Institute of Architecture of Application Systems

Flexible Process-based Applications in Hybrid Clouds

Christoph Fehling[1], Ralf Konrad[2], Frank Leymann[1],
Ralph Mietzner[1], Michael Pauly[2], David Schumm[1]

[1] Institute of Architecture
of Application Systems,
University of Stuttgart, Germany
{fehling, leymann, schumm}
@iaas.uni-stuttgart.de

[2] T-Systems International GmbH,
Frankfurt, Germany

{ralf.konrad, ralph.mietzner, michael.pauly}
@t-systems.com

Universität Stuttgart
Germany

Flexible Process-based Applications in Hybrid Clouds

Christoph Fehling, Frank Leymann,
David Schumm
Institute of Architecture of Application Systems
University of Stuttgart
Stuttgart, Germany
{fehling, leymann, schumm}@iaas.uni-stuttgart.de

Ralf Konrad, Ralph Mietzner,
Michael Pauly
T-Systems International GmbH
Frankfurt, Germany
{ralf.konrad, ralph.mietzner, michael.pauly}
@t-systems.com

Abstract— **Cloud applications target large customer groups to leverage economies of scale. To increa se the number of customers, a flexible application design is of major importance. It enables customers to adjust the app lication to their individual needs in a self-service manner. In this pap er, we classify the required variability of these flexible applications: data variability – changes to hand led data structures; functional variability – changes to the processes that the application supports; user interface variability – changes to the appearance of the application; provisioning variability – the ability of the a pplication to be deployed in different runtime environments. Existing and new technologies and tools are leveraged to realize these classes of variability. Further, we cover architectural principles to follow during the design of flexible cloud applications and we introduce an abstract architectural pattern to enable data variability.**

Keywords- application customization, self-service, orchestration, composite application, provisioning, cloud

I. INTRODUCTION

Today, many cloud applications handle large numbers of customers, whose versatile demands on resources and application performance have to be met in an effective and timely manner. Flexibility regarding application behavior, user interface appearance, and resource provisioning is therefore of major importance for competitiveness of application developers and cloud infrastructure providers.

Prior to the establishment of cloud computing, hardware virtualization has introduced flexibility to hardware management. It forms the basis for on-demand use of cloud resources. Due to hardware virtualization, resources are no longer bound to physical servers. Therefore, their provisioning can be automated and is offered via management interfaces to cloud users. Recently, significant effort was made by the industry to standardize these interfaces [1], [2]. This automation and standardization has lead to the additional properties of computing clouds, which differentiate them from pure virtualization environments. These properties are: Elasticity – cloud resources can be reserved and freed on demand; Pay-per-use – users only pay for the currently required resources; Standardization – management functionality is accessible via standardized interfaces.

Many cloud offerings [3], [4], [5], [6] also brought this flexibility of the infrastructure to the application level. Configurable applications are offered to customers (referred to as "Software as a Service", SaaS). Configurable platform services can be used by customers to develop and execute custom applications (referred to as "Platform as a Service", PaaS). Also, there are offerings that allow users to flexibly compose individual services and platform services into custom applications (referred to as "Composition as a Service", CaaS [7]). In this scope, flexibility is enabled by two types of compositions [8]. "Horizontal composition" refers to the orchestration of services themselves. "Vertical composition" refers to the combination of service implementation, required middleware and runtime environments (especially, different clouds).

In this paper, we classify the variability introduced by horizontal and vertical composition of services and we introduce architectural principles and techniques to create flexible cloud applications. Further, we show how flexibility of the underlying infrastructure can be supported on other application layers. Application architectures following the presented approach allow a flexible (re)composition of their components, which enables reacting to different or changing business demands. An abstract process-based view is introduced on which this (re)composition can be performed. Additionally, we cover techniques to automatically transform component orchestrations to an executable form, which can be provisioned to clouds. Especially, application components can be distributed among different clouds forming a so-called hybrid cloud. This distribution can be specified individually for each customer. For example, one customer may decide to provision all application components in a public cloud, while another customer requires certain application components to be hosted in his private cloud due to security requirements.

As a consequence, the contributions of this paper are (i) a classification of different variability required in flexible cloud applications, (ii) architectural principles to enable configurability and flexibility in applications based on processes and service compositions, and (iii) a framework handling the flexible user-centric (re)composition of application components and their provisioning in a hybrid cloud environment.
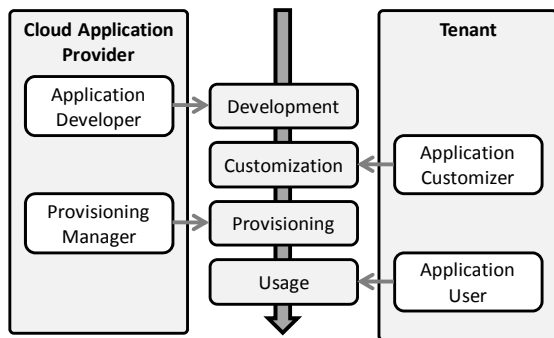
Figure 1: The Cloud Application Offering Process and Stakeholders

To describe the different phases that a cloud application undergoes as well as to introduce associated stakeholders, we propose the *cloud application offering process* depicted in Figure 1. The *cloud application provider* offers a customizable application to multiple *tenants*. Each tenant forms an individual institution, such as a company, that has multiple *application users* associated with it. These users access the application on behalf of the tenant. During the *development phase*, configurable applications are implemented by an application developer. An *application customizer* adjusts such an application to the individual requirements of a tenant during the *customization phase*. This is usually done via a self-service portal that is offered by the cloud application provider. Via this portal, a tenant may also sign-up for the service, specify billing information, manage application user rights etc. After the customization phase, an *infrastructure manager* of the application provider handles the *provisioning phase* of the application. In this phase, the customized application is transformed into an executable form and provisioned specifically for a tenant. Afterwards, application users of that tenant may start using the application during the *usage phase*.

The paper's further structure respects the phases of the cloud application offering process and is the following: Section II describes the motivating use case based on an application that is productively used by T-Systems, the ICT subsidiary of Deutsche Telekom [9]. Section III classifies the different forms of variability required by customers of these flexible applications and describes how to enable them during the application development phase. Section IV introduces a framework used during the subsequent application customization phase and application provisioning phase. Section V discusses an evaluation of the provided framework to handle this application customization and provisioning in a hybrid cloud setup. Section VI covers relevant related work. Finally, Section VII gives a summary and an outlook of future research challenges.

II. MOTIVATING USE CASE

T-Systems provides the *Process Service Platform (PSP)* as a PaaS offering to customers. Using this platform, customers may deploy customized processes and services, while the platform provides the necessary runtime environment as well as additional platform services for monitoring, billing, and access control. The architecture of the PSP is also described in [10].

Especially, the PSP is used by T-Systems to realize SaaS offerings that shall be individually adjusted to customer requirements. The offering considered here, provides management software for the public administration of the city of Friedrichshafen. It is part of the "Smart City" pilot project, T-City [11]. Using this software, processes of public administrations are integrated to provide easier access for inhabitants and increase the quality of the services provided to them. The specific integration process considered in this use case informs inhabitants of Friedrichshafen, which have children of a certain age, about available places in kindergartens. In the following, this process is referred to as the *Kindergarten Information Process*. Prior to integration of information sources, information about children was distributed among different institutions. Therefore, parents tried to sign up to multiple kindergartens in order to increase their chances of having a place assigned to their child. Capacity planning for the kindergartens of Friedrichshafen was significantly hindered by this situation, because the actual demand was obfuscated. Also, an earlier process to inform parents about free kindergarten places could not access information handled by kindergartens or schools to exclude children that had been placed already. This resulted in a lot of redundant communication and unnecessary information being sent out to parents.

As depicted in Figure 2, the new PSP-based process integrates the different information services associated with the city center (information about inhabitants), the kindergartens (information about available places), and local schools (information about children that are already going to school). Access to all this data is required to determine the families to inform about free kindergarten places in their area.
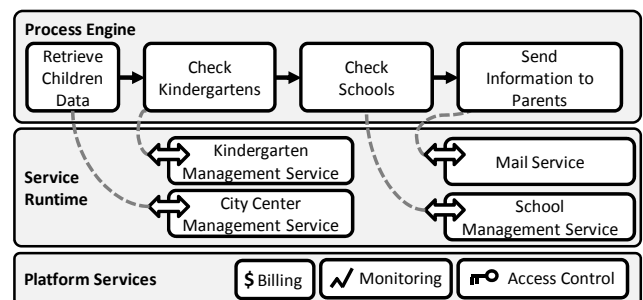

Figure 2: Overview of the Process Service Platform (PSP) in the considered Kindergarten Information Process

After the pilot project, the management software is supposed to be provided as a SaaS offering to other cities (i.e., tenants) as well. However, due to different local laws and different management structures, the requirements of other cities differ significantly. For example, different requirements on security, privacy, and trust can hinder the hosting of certain data on the provider side. Further, the data structure that the software needs to deal with may differ. For example, cities in the German local state "North Rhine-Westphalia" would require additional information about

children's results of a language test to be stored. This information would also have to be considered during the assignment of kindergarten places to provide children with adequate care. Additionally, different customers may demand different media to communicate with parents and kindergartens, like letters, e-mails, or cell-phone text messages.

III. DEVELOPMENT OF FLEXIBLE CLOUD APPLICATIONS

In general, we propose to divide the variability required by flexible SaaS applications into four classes. *Data variability* guarantees that the software can handle additional data fields and is flexible regarding the schema of handled data. In the use case, the information about language test results can be added to a variable data element after implementation. *Functional variability* refers to the capability of the application to activate, deactivate, or replace certain steps in supported processes. Additionally, the user-centric definition of completely custom processes could be desirable. For example, the Kindergarten Information Process can allow replacement of the activity informing the parents to support different communication media. *Provisioning variability* considers how the application can be distributed among different computing environments to fulfill different privacy, security, and trust requirements. For example, the software considered in the use case shall support that certain services are hosted in the customer's private data center. This is mostly to comply with local laws prohibiting public administrations to outsource inhabitants' personal data. In other fields, such as healthcare, similar challenges arise [10]. *User interface variability* describes how the user interface of an application can be adjusted to customers' demands or how individual user interfaces can be developed for a particular customer. In our use case, this variability demanded adjustment of the user interface language, its color, and graphics.

Often, the desired flexibilities must be considered in the applications architecture. In the following we therefore describe architectural principles to follow during the development of flexible application components, as well as a user-centric, process based orchestration and application configuration. One of the main challenges during the development of the Kindergarten Information Process arose from the fact that many process modeling languages used today are focusing either on modeling or on execution [12], [13]. In our case, customers without programming knowledge had to be enabled to orchestrate pre-determined application components or alter reference orchestrations in a fast and flexible manner. A manual transformation from modeling language to execution language was unsuitable for this task. Therefore, we abstracted from predefined execution language constructs towards graphical elements that are orchestrated. This lead to a tight alignment of modeling language and execution language and enabled an automatic transformation. In the following, we introduce the proposed structure of application components and cover how the different classes of variability are supported.

A. General Structure of Application Components

Componentization and loose-coupling of application components ensure the ability of the composite application to scale-out and to ensure high availability in a cloud environment. This is covered in greater detail in Section VI (background and related work).
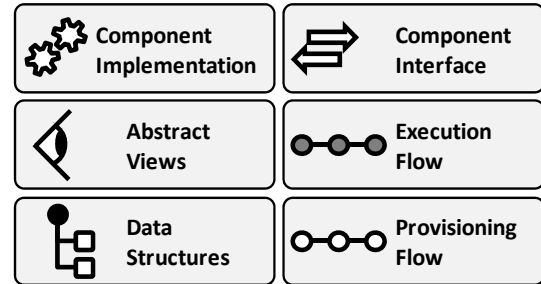

Figure 3: Artifacts that constitute an Application Component

Figure 3 depicts the artifacts that constitute an *application component* of such a componentized application. In scope of the motivating use case, such application components map to services that are deployed on the PSP. The *component implementation* contains the concrete software artifacts that implement the desired functionality. The *component interface* describes how this functionality can be accessed. We used WSDL [14] in the use case, but other interface description languages could also be used. This is mainly influenced by the orchestration language that is employed to orchestrate applications components. For the Kindergarten Information Process, we used BPEL [15] to orchestrate application components. These orchestrations of application components can again be used to form new application components. The *abstract views* of an application component describe its graphical representation to be used during the orchestration. Multiple abstract views can be defined by the application developer. Each abstract view of an application component is made up of (i) an icon used for its graphical representation, (ii) connection points used during graphical orchestration, and (iii) a description of the offered function. Each abstract view is associated with an *execution flow*. This flow subsumes the required execution language constructs needed to access the application component's functions via its interface. It is used during the transformation of the orchestration to an executable form, which is covered in detail in Section IV.B. Additional information described by the *data structures* allows users to customize the data elements handled by an application component. In the use case, this was employed to enable addition of the language test results to children data elements. Finally, each component has a *provisioning flow* that describes the required tasks to configure and provision the component in different environments. How application components are provisioned according to provisioning flows is covered in Section IV.C. In the following, it is described, how the application component artifacts and the architecture of its implementation are used to enable the different classes of variability.

B. Data Variability

Data variability has to be respected by the application component's implementation (handling of variable data elements) as well as by its interface (generic access to data elements). We identified the architectural specifics required for this component behavior as the *variable data component pattern* depicted in Figure 4. Implementation of this architectural pattern enables a variable structure of handled data elements and generic data querying capabilities that are offered by the application component. Data elements can be extended by additional fields and a generic data manipulating interface can be used to handle completely new data elements.
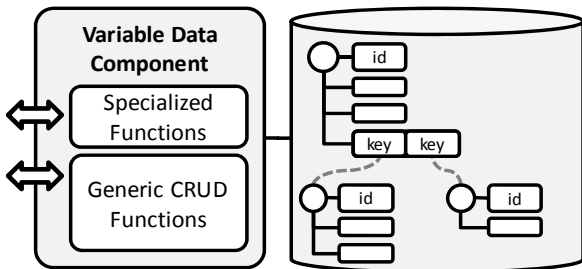

Figure 4: Conceptual Architecture of the Data-flexible Component

The extensibility of data elements is achieved by each data element being associated with an untyped list of further data elements. This allows additional data elements to be associated with existing ones and does not require the structure or type of the new ones to be known during design time of the application. Each entry in this list is identified via a key. Regarding the motivating use case, such a data structure was used to store additional information about language test results associated with children data elements.

In case new data elements shall also be queried directly and not via existing functions, additional generic data manipulating functions can be implemented. These are used to create, read, update, and delete data elements managed by the application component. These are the CRUD functions [16], which are, for example, used by the REST architecture [17]. If this generic data access is desired, additional data elements are identified by a unique identifier (id). This id is passed to the generic functions as a parameter to identify the data element to manipulate. A user is enabled to query and manipulate arbitrary data elements for which special querying functions were unknown during design time.

The drawback of such variable data elements and generic querying interfaces is that a lot of the application functionality is now hidden behind generic interfaces. The readability of such interfaces is therefore drastically reduced. Also, newly required functionality is not implemented in the associated application component but on the orchestration layer, which increases the complexity of the orchestrations. Additionally, the optimization of query execution, database structure, and database partitioning is hindered. This can drastically affect the performance and scaling behavior of customized data components. In contrast to relational databases, this approach moves away from a restricting database schema. It is employed by many databases that emerged in the area of cloud computing. These so-called NoSQL databases are used to serve users with arbitrary requirements on data structures and employ the elasticity of clouds to handle performance issues. Queries are distributed among many compute nodes using a mechanism called Map Reduce [18].

Due to these restrictions, variability has to be weighed against interface readability, performance, and orchestration complexity. While the first two will only affect the application provider, complexity of reference orchestrations hinder customers during application customization. In [19] an approach is given how to make the correct architectural decisions in this scope.

The variable data component pattern is implicitly used by many cloud providers, such as Saleforce.com [3] to customize its CRM software. A similar approach is also employed in XML via the xs:any-tag [20].

C. Provisioning Variability

Application components are provisioned individually for integration into orchestrations that are customized by tenants. When an application component is provisioned, its provisioning flow is being executed. This flow describes automated and manual tasks required to provision the component. For example, this can include starting a virtual machine with a certain operating system, installing required middleware, and deploying the application components on it.

To enable provisioning variability, we defined multiple branches in execution flows of application components to describe different alternatives how the application component may be provisioned. During the provisioning, an application customizer is asked which alternative is preferred. This way, an application component can be provisioned in different environments, such as the PSP, Amazon EC2 [21], or even within the users' private data centers. In the corresponding branches of the application component's provisioning flow, activities are contained which perform an upload of the application component to the PSP, instantiate an Amazon virtual machine image with the required application stack, or ship the component on CD to be installed in users' private data centers.

D. Functional Variability

To enable functional variability, the application components may be orchestrated in a process-based view. In this view, the elementary orchestration language constructs are made available to application customizers while complexity introduced by service invocation, service addressing, variable initialization and variable assignment is hidden. This additional complexity is addressed in the application component's execution flow, which is used during the generation of executable orchestrations. This generation is described in Section IV.B.

The elementary elements used for our process-based application orchestration are mainly influenced by BPEL. Directed control connectors connect any two graphical elements in the orchestration view except other control connectors. Each orchestration has exactly one start element

and exactly one end element. The start element may not have any incoming control connectors. The end element may not have any outgoing ones. Abstract views of application components can be freely interconnected using control connectors or may be part of a sequence. Such a sequence element specifies activities that are executed one after another. Branches can be expressed using an if-element. Additionally, a for-each element enables the iteration of lists of data elements.

E. User Interface Variability

Similar to provisioning variability, variability of the user interface in handled in the provisioning flow of the application components that constitutes the user interface. For example, an application customizer can be asked to specify desired UI colors, logos, or the language to be used. In case of rich clients, the UI can be shipped to applications users and can then be customized in the same form as other desktop applications.

IV. CUSTOMIZATION AND PROVISIONING FRAMEWORK

In this section, we describe the framework providing technologies and tools used to customize and provision flexible cloud applications. The components of the framework are depicted in Figure 5.
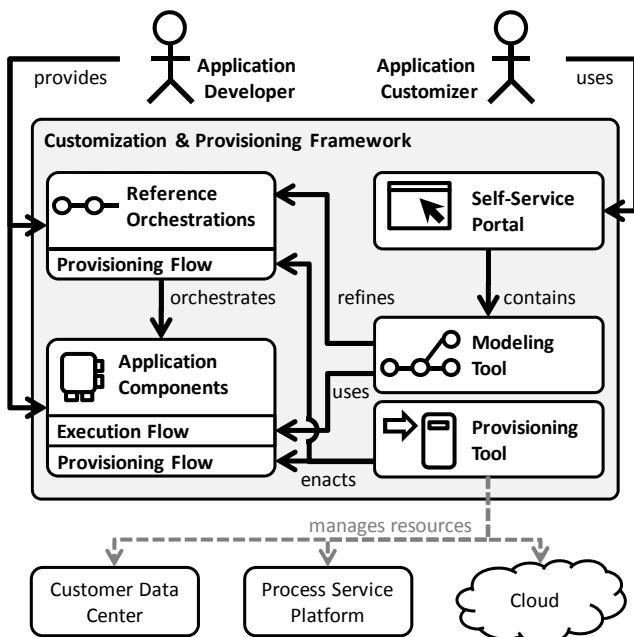

Figure 5: Components of the Customization & Provisioning Framework

The application developer provides reference orchestrations and application components. Application components are orchestrated by reference orchestrations to provide certain functionality, like the Kindergarten Information Process, to tenants. The application customizer uses a modeling tool as part of a self-service portal to refine the reference orchestrations. The modeling tool uses the application components to offer available modeling elements

to the application customizer. After the reference orchestration has been customized, the modeling tool uses the execution flows of application components to generate executable orchestrations that can be provisioned on the PSP. This is handled by the provisioning tool that enacts provisioning flows of the application components to deploy them on cloud resources and other infrastructure resources. Since the reference orchestrations are also application components, their provisioning to the PSP is performed in the same manner. The provisioning tool further integrates the resource management of different computing environments, like customer data centers, the PSP, and different clouds. Due to this integration, the different environments are perceived as one hybrid cloud.

In the following, the phases of application customization, executable orchestration generation, and application provisioning are covered in more detail and the used tools are described.

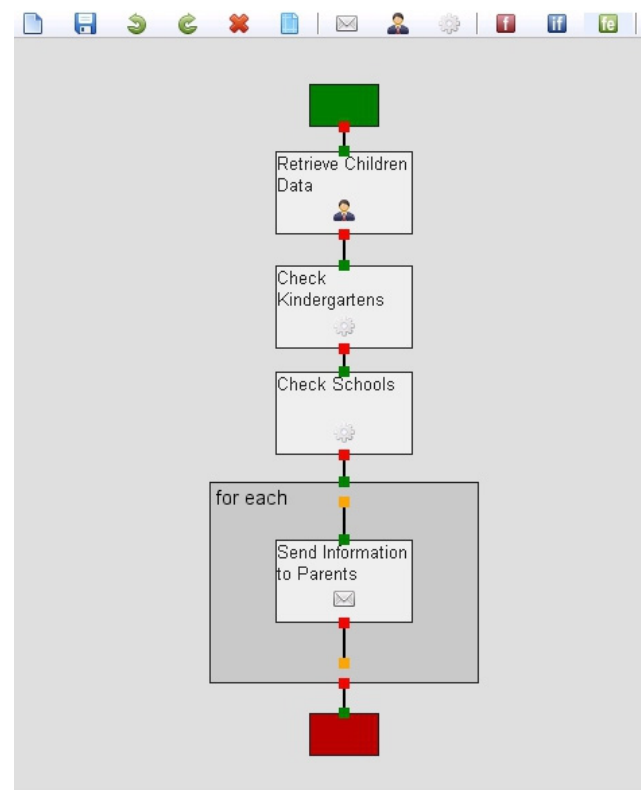### A. Application Customization

Figure 6: Screenshot of the DecidR+ Modeling Tool

The application developer provides application components and reference orchestrations that orchestrate them. These reference orchestrations may be incomplete. For example, the reference Kindergarten Information Process may lack an application component that sends information to parents. The application customizer may then use the modeling tool of the framework to add a component that provides e-mail services, postal services, or cellular text messaging services, to complete the process.

Both, the self-service portal and the modeling tool have been implemented and are available as open source, namely as the DecidR+ tool [22]. Individual abstract views have been created for E-mail interaction as well as for human tasks. E-mail and human task functionality is needed in the motivating use case, when data required in the orchestrations cannot be accessed via Web services. E-Mail interaction allows E-mails to be sent to E-mail addresses. Optionally, these interactions may then provide input data for the process in form of a regular E-mail response. Human tasks allow a similar integration of human beings via a task manager. The tool further supports the orchestration of arbitrary Web services that are integrated using a common abstract view. When the common view is added, a WSDL file may be specified for the service to be accessed. This common abstract view has been used to access some of the services integrated by the Kindergarten Information Process, which is depicted in Figure 6.

In this particular customization, the data of children is retrieved via a human task. Then, kindergartens and schools are checked to exclude children that already have places in these institutions. For each child that was not excluded, the information regarding available kindergarten places is sent to parents via e-mail.

B. Generation of Executable Orchestrations

Each elementary orchestration language element is mapped to constructs of an executable language to allow generation of executable orchestrations. In our implementation, the DecidR+ tool, BPEL was used for execution as depicted exemplarily in Figure 7. The BPEL constructs access application functionality via applications component interfaces. After the modeling, the orchestration of application components is automatically transformed into BPEL code that can be deployed on the PSP. This form is partially incomplete, because the BPEL process as well as the application component interfaces may contain *variability points* [23]. These describe additionally required information, such as addresses, unknown during application customization that becomes available during application provisioning.
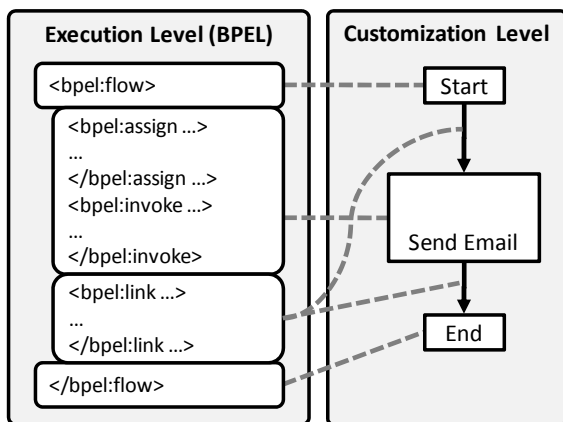

Figure 7: Mapping between Execution Level and Customization Level

Other execution languages could also be supported. The presented approach can even enable the orchestration of non-distributed applications. For example, the component interface could be described using a JAVA [24] interface specification. If a mapping of abstract view elements to JAVA is specified, the orchestration of application components would result in the creation of stand-alone JAVA application.

C. Application Provisioning

After high level customization and subsequent generation of an executable orchestration, the customized application may be provisioned for a customer. To do so, we used an existing provisioning tool, Cafe [25], that enacts the individual provisioning flows of application components. During this process, the variability points of these components become known, especially, their addresses in clouds. Therefore, the corresponding variability points of the BPEL process are set to these addresses and the process is provisioned on the PSP. In the same way, variability points of the user interface, which initiates the BPEL process, are handled. The application customizer can also be integrated via human tasks in which he may select alternative runtime environments to provision application components. Through the integration of human tasks, the provisioning of required infrastructure or application components may also be handled by the application customizer himself. Cafe's concepts and techniques to handle the flexible provisioning of applications is described in further detail by [26], [27], and [28].

V. EVALUATION IN A HYBRID CLOUD SETUP

We have used the provisioning tool Cafe to model application components and their different classes of variability. During the customization phase, this enables tenants to individually distribute application components among computing environments forming a hybrid cloud. For each of the application components the different provisioning alternatives were modeled in their provisioning flows. During their registration to the application, customers can decide to provision the school management service, for example, in their private data center or on an infrastructure offered by the provider. In the provisioning flows, the tasks required to achieve this are fully automated on the provider's side. If the provider does not have direct access to the computing environment, for example to a customer's data center, human tasks are included in the provisioning process to be performed by the customer. The application provider only offers downloads for the application components to the customer that can be installed manually.

We have integrated Amazon EC2 as a public environment, a cloud offered by T-Systems that also hosts the PSP itself, and the above mentioned manual provisioning for private data centers. Each customer may specify his own distribution of application components in this hybrid cloud setup. Exemplary distributions of application components are depicted in Figure 8. In this example, Tenant 1 decides to use some of the shared services offered by the provider. He uses the provisioning flexibility of other services so they are

provisioned as separate instances for him on Amazon EC2. This may be due to legal requirements demanding that some of his data may not be stored on hardware that is shared with others. Tenant 2 has even greater security requirements and decides to host some of the services in his private data center.
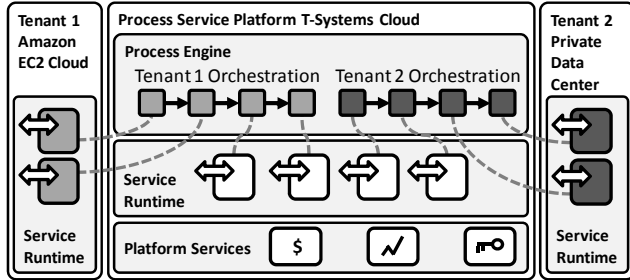

Figure 8: Exemplary Application Component Distributions

VI. Background and Related Work

According to [29] and [30], cloud applications rely on modular architectures, loose coupling of application components, statelessness, and asynchronous communication. These concepts are also employed in the approach presented here. They enable cloud applications to benefit from a cloud's elasticity, its pay-as-you-go pricing models, and the standardization of its management interfaces. In detail, modular architectures allow individual application components to scale independently and thus maximize the beneficial effects of the elastic cloud infrastructure. Loose coupling and statelessness eases these scaling processes and ensures that failing application components do not affect others.

The presented variability of cloud applications introduced in Section III is of vital importance to increase the addressable customer market, because varying customer requirements can be met more easily. A large customer group enables the application provider to leverage economies of scale better, because resources can be shared between more customers. The concept has been introduced as "catching the long tail" by [31]. The sharing of resources between multiple tenants is another important concept of cloud computing. Tenants may however have different requirements regarding the resources they may share with others [32]. These different requirements are covered in the presented approach by the provisioning variability. During the provisioning of application components, a tenant may select if the component can be shared with other tenants or shall be provisioned individually. Optimization of tenants' user distributions among available application component instances, while respecting their requirements on multi-tenancy and service levels, is described by [34].

The covered provisioning variability further enables the distribution of application components among different clouds. Today, applications of companies have versatile requirements on privacy, security, and trust. It is unlikely that a complete application landscape can be moved to one distinct cloud environment. Therefore, companies face the challenge of distributing their applications among different computing environments and integrating them afterwards to form a hybrid cloud [7], while secure access between components is enabled [33]. The presented approach enables a fine-grained and customer-specific distribution of applications and their components among multiple environments of such a hybrid cloud.

The variable data component pattern covered in Section III.B describes concepts that are widely used implicitly, which is why we suggest the abstraction of these concepts to a generic architectural pattern. Examples are NoSQL databases [35], like Apache CouchDB [36] or Amazon SimpleDB [37], that reduce data consistency [38] and do not support database schemas or only very rudimentary ones. Often these databases are queried using Map Reduce [18], which distributes the query load among many cloud resources and consolidates the results afterwards. This enables them to scale-out extremely well using distributed resources and allows tenants with versatile requirements on data structure to share a database.

The presented process-based orchestration of services is going to be a significant architectural principle in cloud applications. While holistic applications realize similar functionality by implementing the model-view-controller pattern [39], [40] motivates that processes, views, and services are going to play a similar role in distributed applications.

VII. Summary and Outlook

New cloud computing technologies and architectural principles lead to flexibility being introduced to cloud infrastructures (dynamic provisioning of customized virtual servers), the cloud application architectures (loose coupling, statelessness), as well as to used middleware (for example, NoSQL databases). Using the presented approach, this flexibility is also introduced to the application development processes and customization processes, to create applications in a flexible and user-centric form.

We classified the desired variability of flexible cloud applications and described how to enable them using certain architectural principles, techniques and tools. Users are now able to create individualized composite applications using a self-service portal. Application providers may use the presented framework to offer application components and referential orchestrations thereof to customers. The provisioning of application components is adjusted individually. This results in customer specific application component distribution among different cloud, especially to support hybrid cloud environments.

The next step is, to bring the flexibility introduced to application design and provisioning to the runtime management of cloud applications. To exploit pay-per-use pricing models even better, application users would then be enabled to suspend complete applications, for example. Another management task would be the redistribution of application components among different environments in hybrid clouds. The need for this arises if customers' requirements on privacy, security, trust, performance,

availability etc. change after the initial provisioning of the customized cloud application.

REFERENCES

[1] Distributed Management Taskforce (DMTF). Interoperable Clouds Whitepaper, 2011.

[2] Storage Networking Industry Association (SNIA). Cloud Data Management Interface (CDMI) Whitepaper, 2010. http://www.snia.org/tech_activities/standards/curr_standards/cdmi/CDMI_SNIA_Architecture_v1.0.pdf

[3] Salesforce. Small Business CRM and Contact Manager, 2011. http://www.salesforce.com/smallbusinesscenter/

[4] Amazon. Amazon Web Services, 2011. http://aws.amazon.com/

[5] RunMyProcess. BPMN Designer, 2011. http://www.runmyprocess.com/

[6] Cordys. Process Factory, 2010. http://www.cordysprocessfactory.com/

[7] F. Leymann. Cloud Computing: The Next Revolution in IT. Proceedings of the 52th Photogrammetric Week, 2009. http://www.ifp.uni-stuttgart.de/publications/phowo09/010Leymann.pdf

[8] R. Mietzner, F. Leymann, T. Unger. Horizontal and Vertical Combination of Multi-Tenancy Patterns in Service-Oriented Applications. Enterprise Distributed Object Computing Conference (EDOC), 2009.

[9] Deutsche Telekom. T-Systems, 2011. http://t-systems.com/

[10] I. Brandic, S. Dustdar, T. Anstett, D. Schumm, F. Leymann, R. Konrad. Compliant Cloud Computing (C3): Architecture and Language Support for User-Driven Compliance Management in Clouds, IEEE 3rd International Conference on Cloud Computing, 2010.

[11] Deutsche Telekom. T-City Friedrichshafen, 2011. http://www.t-city.de/

[12] O. Kopp, D. Martin, D. Wutke, F. Leymann. The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. Enterprise Modelling and Information Systems Architecture, 2009.

[13] N. Palmer. Understanding the BPMN-XPDL-BPEL Value Chain, Business Integration Journal, 2006.

[14] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0, 2007. http://www.w3.org/TR/wsdl20/

[15] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution Language Version 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[16] C. Henderson. Building Scalable Web Sites, O'Reilly, 2006.

[17] L. Richardson, S. Ruby. RESTful Web Services, O'Reilly, 2007.

[18] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters, Communications of the ACM, 2008.

[19] C. Pautasso, O. Zimmermann, F. Leymann. Restful Web Services vs. "Big" Web Services: Making the Right Architectural Decision, 17th International Conference on World Wide Web, 2008.

[20] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008. http://www.w3.org/TR/2008/REC-xml-20081126/

[21] Amazon. Amazon Elastic Compute Cloud (EC2), 2011. http://aws.amazon.com/ec2/

[22] DecidR+ project, 2011. http://www.decidrplus.de

[23] R. Mietzner, F. Leymann. A Self-Service Portal for Service-Based Applications, IEEE International Conference on Service-Oriented Computing and Applications (SOCA), 2010.

[24] Oracle. Java Technology, 2011. http://www.oracle.com/java/

[25] R. Mietzner. Cafe Project Site, 2011. http://www.cloudy-apps.com

[26] R. Mietzner, F. Leymann. Generation of BPEL customization processes for SaaS applications from variability descriptors, IEEE International Conference on Services Computing (SCC), 2008.

[27] R. Mietzner, F. Leymann, M. P. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and SaaS Multi-Tenancy Patterns, 3rd International Conference on Internet and Web Applications and Services (ICIW), 2008.

[28] R. Mietzner. A method and implementation to define and provision variable composite applications, and its usage in cloud computing, Ph.D. Thesis, 2010. http://elib.uni-stuttgart.de/opus/volltexte/2010/5614/

[29] J. Varia. Architecting for the Cloud: Best Practices. Technical Report, Amazon, 2010. http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf

[30] J. Varia. Cloud Architectures. Technical Report, Amazon, 2010. http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf

[31] F. Chong, G. Carraro. Architecture strategies for catching the long tail. MSDN Library, Microsoft Corporation, 2006. http://msdn.microsoft.com/en-us/library/aa479069.aspx

[32] R. Mietzner, T. Unger, R. Titze, F. Leymann. Combining different multi-tenancy patterns in service-oriented applications. Enterprise Distributed Object Computing Conference (EDOC), 2009.

[33] Spillner J. Privacy-enhanced Service Execution. Westnik DUIKT-Proceedings of the International Conference for Modern Information and Telecommunication Technologies, 2008.

[34] C. Fehling, F. Leymann, R. Mietzner. A Framework for Optimized Distribution of Tenants in Cloud Applications. IEEE 3rd International Conference on Cloud Computing, 2010.

[35] R. Cattell. Scalable SQL and NoSQL Data Stores, 2011. http://www.cattell.net/datastores/Datastores.pdf

[36] The Apache Software Foundation. CouchDB, 2011. http://couchdb.apache.org/

[37] Amazon. Amazon SimpleDB, 2011. http://aws.amazon.com/simpledb/

[38] W. Vogels. Eventually consistent. Communications of the ACM, 2009. http://queue.acm.org/detail.cfm?id=1466448

[39] F. Buschmann, K. Henney, D.C. Schmidt. Pattern-oriented Software Architecture. A System of Patterns, Addison-Wesley, 1998.

[40] R. Heffner. Process-Views-Services: A Better Design Paradigm For Applications. Forrester Research, 2008.