Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. ????

# Bootstrapping Provisioning Engines for On-demand Provisioning in Cloud Environments

Lukas Reinfurt

**Studiengang:** Informatik

**Prüfer:** Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
**Betreuer:** Dipl.-Inf., Dipl.-Wirt. Ing.(FH) Karolina Vukojevic
**begonnen am:** ??.??.2013
**beendet am:** ??.??.2014

**CR-Klassifikation:** ???

# Abstract

# Table of Contents

# List of Figures

# List of Tables

# List of Listings

# List of Abbreviations

| | |
|---|---|
| AMI | Amazon Machine Images, page 13 |
| AWS | Amazon Web Services, page 11 |
| BPEL | Business Process Execution Language, page 12 |
| BPMN | Business Process Model and Notation, page 12 |
| CSAR | Cloud Service Archive, page 13 |
| ESB | Enterprise Service Bus, page 18 |
| JPF | Java Plugin Framework, page 31 |
| JSPF | Java Simple Plugin Framework, page 31 |
| OASIS | Organization for the Advancement of Structured Information Standards, page 11 |
| OSGi | Open Service Gateway initiative, page 31 |
| PubSub | publish-subscribe pattern, page 28 |
| SimTech | Simulation Technology, page 10 |
| TOSCA | Topology and Orchestration Specification for Cloud Applications, page 11 |

# 1 Introduction

Workflow technology and the service based computing paradigm were mostly used in a business context until now. But slowly they are extended to be used in other fields, such as eScience, where business centric assumptions that where previously true aren't reasonable anymore. One of these assumptions is that services should run continuously. This made sense in large enterprise where those services are used often. Science, on the other hand, often takes a more dynamic approach, where certain services, for example for simulation purposes, are only used at certain times. In those cases, it would make more sense to dynamically provision service only when they are needed.

## 1.1 Task of this Diploma Thesis

The task of this diploma thesis is to design a lightweight bootstrapping system that can kick off dynamic provisioning in cloud environments. It should be able to provision various provisioning engines in all kinds of cloud environments. The provisioning engines then handle the actual provisioning of required workflow systems and services. A managing component that keeps track of provisioned environments is also part of this system.

Support for different cloud environments and provisioning engines should be achieved through [redacted] means of software engineering. A functioning prototype that supports Amazon as cloud environment and TOSCA as provisioning engine should be implemented.

## 1.2 Structure of this Document

...

# 2 Fundamentals

This chapter starts with a short overview of the SimTech project, of which this diploma thesis is a part of. Next, bootstrapping is defined in the context of this diploma thesis, since it can have various different meanings. Then, a short overview of the cloud landscape is presented, with focus on Amazons cloud offerings, since these are used in this diploma thesis. Finally, we take a look at provisioning solutions, in particular TOSCA, which is also used later in this thesis.

## 2.1 SimTech

Since 2005, the German federal and state government have been running the Excellence Initiative[1], which aims to promote cutting-edge research, thereby increasing the quality and international competitiveness of German universities. In three rounds of funding, universities have competed with project proposals in three areas: Institutional Strategies, Graduate Schools, and Clusters of Excellence. In total, the winners received approximately 3.3 billion euros since 2005, split up between 14 Instituational Strategies, 51 Graduate Schools, and 49 Clusters of Excellence [3, pp. 16-18].

Simulation Technology (SimTech) is one of the Clusters of Excellence that are funded by the Excellence Initiative. In a partnership between the University of Stuttgart, the German Aerospace Center, the Fraunhofer Institute for Manufacturing Engineering and Automation, and the Max Planck Institute for Intelligent Systems, it combines over 60 project from researchers in Engineering, Natural Science, and the Life and Social Sciences. The aim of SimTech is to improve existing simulation strategies and to create new simulation solutions [3, pp. 109].

In the SimTech project, seven individual research projects collaborate in nine different project networks, one of which is project network 8: *Integrated data management, workflow and visualization to enable an integrative systems science*. The goal of this project network is to build

[1] `http://www.dfg.de/en/research_funding/programmes/excellence_initiative/index.html`

an easy-to-use infrastructure that supports scientists in their day to day work with simulations [8].

## 2.2 Bootstrapping

## 2.3 Cloud

### 2.3.1 Amazon Web Services

In 2006, Amazon started offering cloud resource under the umbrella of Amazon Web Services (AWS). Since then, their offerings steadily increased and do now comprise over 20 different products and services for computing, data storage, content delivery, analytics, deployment, management, and payment in the cloud [1].

## 2.4 Provisioning Solutions

This section describes some of the provisioning solution available today, in particular TOSCA and Open TOSCA, since those are used in the prototypical implementation later on.

### 2.4.1 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard that is currently being worked on by the Organization for the Advancement of Structured Information Standards (OASIS)[2]. Its development is also supported by various industry partners, which include IBM, Cisco, SAP, HP and others. It's aim is to provide a language that can describe service components and their relations in a cloud environment independent fashion [10].

TOSCA defines an XML syntax, which describes services and their relations in a so called service templates. Figure 2.1 shows that a service template can be made of up of four distinct parts: Topology templates, orchestration plans, reusable entities, and artifact templates.

[2] https://www.oasis-open.org/

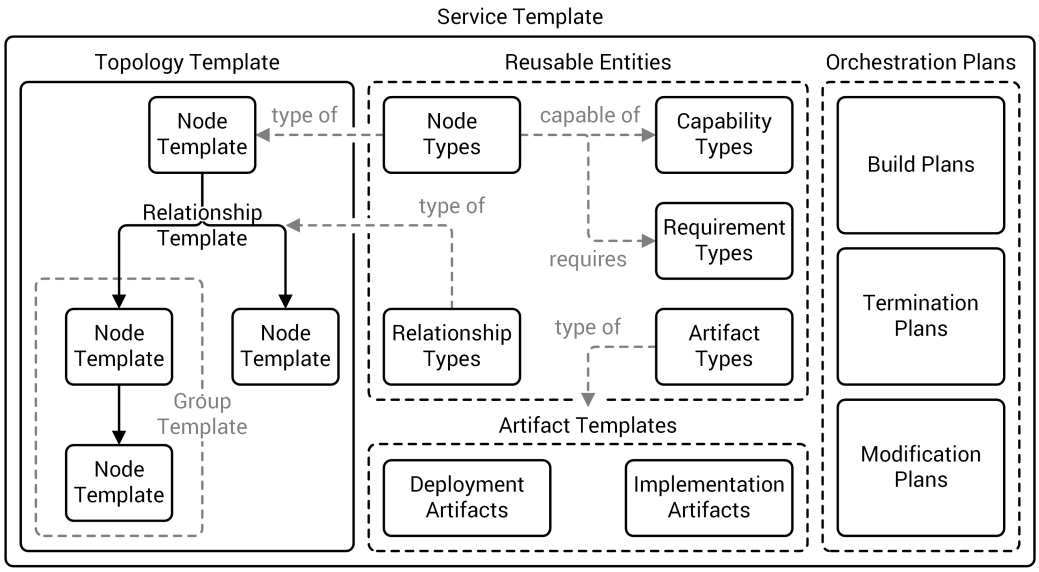


**Figure 2.1:** TOSCA service template structure [based on 10].

Topology templates, as seen on the left side of Figure 2.1, model the structure of a service as a directed graph. The vertices of the graph represent nodes, which are occurrences of a specific component, for example, an application server or a database. These nodes are defined by node types, or by other service templates. Node types are reusable entities, as shown in the top center of Figure 2.1. They define the properties of a component, as well as operations to manipulate a component, so called interfaces. Additionally, node types can be annotated with requirements and capabilities. These, in turn, are defined by requirement and capability types, which also belong to the group of reusable entities. This allows for requirement and capability matching between different components. The edges of the graph represent connections between nodes, which are defined by relationship templates that specify the properties of the relation. An example for such a connection would be a node A, representing a web service, which is deployed on node B, an application server. Relationship types are also used to connect requirements and capabilities.

Orchestration plans, shown on the left of Figure 2.1, are used to manage the service that is defined by the service template. TOSCA distinguishes between three types of plans: Build plans, termination plans, and modification plans. Build plans describe, how instances of a service are created. Termination plans describe, how such a service is removed. Modification plans manage a service during its runtime. These plans consist of one or more tasks, i.e., an operation on a node (via an interface) or an external service call, and the order in which these tasks should be performed. They can be written in Business Process Model and Notation (BPMN) or Business Process Execution Language (BPEL), which are already existing

languages to describe process models.

The bottom center of Figure 2.1 shows artifact templates, which represent artifact. Artifacts are things that can be executed directly (e.g.: scripts, archives) or indirectly (e.g.: URL, ports). TOSCA further distinguishes between two types of artifacts, namely deployment and implementation artifacts. Deployment artifacts materialize instances of a node and are used by a build plan to create a service. An example for this is an Amazon Machine Images (AMI), which creates an Apache server once deployed in a VM. Implementation artifacts represent the interfaces of components. Here, an example would be a node that has an interface for starting the particular component described by the node. This interfaces could be implemented by an implementation artifact like a *.jar* file.

One or more TOSCA service templates are packaged, together with some meta data, into a Cloud Service Archive (CSAR), which is essentially a zip file that contains all files necessary to create and manage a service. CSAR files can then be executed in a TOSCA runtime environment, also called TOSCA container, to create the service described within.

### 2.4.2 OpenTOSCA

OpenTOSCA is a browser based open-source implementation of a TOSCA container, created at the IAAS at University Stuttgart, which supports the execution of TOSCA CSAR archives. Figure 2.2 shows the architecture of OpenTOSCA. Its functionality is realized in three main components, which are the Controller, the Implementation Artifact Engine, and the Plan Engine. After a CSAR is uploaded to OpenTOSCA it can be deployed in three steps. In the first step, the CSAR file is unpacked and its content is stored for further use. The TOSCA XML files are then loaded and processed by the Controller. The Controller in turn calls the Implementation Artifact Engine and the Plan Engine. The Implementation Artifact Engine knows how to deploy and store the provided implementation artifacts via plugins. Plans are then run by the Plan Engine, which also uses plugins to support different plan formats. OpenTOSCA also offers two APIs, the Container API and the Plan Portability API. The Container API can be used to access the functionality provided by the container from outside and to provide additional interfaces to the container, like the already existing admin UI, self-service portal, or modeling tool. The Plan Portability API is used by plans to access topology and instance information [2].

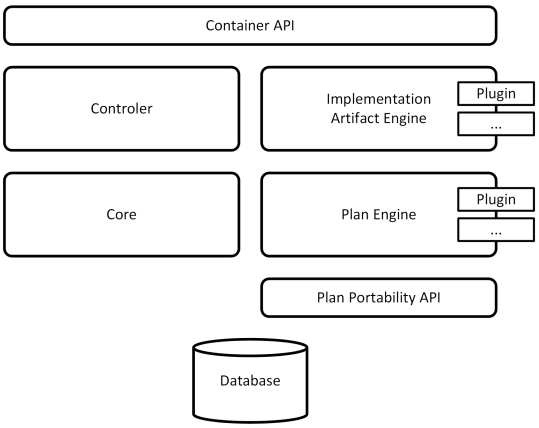


**Figure 2.2:** OpenTOSCA architecture [based on 2].

# 3 Related Work

This chapter summarizes related work of other authors that is of interest to this thesis. First, we present the paper that laid the foundation for this diploma thesis. Then we take a look at another diploma thesis which expanded the ideas presented in the first paper. We also take a look at a paper that presents work similar to this diploma thesis.

## 3.1 On demand Provisioning

Vukojevic-Haupt, Karastoyanova, and Leymann identify requirements that need to be addressed to make the current approach for scientific workflows used by the SimTech SWfMS more suitable for scientific simulation work. These requirements are: Dynamic allocation as well as release of computing resources, on-demand provisioning and deprovisioning of workflow middleware and infrastructure, and dynamic deployment and undeployment of simulation services and their software stacks. To fulfill these requirements, they propose a new service binding strategy that supports dynamic service deployment, an approach for dynamic provisioning and deprovisioning of workflow middleware, an architecture that is capable of these dynamic deployment and provisioning operations, and, as part of this architecture, the bootware - the subject of this diploma thesis - that kicks of these dynamic processes [11].

The new service binding strategy is necessary, because existing static and dynamic binding strategies, as shown on the left and in the center of Figure 3.1, rely on services that are always running, or, as in the case of dynamic binding with service deployment, only dynamically deploy the service, but not its middleware and infrastructure. The new service binding strategy, shown on the right of Figure 3.1, called *dynamic binding with software stack provisioning*, is similar to the already existing dynamic binding with service deployment strategy, but adds the dynamic provisioning of the middleware and infrastructure required by the service [11].

Their approach for dynamic provisioning and deprovisioning of workflow middleware is separated into six steps. The first step is to model and start the execution of a simulation work-

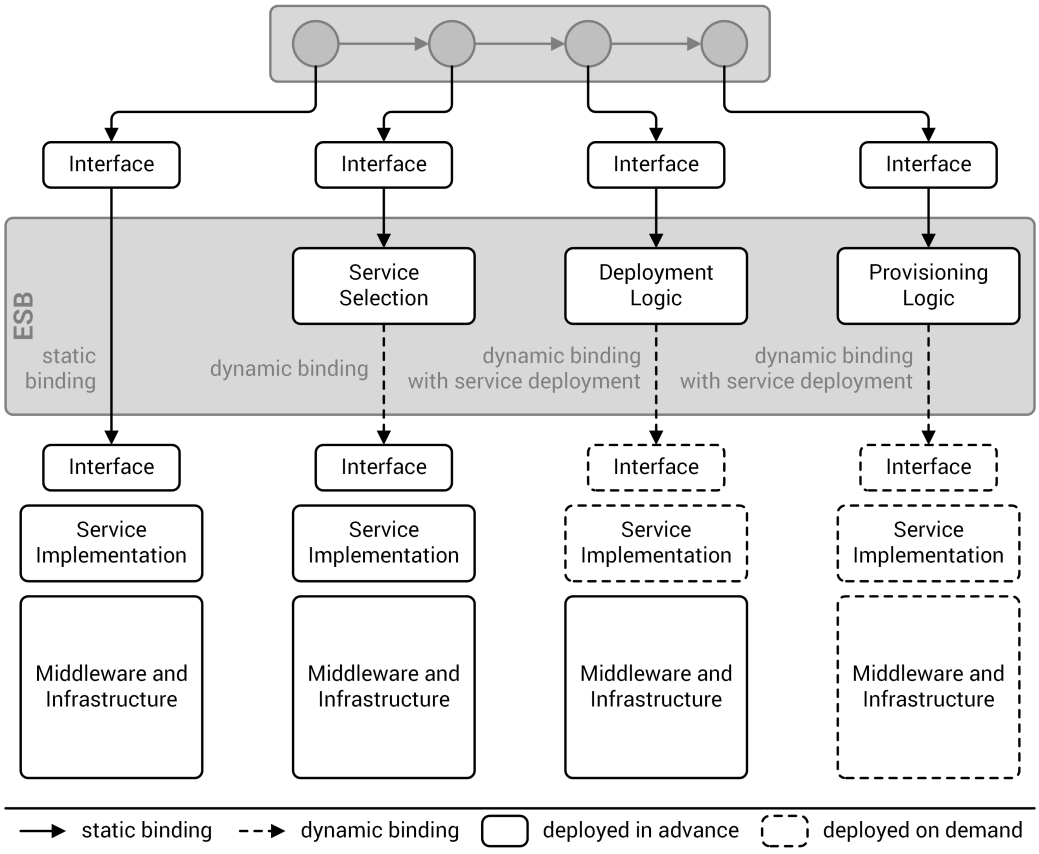


**Figure 3.1**: Simplified overview of service binding strategies [based on 11].

flow. For this, the Modeler component shown on the left of Figure 3.2 is used. In the second step, the middleware for executing the worklfow, e.g. the SWfMS, and its underlying infrastructure are provisioned. This is shown in Figure 3.2 as deployment step one and two. In deployment step one, the bootware deploys a provisioning engine into a cloud environment. In deployment step two, this provisioning engine is used to deploy the actual middleware. Now, the workflow can be deployed on this middleware, which is step three. In step four, an instance of this workflow is executed. During this execution, some external service might be required that are not yet available. The ESB determines this by checking the service registry. If the requested service isn't available, the ESB tells the provisioning engine to provision this service. The on-demand provisioning of services is step five, which corresponds to deployment step three in Figure 3.2. The service is also deprovisioned if it is no longer needed. The final step is to deprovision the workflow model and the workflow execution middleware after the execution of the workflow instance is finished [11].

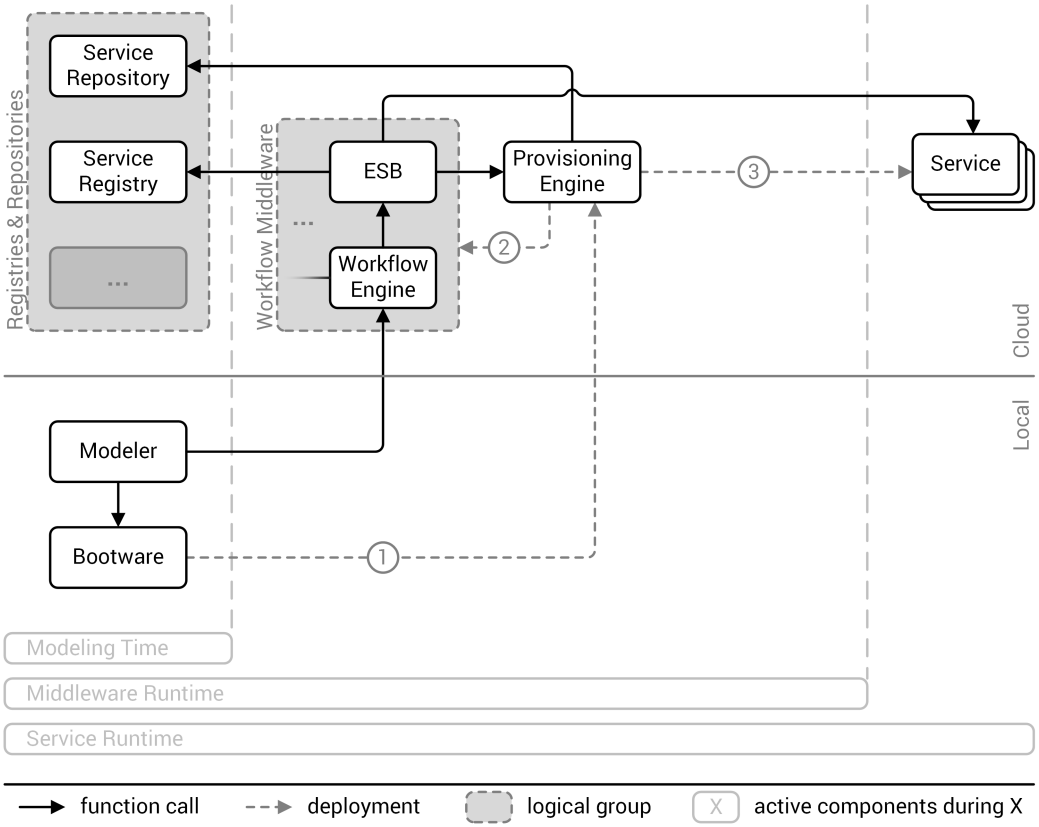


Figure 3.2: Proposed architecture [based on 11].

The architecture they present can be separated into a local part and a cloud part, as well as different phases. Figure 3.2 shows that the only local components are the modeler and

the bootware, while all other components are hosted in the cloud. In the modeling phase, a scientist uses local modeling and monitoring tools in combination with cloud hosted repositories and registries to create a workflow. These components are always running. When he deploys the workflow, the local bootware component kicks of the on demand provisioning process and therefore the second phase, called middlware runtime phase. In this phase, the bootloader deploys a provisioning engine in the cloud, which in turn deploys the workflow middleware. Once the middleware is up and running, the workflow can be executed. During the execution, the ESB receives service calls from the workflow engine. Services, that are not running at this time can then be provisioned by the provisioning engine. This takes place in the third phase, the service runtime phase [11]. The bars at the bottom of Figure 3.2 show, which components are active during which phase.

## 3.2 Dynamic

Schneider further refines the previously shown middleware architecture by adding a provisioning manager as intermediary between the Enterprise Service Bus (ESB) and the provisioning engines [9]. This addition improves the original architecture in three aspects.

The ESB can now use the stable interface of the provisioning manager to trigger provisioning engines instead of calling those provisioning engines directly. The provisioning manager handles the differences between the provisioning engines. This makes it also possible to use multiple different provisioning engines during one workflow execution.

The provisioning manager also handles the communication with the service registry or possibly multiple service registries for different provisioning engines.

The provisioning manager could also translate different service distribution formats so that provisioning engines could be used with formats that they don't support

## 3.3 Architecture

Kirschnick et al. present an architecture for automatic provisioning of cloud infrastructure and services [5].

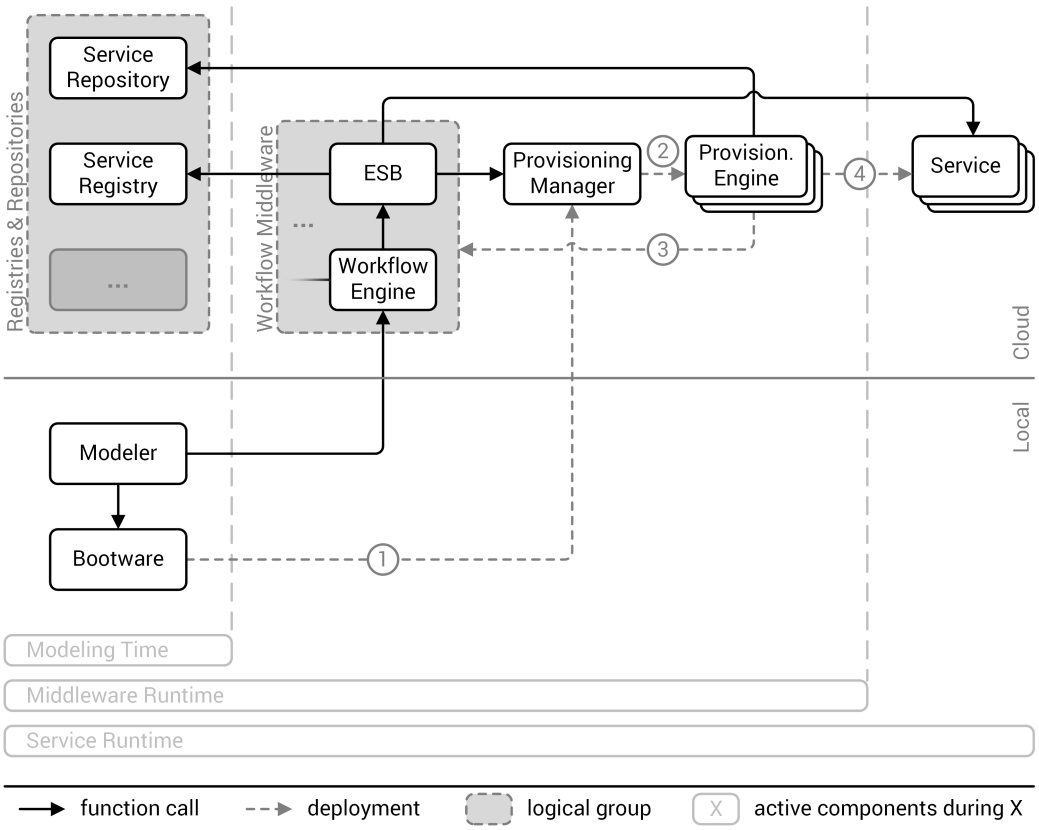


**Figure 3.3:** Extended architecture with added provisioning manager

# 4 Requirements, Constraints and Design

## 4.1 Requirements

Requirements that should be fulfilled:

- The Bootware component should be as lightweight as possible.
- The Bootware component should be as generic as possible.
- The Bootware component should be extensible.
- Multiple cloud providers should be supported. At least Amazon should be implemented.
- Multiple provisioning engines should be supported. At least OpenTosca should be implemented.
- It should be possible to start the bootware process by deploying a workflow in the modeler.
- Open standards should be used where possible.
- As much as possible of the internals of the bootware component should be hidden from the end user

Added requirements:

- It should be possible to get feedback of the state of the provisioning process. More than start/end of Web Service Call/Return.

implicit requirements:

- Java 1.6

## 4.2 Constrains

## 4.3 Design

In this section we will develop the design of the bootware. This design is held intentionally abstract. Specific implementation details will be selected and described in section 5.1.

### 4.3.1 Component Division

The first question that has to be answered is whether or not it makes sense to divide the bootware component into two or more cooperating components, i.e. a server client division or a similar structure. As described in section 3.1 on page 15, the proposed architecture initially only envisioned one bootware component. This architecture was expanded with the introduction of the provisioning manager, as described in section 3.2 on page 18. We will now discuss the viability of these architectures.

#### Single Local Component

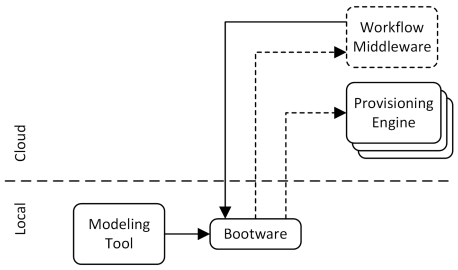


Figure 4.1: Simplified overview of the single local component architecture

First we consider the simplest case: A single local component. In this scenario, all provisioning processes are initiated from a component installed locally on the users machine, alongside or as part of the workflow modeler.

The advantages of this architecture lie in its simplicity. Only one component has to be created and managed. There is no possible overlap in functionality, as in a 2-tier architecture (see section 4.3.1). Communication between multiple bootware components doesn't have to be considered.

The disadvantages are caused by the component being local. Since all the functionality is concentrated in one component, it can become quite large and complicated, is is one thing that should be avoided according to the requirements. A much bigger problem however is the remote communication happening in this scenario. All calls to the bootware component from the ESB of the workflow middleware would leave the remote environment. Also, all calls from the bootware component to the provisioning engines would enter the remote environment. This type of split communication can be costly and slow, as shown by Li et al. They compared public cloud providers and measured that intra-datacenter communication can be two to three times faster and also less expensive (often free) compared to inter-datacenter communication [6].

#### Single Remote Component

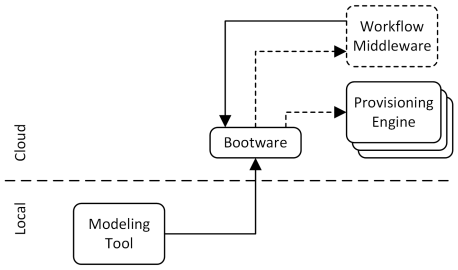


Figure 4.2: Simplified overview of the single remote component architecture

The next obvious choice is to put the single bootware component into a remote environment, where the disadvantages of local to remote communication would disappear.

However, this creates a new problem. We don't know where exactly to put the bootware component. Since one requirement is that multiple cloud environments should be supported, it's possible that the bootware component is not located anywhere near the cloud environment where it should provision further components. The communication problem of the single local bootware component still exist.

Another problem is that now we would have to manage some sort of load balancing and the bootware component would have to support multiple tenancy or be stateless, or otherwise every user needs its own remote instance. The first case would further complicate the design and implementation of the bootware component. The second case would increase monetary and management costs.

### 2-Tier Architecture

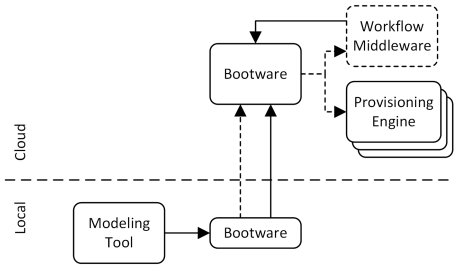


Figure 4.3: Simplified overview of the 2-tier architecture

Next, we take a look at a 2-tier architecture, where the bootware is divided into two components. On the local side we have a small and simple component which has mainly one function: To provision the larger second part of the bootware in a remote environment near to the environment, where other components will be provisioned later.

This eliminates the problems of a single bootware component. We can now keep the local part as simple as possible and make the remote part as complicated as it has to be. We now also can position the remote component close to other remote components to minimize local/remote communication and the problems resulting of it.

But we also introduce new problems. For one, we now have duplicate functionality between the two components. Both components have to know how to provision a component into multiple cloud environments. The local component has to be able to put its remote counterpart into any cloud environment. The remote component has to be able to provision other components into the same environment in which it runs (ideally, to minimize costs). Since itself can be located in any cloud environment, it has to be able to do this in any cloud environment. Independent from this, it has to be able to provision to any environment that the user/service package chooses. But this problem can be solved by using a plugin architecture, which allows both components to use the same plugins. We discuss plugins in detail in subsection 4.3.2.

A second problem which we can't avoid but can solve is the communication which is now necessary between the different parts of the bootware. More on this in subsection 4.3.3

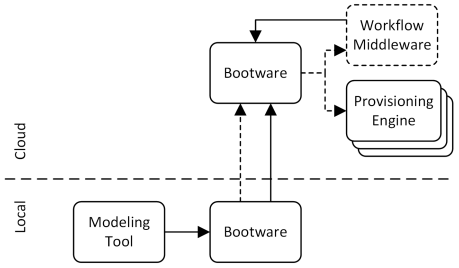


Figure 4.4: Simplified overview of the cloned component architecture

#### Cloning

This architecture can be seen as an alternative form of the 2-tier architecture described in section 4.3.1. In this case there are also two parts working together and the remote part does most of the work. However, the local and the remote components are identical. Instead of provisioning a bigger component in a remote environment, the local component clones itself. Compared to the 2-tier architecture described before, this has the advantage that only one component has to be designed and implemented and that function duplication is not an issue. On the other hand, this architecture makes only sense if the functionality of the two separate components in the 2-tier architecture turns out to be mostly identical. Therefore we can't decide yet if this architecture should be used.

#### Decision

Of the four alternative presented here, alternative three - the 2-tier architecture - makes the most sense. Therefore it is selected as the alternative of choice and used for further discussion. We do however retain the option to transform it into alternative four if this seems advantageous.

### 4.3.2 Extensibility

The requirements for the bootware component state that support for different cloud environments and provisioning engines should be achieved through means of software engineering. This requirement is intentionally vague to allow to select a fitting extension mechanism during the design process.

### Extension Mechanisms

other mechanisms?

One possibility that would satisfy this criteria is to design interfaces for all extension points of the bootware component. New cloud environments, provisioning engines, or other extension could then implement these interfaces. The disadvantage of this approach is that the whole bootware component has to be recompiled, redistributed and redeployed if one extension is added or changed.

To remove this disadvantage, a more flexible architecture is needed, for example a plugin architecture. (pattern?) Interfaces for the extension points still exist but the extension are no longer part of the main bootware component. They are compiled separately into plugins that can be loaded into the main bootware component on the fly. There are several possibilities to realize such an architecture.

It is certainly possible to implement a plugin framework from scratch. An advantage of this approach would be that the design of the plugin architecture could be tailored to our use case and would be as simple or complex as needed. But there are also several disadvantages. For one, we would reinvent the wheel, since multiple such frameworks already exist. It would also shift resources away from the actual goal of this thesis, which is designing the bootware component. Furthermore it would require a deep understanding of the language used for the implementation (in this case Java), which is not necessarily given. Therefore it seems more reasonable to use one of the already existing plugin frameworks.Three such alternatives will be compared next. (more?)

### Plugin Repository

Now that we have introduced plugins we face new problems. Figure 4.5 shows the current architecture, where both bootware components use their own plugins. If a plugin is added or updated, the user has to manually copy this plugin to the right folder of one or both of the bootware components. Furthermore, if both components use the same plugins, which they will (cloud plugins), we will have duplicate plugins scattered around. This is inefficient, probably annoying for the user and possibly dangerous (other world, fehler hervorrufend) if plugins get out of sync.

To remedy this situation we introduce a central plugin repository, as shown in Figure 4.6. This repository holds all plugins of both components so it eliminates duplicate plugins. If plugins are added or modified it has only to be done in one place. Plugin synchronization

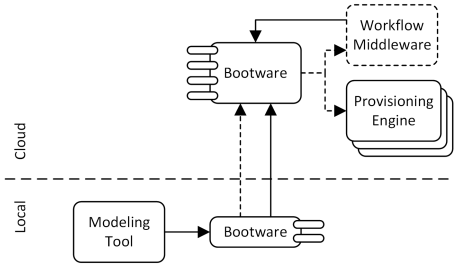


Figure 4.5: Simplified overview of the 2-tier architecture with plugins

can happen automatically when the bootware components start, so that the user is no longer involved in plugin management. The repository also enables easy plugin sharing, which was cumbersome earlier.

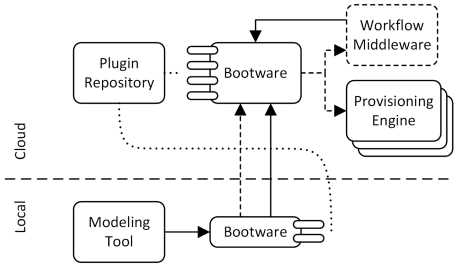


Figure 4.6: Simplified overview of the 2-tier architecture with a plugin repository

While a central plugin repository is a sensible addition to the proposed bootware architecture, its design and implementation are out of scope of this thesis.

### 4.3.3 External Communication

Since we will use a 2-tiered approach for the bootware component, we now have to decide, how the communication between the two components will work. There are several factors that impact this decision.

Communication between the components should be as simple as possible, but has to support some critical features.

Since the provisioning processes kicked off by the bootware can potentially take a lot of time to finish (in the range of minutes to hours), asynchronous communication should be used between the components to avoid timeouts and blocking resources.

For the same reason, there should be some mechanism to get feedback on the current status during a long running provisioning process.

There will be other components that have to call the bootware to start some provisioning process, so communication has to be open for other parties.

The communication with the bootware components will contain sensitive data, for example login information for cloud providers that the bootware needs to do its job, but has to be provided from the outside on a call to call basis. This information has to be transported securely to prevent malicious or fraudulent attacks. The selected communication method therefore has to support some sort of security mechanisms, ideally end-to-end security via message encryption. While these security mechanisms will not be used in this thesis due to time constraints, selecting the right communication method is still critical for future development.

Since this whole project is concerned with services, it liegt nahe to use web service calls and returns as main communication method, internally as well as externally. This way we offer a consistent interface for all components to use. Figure 4.7 shows the addition of asynchronous web service call and return communication to the proposed architecture. Using a web service with soap messaging also gives us access to Web Service Security Framework, which supports end-to-end security with encryption and signatures.

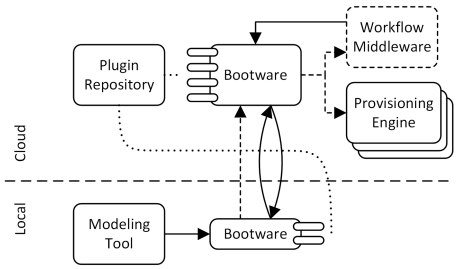


**Figure 4.7:** Simplified overview of the 2-tier architecture with asynchronous web service communication

Web Services also support asynchronous communication so that long running provisioning processes won't pose a problem. We do however still need information during those long running processes to give the user some feedback. This can't (???) be accomplished by this simple request/response pattern. For this, a secondary communication mechanism which supports sending multiple feedback messages has to be used.

Since it is not necessary for the successful use of the bootware it would make sense to implement this secondary communication mechanism as a plugin. This would allow us to add arbitrary communication plugins to the bootware depending on future needs.

One natural choice for this is the PubSub pattern implemented by some messaging middleware. Using this pattern, the remote bootware component pushes messages to a message queue to which the local bootware component (and other components if needs be) can subscribe to receive future messages. Figure 4.8 shows an additional message queue.

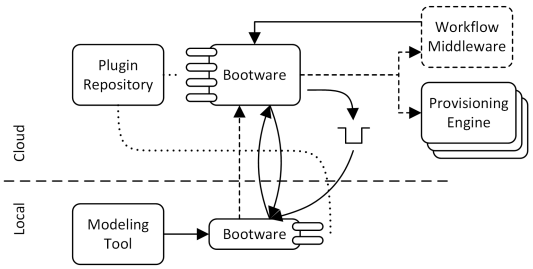


Figure 4.8: Simplified overview of the 2-tier architecture with asynchronous web service and a messaging queue communication

### 4.3.4 Internal Communication

We also have to consider internal communication between the bootware core and plugins, and possibly also in between plugin. Ideally, every plugin will be able to react to events from the bootware. These event could be triggered by the bootware core or by any plugin, but plugins should be completely independent from each other. Since a plugin doesn't know about other plugins, it can't listen for events at other plugins directly. The only known constant to a plugin is the bootware core. Therefore we need a communication mechanism which allows for loosely coupled communication between the bootware core and the plugins, where plugins can register their interest for certain events with the core and also publish their own events to the core for other plugins to consume. This essentially describes the publish-subscribe pattern (ref).

#### PubSub

The publish-subscribe pattern (PubSub) is a messaging pattern that consists of three types of participant who aren't necessarily mutually exclusive: Event busses (or message broker), publishers, and subscribers. The event bus sits at the center of the communication. He receives messages from publishers and distributes them to all subscribers that have voiced their interest in messages of a certain type by subscribing at the event bus.

Using this pattern in our bootware component, we would create an event bus at the bootware

core and plugins, as well as other parts of the core, could subscribe at this event bus and also publish messages through this event bus.

# 5 Implementation

## 5.1 Selecting Frameworks and Libraries

### 5.1.1 Plugin Frameworks

| | | *Plugin Frameworks* | | |
|---|---|---|---|---|
| | | JSPF[1] | JPF[2] | OSGi[3] |
| *Features* | Security | ✗ | ✗ | ✓ |
| | Dynamic Loading | ✗ | ✓ | ✓ |
| | Complexity | low | medium | high |
| | Active Development | ✗ | ✗ | ✓ |
| | Popularity | low | low | high |
| | Standard | ✗ | ✗ | ✓ |
| | Used in SimTech | ✗ | ✗ | ✓ |

Table 5.1: Feature comparison of Java plugin frameworks

All of the frameworks that we compare here offer the basic functionality that we need to extend the core bootloader component, i.e. the developer defines interfaces that then are implemented by one or more plugins. These plugins are compiled separately from the main component and are then packaged in *.jar* files for distribution. These packages are loaded during runtime and provide the implementation for the specific interface they implement. There are however some advanced functional differences and some non-functional differences that will be considered here.

Dynamic loading allows us to load and replace plugins during runtime, without completely restarting the application. While we don't know for certain if dynamic loading is needed in our case, it's one of the advanced features that might be nice to have in the future.

[1] `https://code.google.com/p/jspf/`
[2] `http://jpf.sourceforge.net/`
[3] `http://www.osgi.org/`

Security is a must have feature but is out of the scope of this thesis. Consider the following scenario: The bootware component is used by multiple separate users who can share plugins using a plugin repository. Without security features, a malicious user could upload a plugin to this repository which, in theory, could contain any code. Therefore it's important to select the right framework now, so that security features can be implemented in the future.

Some non-functional features should also be considered, such as complexity, popularity, and if the framework is still in active development.

Java Simple Plugin Framework (JSPF)[1] is a plugin framework build for small to medium sized projects. Its main focus is simplicity. Therefore it does not support many of the advanced features like dynamic loading or security that other solution support. The author explicitly states that it is not intended to replace JPF or OSGi [4].

Java Plugin Framework (JPF)[2] is an open-source plugin framework. Compared to JSPF it supports some advanced features like dynamic loading of plugins during runtime. It is also more popular then JSPF. However, the last version was released in 2007. This is not necessarily bad but might show that there will be no future development of this framework.

Open Service Gateway initiative (OSGi)[3] is a plugin framework standard developed by OSGi Alliance. It provides a general-purpose Java framework that supports the deployment of extensible bundles [7].

Decision

### 5.1.2 Web Service

### 5.1.3 PubSub Libraries

Many of the well know Messaging Middlewares offer support for PubSub, for example ActiveMQ,.... But, since we are looking for an internal communication mechanism only, all of these solutions are somewhat overpowered. We don't need guaranteed delivery, message queuing, scalability,.... Instead, we need a lightweight in-memory solution. Therefor we will ignore the middleware heavyweights and look for smaller, more light weight PubSub

libraries.

| | | *PubSub Libraries* | | | | |
|---|---|---|---|---|---|---|
| | | MBassador[4] | Guava Event Bus[5] | Simple Java Event Bus[6] | EventBus[7] | Mycila PubSub[8] |
| *Features* | Something | ✗ | ✗ | ✓ | ✓ | ✓ |

Table 5.2: Feature comparison of Java PubSub libraries

[4] https://github.com/bennidi/mbassador
[5] https://code.google.com/p/guava-libraries/wiki/EventBusExplained
[6] https://code.google.com/p/simpleeventbus/
[7] http://eventbus.org/
[8] https://github.com/mycila/pubsub

# 6 Summary and Conclusion

# Bibliography


[1] *About AWS*. Amazon Web Services, Inc. URL: http://aws.amazon.com/about-aws/.

[2] Tobias Binz et al. "OpenTOSCA -- A Runtime for TOSCA-based Cloud Applications". In: International Conference on Service-Oriented Computing. LNCS. Springer, 2013. URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2013-45%20-%20OpenTOSCA_A_Runtime_for_TOSCA-based_Cloud_Applications.pdf.

[3] *Excellence Initiative at a Glance*. Tech. rep. German Research Foundation, Aug. 2013. URL: http://www.dfg.de/download/pdf/dfg_im_profil/geschaeftsstelle/publikationen/exin_broschuere_1307_en.pdf.

[4] *Java Simple Plugin Framework F.A.Q.* Ralf Biedert. URL: https://code.google.com/p/jspf/wiki/FAQ.

[5] Johannes Kirschnick et al. "Toward an Architecture for the Automated Provisioning of Cloud Services". In: *Communications Magazine* 48.12 (2010), pp. 124–131. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5673082.

[6] Ang Li et al. "CloudCmp: Comparing Public Cloud Providers". In: 10th ACM SIGCOMM conference on Internet measurement. 2010, pp. 1–14. URL: http://ftp.cs.duke.edu/~xwy/publications/cloudcmp-imc10.pdf.

[7] *OSGi Service Platform core Specification*. Tech. rep. OSGi Alliance, Apr. 2011. URL: http://www.osgi.org/download/r4v43/osgi.core-4.3.0.pdf.

[8] *PN 8: Integrated data management, workflow and visualisation to enable an integrative systems science*. SRC Simulation Technology. URL: http://www.simtech.uni-stuttgart.de/forschung/pn/pn8/index.en.html.

[9] Valeri Schneider. "Dynamische Provisionierung von Web Services für Simulationsworkflows". Diploma. University Stuttgart, 2013. URL: ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-3473/DIP-3473.pdf.

[10] *Topology and Orchestration Specification for Cloud Apllications*. Committee Specification 01. OASIS, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.pdf.

[11] Karolina Vukojevic-Haupt, Dimka Karastoyanova, and Frank Leymann. “On-demand Provisioning of Infrastructure, Middleware and Services for Simulation Workflows”. In: Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013). 2013. URL: ???.

All links were last visited on April 7, 2014.

# Declaration of Authorship

I hereby certify that the diploma thesis entitled

*Bootstrapping Provisioning Engines for On-demand Provisioning in Cloud Environments*

is entirely my own work except where otherwise indicated. Passages and ideas from other sources have been clearly indicated. To date, neither this diploma thesis nor essential parts thereof were subject of an examination procedure. Until now, I don't have published this diploma thesis or parts thereof. The electronic copy is identical to the submitted copy.

Stuttgart, April 7, 2014, .....

(Lukas Reinfurt)

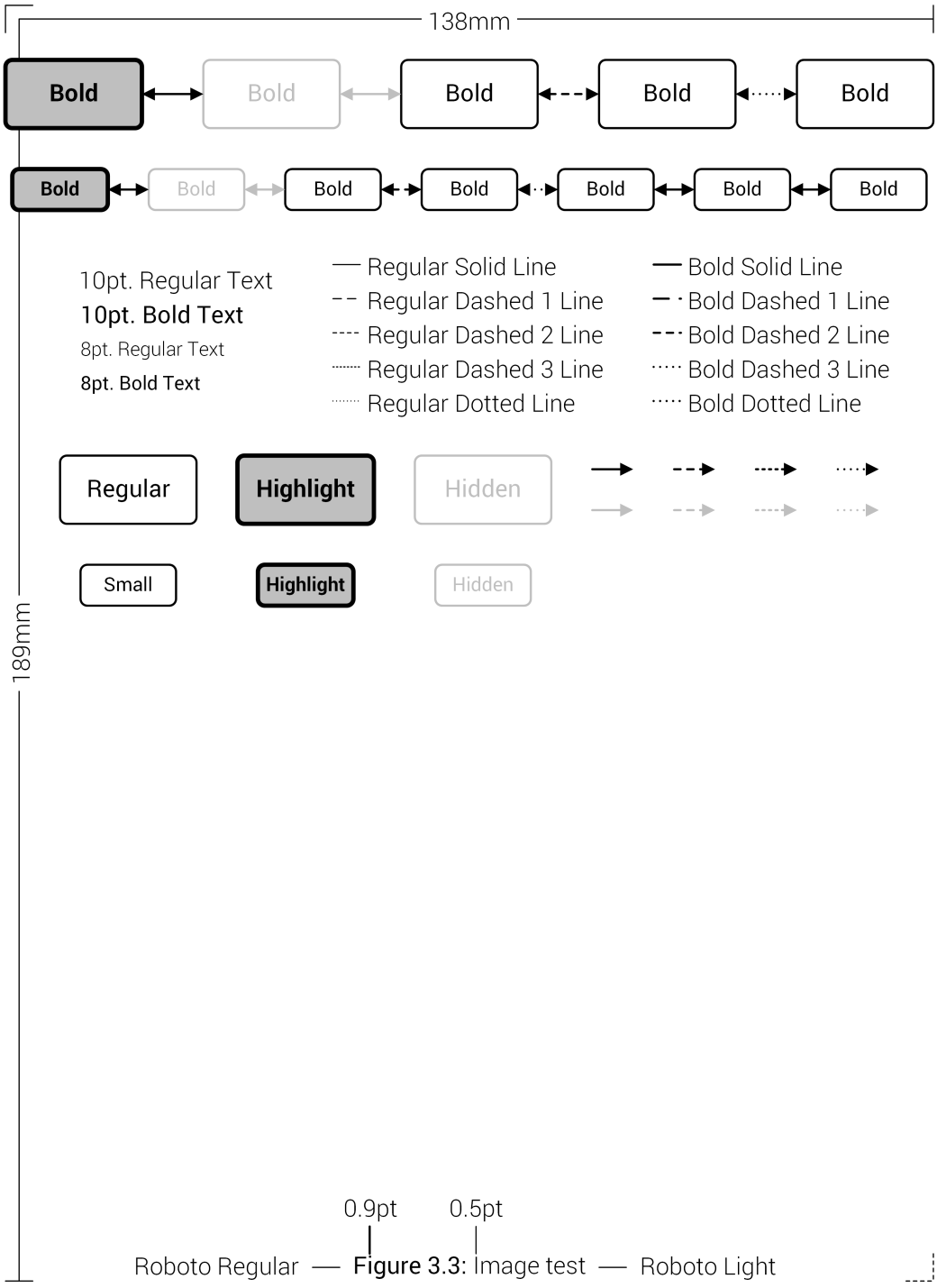


Figure 6.1: Image test