific deployment in an open key value format that can later be used by the framework components to support deployment decisions.

Listing 4: Deployment Instance - Structure

```
{
  "@context": "http://smartfabric.dsg.tuwien.ac.at",
  "@type": "DeploymentInstance",
  "name": "....",
  "deploymentUnit":"CitizenInformationSystem/
      DedicatedServer",
  "machine": {"id":, ....},
  "application":{"name":"...","environment":[{"key":
      "..."}]},
  "global":{...}
}
```

For further information regarding the elements of a *TU*, *DU*, *DI* and *IS*, we provide detailed example representation of all of them online[3].

III. The Smart Fabric Framework

In this section we introduce the Smart Fabric framework for infrastructure-agnostic artifact topology deployment that implements the system model presented above. We start with a framework overview, followed by fundamental framework rationales, and conclude with a detailed description of all framework elements.

A. Framework Rationales

The Smart Fabric framework follows the microservice [8] architecture paradigm. An overview of the main components is shown in Fig. 2. The framework is logically organized into three main facets which group areas of responsibility. Each of these facets is composed of multiple components where each of these components is a microservice. The components in the *Analyzer* and *Handler Facet* are managed as self-assembling components[4] following a functional approach based on the

[3] http://www.github.com/jomis/smartfabric

[4] http://techblog.netflix.com/2014/06/building-netflix-playback-with-self.html

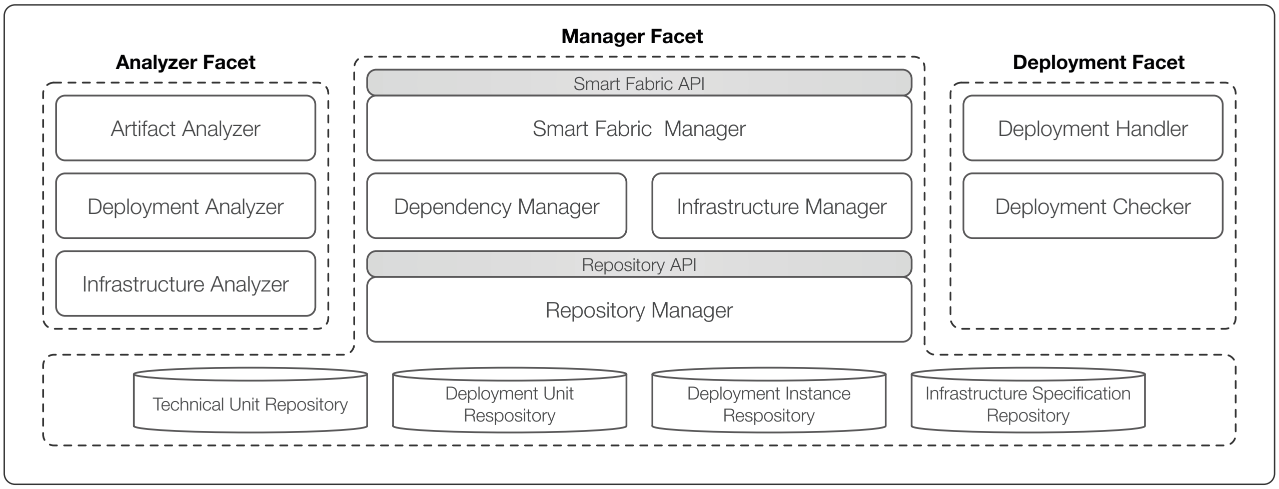

Fig. 2: Smart Fabric Framework Overview

Command Pattern [9]. In this approach each component consists of multiple *processors*, where each *processor* is able to accept multiple inputs and produces exactly one output, resembling a classic functional approach, as illustrated in Fig. 3. This allows for a clean separation of concerns into distinct functional steps that can be as specific as necessary in order to decompose complex problems into manageable units. Each of these *processors* in turn announces which inputs it requires, as well as which output it produces. This enables a straightforward auto assembly approach, where connecting previous outputs to desired inputs leads to an automatically assembled complex system consisting of simple manageable *processors*. It also eliminates the necessity of complex composition and organization mechanisms enabling dynamic and elastic composition of desired functionality, where processors can be added on demand and at runtime.

The second foundational framework rationale is that the components follow the principle of *Confidence Elasticity*, which means that each component follows a confidence-based adaptation model. If a component or processor produces a result, it augments this result with confidence value ($c \in \mathbb{R}, 0 \leq c \leq 1$), with 0 representing no certainty and 1 representing absolute certainty about the produced result. This convention allows the framework to configure certain confidence intervals to augment the auto assembly mechanism. These confidence intervals are provided as configuration elements for the framework. If these confidence thresholds are not met, the framework follows an escalation model to find the next component or processor that is able to provide results with higher confidence until it reaches the point where human interaction is necessary to produce a satisfactory result. This mechanism is outlined in Fig. 4 illustrating the process by trying to determine if an artifact is a Ruby on Rails application or not. If it is not possible to determine this within the specified confidence interval by utilizing fully automated configuration analyses, the framework escalates until a result is produced, ultimately consulting human experts to determine the nature of the artifact.

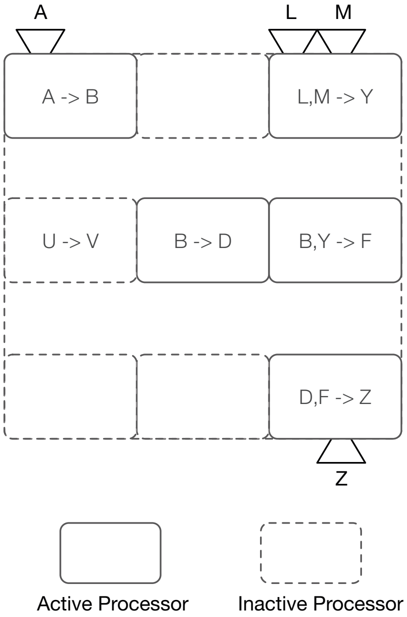

Fig. 3: Example of auto assembling processors within a component. A,L and M are initial inputs for the component, Z the final output. Each processor is active and produces an output if the expected inputs are available (e.g. L,M produces Y)

B. Smart Fabric Manager

In order to initiate a concrete deployment of an artifact on an infrastructure a user invokes the *Smart Fabric API*

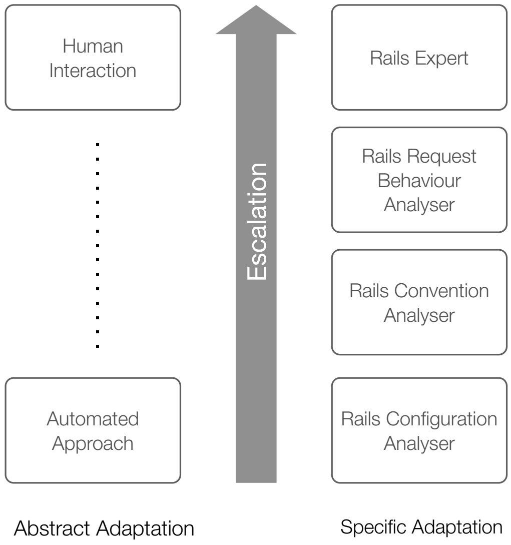

Fig. 4: Confidence Adaptation Model Escalation

with the following parameters: (i) names of *TUs* to be deployed, (ii) name of *IS* they should be deployed to, and (iii) any additional artifacts necessary that cannot be accessed by the framework itself (e.g., private repositories or external executables). The *Smart Fabric Manager* first tries to find the specified *TU* and *IS* by querying the *Repository Manager*. If both are found it hands the *TUs* and *IS* over to the *Dependency Manager*, which in turn resolves all dependencies between *TUs* and provides the corresponding *DUs*. All elements of the system model are then forwarded to the *Infrastructure Manager* that verifies if all constraints can be satisfied in order to deploy the *TUs* on the *IS* according to a *DU* via a *DI*. If this is the case the *Infrastructure Manager* produces an augmented *DI*. The *DU* and *DI* are then forwarded to the *Deployment Handler*, which actually deploys the *TU* on the *IS* using the contained deployment directives. After successful deployment it updates the *DI* and returns it to the *Smart Fabric Manager* that in turn persists it using the *Repository Manager*.

If in this process *TUs*, *DUs* or *ISs* cannot be found the *Smart Fabric Manager* utilizes the components in the *Analyzer Facet* to derive or generate them.

C. Artifact Analyzer

The task of the *Artifact Analyzer* is to generate a *TU* based on a provided artifact. It is invoked by the *Smart Fabric Manager* if no *TU* can be found for a given artifact. To accomplish this it relies on an open and extendable set of processors which follow the previously discussed framework rationales to analyze provided artifacts. Possible processors in this component are: (i) Configuration Processors that try to select a *TU* based on certain configuration files of the artifacts, (ii) Similarity Processors that try to select a *TU* based on similarities in the *TUs* values by performing actions like collaborative filtering, (iii) Convention Processors that try to derive a *TU* based on certain conventions an artifact follows (e.g., folder or file naming conventions), (iv) Behavior Processors that try to select a *TU* based on the behavior of an artifact (e.g., deriving a *TU* by sending specific requests and analyzing the responses), and (v) Human Provided Processors, which are human experts that manually create a *TU*.

D. Deployment Analyzer

The *Deployment Analyzer* generates a *DU* based on provided *TU* and *IS* if no suitable *DU* could be found by the *Smart Fabric Manager*. This component, like the *Artifact Analyzer* and the *Infrastructure Analyzer*, relies on an open and extensible set of processors. Possible processors are: (i) Similarity Processors that try to select a *DU* based on similar *DUs* that have been used for similar *TU* and *IS* configurations, (ii) Convention Processors that try to derive a *TU* based on certain conventions an infrastructure follows (e.g., to deploy to a PaaS like Heroku *DUs* for specific *TUs* follow the same conventions), and (iii) Human Provided Processors, which are human experts that manually create a *DU*.

E. Infrastructure Analyzer

The role of the *Infrastructure Analyzer* is to generate a *IS* if no suitable specification can be found in the repository. Possible processors are: (i) Documentation Processors that try to generate an *IS* based on the accessible documentation of the infrastructure. This can range from querying machine readable linked data (e.g., service registries) about the infrastructure, to analysis of electronic documentation using machine learning techniques, (ii) Behavior Processors that try to derive an *IS* based on externally observable aspects of an infrastructure (e.g., log analytics, system statistics etc.), and (iii) Human Provided Processors, which are human experts that manually create an *IS*.

F. Dependency Manager

The *Dependency Manager* is responsible for resolving unit dependencies between the modeled entities, as described in the system model above. To achieve this the dependencies between entities in the system model are represented as a tree structure. Based on this tree structure the *Dependency Manager* creates a root node for each *TU*. It then creates a corresponding leaf node for each *TU* that is referenced in the dependency section of the *TU*. After this it checks the related *DUs* and adds them as leaves. The final step is to add the referenced *DIs* to the respective *DUs* as a leaf nodes. The final result is a complete tree structure that is sufficient for the *Deployment Handler* to effectively deploy a *TU* to an *IS*.

G. Infrastructure Manager

The role of the *Infrastructure Manager* is to handle all concerns regarding the infrastructure, where a specific deployment represented as a *DI* takes place. Its main responsibility is to ensure that all necessary resources described by the *IS* are accessible and available to successfully deploy the *TU*. This also includes all issues of authentication and authorization by ensuring the relevant credentials are provided. Additionally, it ensures that all constraints defined in *TUs* and *DUs* are satisfied. The system model distinguishes between hardware constraints and software constraints. Hardware constraints cover all constraints related to machine-specific aspects, including processors, memory, and disk space, as well as specific machine types in the context of IaaS and PaaS providers. Software constraints cover programming languages, runtime environments, or framework related aspects that need to be satisfied in order to ensure that an artifact can be successfully deployed on a specific *IS*. The *Infrastructure Manager* augments the *DI* accordingly and when finished hands it over to the *Deployment Handler*.

H. Repository Manager

The *Repository Manager* provides repositories for all units described in the system model and acts as a distributed registry keeping track of deployments and participating entities. It is responsible for system model storage and retrieval. It manages four distinct system model repositories utilizing distributed key value stores, which store the JSON-LD files that represent *TU, DU, IS and DIs* in a structured way. The *Repository Manager* provides a service interface to store and retrieve these files as well as a search interface to query *TUs*, *DUs*, *ISs*, and *DIs* based on specific elements.

I. Deployment Handler

The *Deployment Handler* is responsible for effectively executing a deployment. It follows the same processor-based principle as the previously introduced *Analyzers*. The *Deployment Handler* receives the resolved dependency structure as a tree model from the *Dependency Manager*. Each processor in the component is responsible for executing the deployment for specific infrastructure types. Processors can utilize different deployment mechanisms, including (i) Script Processors that utilize simple script-oriented approaches to execute the deployment (e.g., a processor using Bash scripts), (ii) Tool Processors that use more sophisticated tool-based approaches (e.g., Chef[5], LEONORE [10], or Capistrano[6]), and (iii) Human Interaction Processors to allow for deployments that need human interaction (e.g., command line input or user interface based approaches).

In order to ensure all credentials are available and all constraints are met the *Deployment Handler* interacts with the *Infrastructure Manager*. After the *Deployment Handler* has executed the deployment it invokes the *Deployment Checker* to verify that the deployment was successful. In case of a successful deployment the *Deployment Handler* augments the *DI* with all values that have been gathered during the deployment, such as specific service endpoints and IP addresses. In case of an unsuccessful deployment the *Deployment Handler* escalates to the *Infrastructure Manager*.

J. Deployment Checker

The *Deployment Checker* is responsible for verifying if a completed deployment was successful or not. To verify this the *Deployment Checker* retrieves the *TU* and *DI* from the *Deployment Handler* and executes the steps in the `verification` section of the *TU*. As mentioned above, verification steps default to following UNIX conventions, considering commands that exit with a result code of 0 to be successful, whereas other result codes are interpreted as errors. Additionally, *Deployment Checker* implementations can choose to use the `expected-result` element of the given *TU* to perform custom validation steps, e.g., based on pattern matching.

IV. Validation

A. Implementation

We implemented a preliminary prototype of the Smart Fabric framework in Ruby. To establish the frameworks microservice architecture we rely on REST as message exchange and interface technology. To serve the RESTful interfaces we use Sinatra[7] as web server for each of the implemented components.

The *Repository Manager* utilizes Redis[8] as the key value store for the repositories in order to provide fast and efficient storage for all elements of the system model. To enable the auto assembly mechanism for each processor within the framework we use RabbitMQ[9] as a message exchange middleware. Each implemented processor of every auto assembled component publishes its output and listens for its desired inputs on dedicated topics. This gives us the opportunity to receive messages based on patterns and allows for finer grained control over processor input values.

To implement the *Dependency Manager* we rely on the RubyTree library[10] to create the described tree structure. To enable system model entity augmentation and generation for the *Analyzer Processors* as well as for all *Deployment Handlers* we use a simple generator mechanism. This mechanism relies on predefined templates that are augmented with ruby code in order to easily build and modify system model entities. To achieve this we rely on the eRuby templating system provided by the Ruby standard library. The source code of our prototype implementation is available online[11].

B. Validation Scenario

For validation purposes we use the Smart Fabric framework to deploy and redeploy different sample applications. We demonstrate this based on two typical Smart City and IoT applications. The first one is an exemplary Ruby on Rails based Web Application with a RESTful service interface that represents a *Citizen Information System (CIS)*. The *CIS* is a common application type in the Smart City domain, which provides open data access for various city aspects such as traffic information and municipal energy usage. We chose this application type because on the one hand it is not fragmented into several artifacts and on the other hand because this type of application undergoes a natural infrastructure evolution due to increasing data and demand.

The second application is a sample IoT application based on a case study conducted in our lab in cooperation with a business partner and represents a *Building Management System (BMS)*. The *BMS* is a Java application based on the Spring Boot framework and is built as a microservice architecture

[5] https://www.chef.io/chef/

[6] http://capistranorb.com/

[7] http://www.sinatrarb.com/

[8] http://redis.io/

[9] https://www.rabbitmq.com/

[10] https://github.com/evolve75/RubyTree

[11] http://www.github.com/jomis/smartfabric

with multiple artifact dependencies. We choose the *BMS* since it provides a good contrast to the *CIS* due to its distributed nature with high artifact fragmentation, as well as its inherently large scale. For both applications we created appropriate *TUs*.

We then select four different infrastructures to deploy them to. These infrastructure types are a *Dedicated Server* hosted at our group, *Amazon AWS*[12] as an IaaS, *Heroku*[13] as a PaaS and an *OpenStack*[14] based private cloud hosted at our group. We chose this kind of infrastructure fragmentation because it represents a good coverage of different heterogenous infrastructures that are used to host todays application landscape. They also reflect the trend towards cloud based platforms and provide a good baseline for application evolution challenges [11]. The selected infrastructure types offer different services and capabilities, which leads to several challenges in redeploying them. The *Dedicated Server*, for example, imposes no restriction on the choice of databases (since deployed application packages must be managed by operators), whereas *Heroku* or *Amazon AWS* offer a restricted set of database services that are managed by the provider. This is the case for multiple aspects of these infrastructures and makes them an optimal testing scenario to show the effects of infrastructure evolution and the challenges this brings to infrastructure-agnostic application deployment.

For each of these infrastructures we create corresponding *ISs* as well as *DUs*. We then also provide exemplary *DIs* representing different specific deployments of our two *TUs* to the *ISs* based on the defined *DUs*. After this we test the following scenario. We initialize the Smart Fabric framework and load all previously defined system model entities. In the first step of our test we deploy the *CIS* and the *BMS* to the *Dedicated Server* by initiating two deployment requests to the Smart Fabric framework with the according *TUs* and *IS*. After the successful deployment notification by the framework we test the deployments with the information in the augmented *DIs*. In the case of the *CIS* we query the REST interface of the Traffic Resource and log the results. In the case of the *BMS* we query the REST interface of the Controller and log the results.

We then initiate two service requests to the Smart Fabric framework to transfer the *CIS* as well as the *BMS* to Heroku. We wait for the successful framework deployment notification and perform the previously described test again with the augmented information in the *DIs* and log the results accordingly.

In the final step we execute two additional transfer requests to the Smart Fabric framework, deploying the *CIS* to AWS and the *BMS* to our OpenStack cloud. After the successful framework deployment notification, we again test both application as previously described and log the results. We then compare the logged results with each other and ensure that deployment was successful on all infrastructures and produced the expected results.

In both sample applications we showed that Smart Fabric was able to successfully redeploy artifacts on heterogenous infrastructures without modifying application artifacts. For instance, Smart Fabric deployed the CIS on Amazon AWS with RDS as database backend whereas it deployed on Heroku using Heroku Postgres.

V. Related Work

The migration of application topologies between different deployment targets has been studied at various levels in the literature. With the adoption of the cloud paradigm, for instance, the problem of developing applications for and migrating existing applications to the cloud emerged [4]. To address this problems, Leymann et al. [12] present a meta model and tool that supports splitting an existing application in several parts, such that these parts can be moved to the cloud. In order to develop cloud infrastructure agnostic applications, Ardagna et al. [13] propose MODAClouds that provides a model-driven solution. Kwon et al. [14] present refactoring techniques and automated program transformations that help transitioning an application to use cloud-based services. In contrast to our work, these approaches solely focus on the internal design and execution of singular components of cloud-based applications, rather than providing a solution for overall architecture of a such applications.

Next to the migration of applications to, and designing applications for the cloud, another important area is the migration of an existing cloud application to a different cloud offering [15], [16], [17]. Binz et al. [18] present the Topology and Orchestration Specification for Cloud Applications (TOSCA), which aims for portable and standardized management of cloud services. TOSCA provides means for describing portable application deployment topologies consisting of nodes and their relationships. By specifying possible plans, TOSCA allows for governing the complex workflow of provisioning and deploying an application on cloud infrastructure. Based on TOSCA, Andrikopoulos et al. [19] propose the Generalized Topology Language (GENTL), an application topology modeling language, which provides the foundation for possible optimizations of the distributed deployment of the application.

Additionally, recent work was done to address the problem of vendor lock-in in the cloud [3], [20]. Satzger et al. [21], for instance, propose the meta cloud concept that proposes both design and run-time components to mitigate vendor lock-in by abstracting away technical incompatibilities from existing offerings. This approach helps in finding the right amount of cloud services that are needed for a specific use case, to support the initial deployment and runtime migration of an application. Sellami et al. [22] present a unified description model that represents an application and its requirements independently of the targeted PaaS. In addition, a generic application provisioning and management API is proposed to abstract away PaaS-dependent functionality. These two concepts combined, represent a PaaS-independent solution for the provisioning and management of applications in the cloud to avoid vendor lock-in. The aforementioned approaches have in common that they focus on deployment topologies of cloud applications, and therefore are complementary to the methodology presented in this paper. However, they do not provide a constraint-based, declarative specification of the required deployment infrastructure in order to allow for an infrastructure-agnostic deployment. Furthermore, all approaches have in common that they do not provide a mechanism and toolset that also deals

[12] http://aws.amazon.com/

[13] http://www.heroku.com

[14] https://www.openstack.org/

with the seamless migration between deployment targets, i.e., across infrastructure boundaries.

VI. Conclusion

The emergence of the cloud computing paradigm enables applications to autonomously provision and also release infrastructure resources in order to elastically respond to environmental changes (e.g., increased request load). To fully leverage the potential of the cloud however, applications need to be specifically designed and developed for a concrete cloud provider infrastructure, which leads to a strong dependence on specific offerings provided by the cloud provider. As a result, moving applications to the cloud or migrating an application between cloud providers is a tedious task due to variations of provided services, interfaces and deployment tools among providers. To address these problems, in this paper we presented Smart Fabric, a methodology and toolset to enable infrastructure-agnostic deployment of artifact topologies based on a declarative, constraint-based specification of the required deployment infrastructure. Current approaches do not sufficiently consider the specific, practical problems of dealing with evolving deployment infrastructure and closely tie application artifacts to their deployment targets. By extending the MADCAT methodology with a dedicated abstraction layer to clearly separate deployment infrastructure and application deployment, Smart Fabric allows for seamless, independent evolution of both, application components, as well as the underlying infrastructure. Moreover, our approach enables transparent application deployment and evolution between deployment targets, i.e., across traditional infrastructure boundaries (e.g., migrating applications between on-premise and PaaS offerings, or between PaaS and IaaS), without changes to application code. Smart Fabric implements a confidence-based decision model that aims to automate application deployment when possible, and will escalate to human operators when necessary. We discussed the feasibility of the introduced methodology and developed a prototype by using representative applications from the Smart City and IoT domains.

As part of our ongoing and future work, we will extend the presented framework with mechanisms to automatically gather infrastructure specifications from available cloud services, as well as deployment units and deployment instances from existing deployment directives to simplify adoption. Furthermore, we aim to extend and integrate Smart Fabric with our work on IoT cloud applications [10], [23], as well as our research on Smart City applications [24].

References

[1] M. Armbrust *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] P. Jamshidi, A. Ahmad, and C. Pahl, "Cloud Migration Research: A Systematic Review," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, pp. 142–157, 2013.

[3] G. C. Silva, L. M. Rose, and R. Calinescu, "A Systematic Review of Cloud Lock-In Solutions," in *Proceedings of the IEEE 5th International Conference on Cloud Computing Technology and Science*, 2013, pp. 363–368.

[4] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville, "Cloud migration: A case study of migrating an enterprise IT system to IaaS," in *Proceedings of the IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 450–457.

[5] M. Hüttermann, *DevOps for Developers*. Apress, 2012.

[6] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing Idempotence for Infrastructure as Code," in *Proceedings of the ACM/IFIP/USENIX 14th International Middleware Conference*. Springer Berlin Heidelberg, 2013, pp. 368–388.

[7] C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li, and S. Dustdar, "MADCAT: A methodology for architecture and deployment of cloud application topologies," in *Proceedings of the 8th International Symposium on Service Oriented System Engineering*. IEEE, 2014, pp. 13–22.

[8] S. Newman, *Building Microservices*. O'Reilly Media, Inc., Feb. 2015.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[10] M. Vögler, J. M. Schleicher, C. Inzinger, S. Nastic, S. Sehic, and S. Dustdar, "LEONORE - Large-Scale Provisioning of Resource-Constrained IoT Deployments," in *Proceedings of the 9th International Symposium on Service-Oriented System Engineering*. IEEE, 2015, pp. 78–87.

[11] N. Serrano, J. Hernantes, and G. Gallardo, "Service-Oriented Architecture and Legacy Systems," *Software, IEEE*, vol. 31, no. 5, pp. 15–19, 2014.

[12] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, and S. Dustdar, "Moving Applications To the Cloud: an Approach Based on Application Model Enrichment," *International Journal of Cooperative Information Systems*, vol. 20, no. 03, pp. 307–356, 2011.

[13] D. Ardagna *et al.*, "MODAClouds: A Model-driven Approach for the Design and Execution of Applications on Multiple Clouds," in *Proceedings of the 4th International Workshop on Modeling in Software Engineering*. IEEE, 2012, pp. 50–56.

[14] Y. W. Kwon and E. Tilevich, "Cloud refactoring: Automated transitioning to cloud-based services," *Automated Software Engineering*, vol. 21, no. 3, pp. 345–372, 2014.

[15] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[16] E. Hossny, S. Khattab, F. Omara, and H. Hassan, "A case study for deploying applications on heterogeneous paas platforms," in *Proceedings of the 2013 International Conference on Cloud Computing and Big Data*. IEEE, 2013, pp. 246–253.

[17] M. P. Papazoglou and W. J. V. D. Heuvel, "Blueprinting the cloud," *Internet Computing, IEEE*, vol. 15, no. 6, pp. 74–79, 2011.

[18] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, "Portable Cloud Services Using TOSCA," *Internet Computing, IEEE*, vol. 16, no. 3, pp. 80–85, 2012.

[19] V. Andrikopoulos, A. Reuter, S. Gómez Sáez, and F. Leymann, "A GENTL Approach for Cloud Application Topologies," in *Service-Oriented and Cloud Computing*. Springer Berlin Heidelberg, 2014, vol. 8745, pp. 148–159.

[20] G. C. Silva, L. M. Rose, and R. Calinescu, "Towards a Model-Driven Solution to the Vendor Lock-In Problem in Cloud Computing," in *Proceedings of the 5th International Conference on Cloud Computing Technology and Science*. IEEE, 2013, pp. 711–716.

[21] B. Satzger, W. Hummer, C. Inzinger, P. Leitner, and S. Dustdar, "Winds of change: From vendor lock-in to the meta cloud," *Internet Computing, IEEE*, vol. 17, no. 1, pp. 69–73, 2013.

[22] M. Sellami, S. Yangui, M. Mohamed, and S. Tata, "PaaS-independent provisioning and management of applications in the cloud," in *Proceedings of the IEEE International Conference on Cloud Computing*. IEEE, 2013, pp. 693–700.

[23] M. Vögler, F. Li, M. Claeßens, J. M. Schleicher, S. Nastic, and S. Sehic, "COLT Collaborative Delivery of lightweight IoT Applications," in *Proceedings of the International Conference on IoT as a Service*. Springer, 2014, p. to appear.

[24] J. M. Schleicher, M. Vögler, C. Inzinger, W. Hummer, and S. Dustdar, "Nomads - Enabling Distributed Analytical Service Environments for the Smart City domain," in *Proceedings of the IEEE International Conference on Web Services, Application Track*. IEEE, 2015, p. to appear.