An Architectural Pattern Language of Cloud-based Applications

Christoph Fehling, Frank Leymann,
Ralph Retter, David Schumm
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstr. 38, 70563 Stuttgart, Germany
{fehling, leymann, retter, schumm}
@iaas.uni-stuttgart.de

Walter Schupeck

Daimler AG
Epplestr. 225,70546 Stuttgart, Germany
walter.schupeck@daimler.com

ABSTRACT
The properties of clouds – elasticity, pay-per-use, and standardization of the runtime infrastructure – enable cloud providers and users alike to benefit from economies of scale, faster provisioning times, and reduced runtime costs. However, to achieve these benefits, application architects and developers have to respect the characteristics of the cloud environment.

To reduce the complexity of cloud application architectures, we propose a pattern-based approach for their design and development. We defined a pattern format to describe the principles of cloud computing, available cloud offerings, and cloud application architectures. Based on this format we developed an architectural pattern language of cloud-based applications. Through interrelation of patterns for cloud offering descriptions and cloud application architectures, developers are guided during the identification of cloud environments and architecture patterns applicable to their problems. We cover the proceeding how we identified patterns, give an overview of previously discovered patterns, and introduce one new pattern. Further, we propose a framework for the organizations of patterns and the guidance of developers during their instantiation.

Categories and Subject Descriptors
C.2.4 [Computer System Organization]: Computer-Communication Networks – *distributed systems*. D.2.11 [Software]: Software Architectures – *patterns*. D.2.2 [Software]: Software Engineering – *design tools and techniques, decision tables.*

General Terms
Management, Documentation, Performance, Design, Reliability, Standardization, Languages.

Keywords
Cloud Computing, Architecture, Patterns, Decision Table.

1. INTRODUCTION
Cloud computing has significantly changed the way in which IT resources can be used. Required resources can be reserved on-demand and freed when no longer needed. They are billed on a pay-as-you-go basis [24]. Further, it currently seems that the use of cloud technologies leads to a standardization of software stacks [34]: ready-to-use machine images and services are either offered by the cloud provider or created by application developers for reuse in multiple applications and environment configurations. The most significant properties of clouds can be summarized as follows [21] [38]: *elasticity* – the number of resources available to an application can be flexibly adjusted to fit with the current demand; *pay-per-use* – resource costs are billed based on the time interval during which they are used; *standardization* – the use of hardware virtualization and provider-supplied platform services standardizes runtime environments of applications in the cloud.

To use the full capacity of such a powerful environment, a cloud application developer, however, needs to incorporate certain *architectural principles* and functionality in cloud-based applications. A simple migration of an existing application running on a single machine to the cloud only results in minor benefits and may even reduce the applications availability, if architectural principles are not followed properly [44]. Instead, the elasticity and pay-per-use pricing models of clouds benefit the application developer best, if resources are distributed and their management is automated. This automation has to be enabled in the application itself or by using management functionality of the provider. Therefore, application developers need to employ the management interfaces of the used clouds to start and stop machines, monitor resource usage and access functionality supplied by the provider. Even though there is significant effort made by the industry to standardize these cloud management interfaces [11] [39] [19] [20], the usage of provider-specific functionality often leads to a tight coupling of the developed application to a specific cloud provider. The industry driven evolution of cloud interfaces and functionality also leads to cloud offerings being hard to compare. Their common underlying concepts are often obfuscated or remain non-public. As a consequence, cloud application development is difficult in requirements management and is bound to a specific cloud provider, e.g., development for Windows Azure [29] differs from development for Google's AppEngine [7] or Amazon Web Services (AWS) [43].

The problem to make cloud providers interchangeable cannot be solved in general, because offered platforms comprising services and runtime infrastructures differ in functionality and supported programming languages, for example. However, *patterns* of frequently applied practice can help to describe reoccurring requirements of applications and guide developers during the creation of such applications. In this scope, we contribute a proceeding how to discover patterns in different information sources. In addition to describing reoccurring good solutions in a pattern format, we propose to also provide abstract descriptions of frequent requirements in the format as well. We exemplify the proposed pattern format in one new pattern and give an overview of other patterns we discovered. A catalog of these existing patterns that we identified during the collaboration with Daimler AG, a large German car manufacturer, is given in [13] and is available online[1]. We used the pattern format to describe (i) different cloud types, (ii) the resources they offer, (iii) the way in which they offer these resources, and finally (iv) how to build cloud applications on top of cloud resources. We captured interrelations between patterns in a decision recommendation table that guides application developers during the identification of applicable patterns. Further, a framework is proposed that supports tasks for pattern organization, identification, and automated instantiation.

We claim that the use of a pattern format beyond good solutions for frequent problems eases requirements management, while the introduced framework enforces the standardization of the application development and application runtimes. This standardization is required to reduce the management complexity of a company's application landscape [36], which is a major cost driver for IT infrastructures [18]. Further, a pattern catalog may be used to document architectures of developed applications in a more standardized manner and teach cloud computing concepts to developers [16].

The paper's further structure is the following: Section 2 gives an overview of the research design we employed for pattern identification. In Section 3, we introduce the format for cloud application architecture patterns and cover one new pattern. Further, this section gives a summary of the catalog of patterns that we compiled for cloud types, cloud service models, and cloud application architectures using the described research design. Based on this catalog, Section 4 defines a pattern-driven application development method for cloud applications and proposes a framework to guide developers during this process. Section 5 concludes the paper by summarizing the aspects covered in the paper and provides an outlook on future research in the field.

2. RESEARCH DESIGN

In this section, we discuss the research design and the steps, which we followed to identify the patterns and to capture them in a pattern catalog. In this scope, the sources of information were (i) collaboration with the Daimler AG (ii) literature survey of exiting work on pattern languages, cloud provider documentation and whitepapers, and (iii) self-experience originating from our work on provisioning systems and cloud application development. Based on this information, we performed a set of steps to identify patterns.

[1] http://www.cloudcomputingpatterns.org

2.1 Industry Collaboration

During the collaboration with the Daimler AG, we investigated how cloud customers employ cloud resources to build distributed applications, for example, to let drivers determine how far their electric cars can travel [26]. Daimler AG has also used software design patterns successfully in the past to homogenize distributed application landscapes [8]. Additionally, to using cloud resources as a customer, Daimler AG offers multiple applications as a service, which experience challenges that can be addressed using cloud computing technologies. For example, car2go [9] is a car sharing service offered in different cities in Germany, the US, and Canada. Users of this service use an online application to find available cars, make reservations etc. Similar functionality is offered to individual companies by FleetBoard [10]. Architectural concepts and principles of these applications were abstracted to general patterns presented in this paper.

2.2 Literature Survey

To identify a suitable pattern format, we covered existing work on pattern languages. [23] and [25] cover a general structure of such languages and their design. The concrete pattern format that we employed is mainly inspired by existing pattern language definitions, most important [17] and [14]. Existing patterns were also reviewed for their applicability to cloud computing, because many of the challenges faced by cloud applications are similar to those of standalone applications, grid applications, messaging based applications etc. Existing patterns having a different scope, for example, describing concepts for standalone application development have been transferred to distributed cloud applications. Also, existing patterns were identified that can be applied to cloud applications with minor to no adjustments. Concepts of these patterns were expressed in the same pattern format as the rest of the catalog to increase readability by using a homogeneous representation of patterns [35].

To determine common principles of cloud computing, we have reviewed further literature. In [21], the basics of cloud computing are covered. The relevant cloud service models are described, which are used to offer different type of IT resources to customers on demand. Further, we reviewed industry definitions of cloud computing [24] and its use cases [42] as well as its business models and multi-tenancy concepts [6].

To cover architectural guidelines of cloud providers, we have evaluated cloud offerings of Amazon [1], Windows Azure [29], T-Systems [41], and Google [15]. Central architectural novelties introduced by cloud computing that were observed are the relaxation of consistency models and componentization of applications. Relaxation of consistency is used to increase availability and performance, but it also results in eventual consistency of data stores [45] [40] [4]. Componentization of applications is used to scale each application component individually according to the workload currently experienced [43] [44].

The central focus during the literature survey was the abstraction of common underlying concepts to form the basis for pattern descriptions.

2.3 Self-Experience with Cloud Applications

User-centric customization of applications and their automated provisioning is of major importance in the area of cloud computing. Customizability enables a cloud application provider to increase the size of the addressable customer market, which is important to the economic success of a cloud offering [6].

[30] describes means to model applications and their variability, so that cloud application providers may offer applications via self-service interfaces and automate their provisioning. Customers may use the self-service interfaces to adjust an application to their needs and provision it in different environments depending on the desired service levels.

Together with another industry partner, T-Systems, the ICT subsidiary of Deutsche Telekom, we investigated how to address law and security regulations in cloud architectures [3]. Further, we investigated together how to enable application variability [31]. We found that the application architecture needs to be designed specifically with flexibility in mind and identified four different classes of variability: *data variability* means that the data objects stored by the application can be adjusted; *provisioning variability* refers to the above mentioned possibility to use different environments for the execution of the application; *functional variability* allows the user to adjust the processes supported by the application; *user interface variability* refers to the interface of an application being adjustable, for example, regarding its language. We also identified architectural principles and one pattern that capture the essence of variability and configurability in the application architecture [12].

2.4 Pattern Identification

To identify and describe reoccurring good solutions to problems in cloud application development and to capture common concepts of clouds and their offerings, the following steps have been performed.

Step 1: Definition of the pattern format – based on the available work on pattern languages covered, we chose the sections of our pattern format.

Step 2: Identification of significant concepts – the significant concepts were extracted from the set of information sources, such as architectural guidelines of cloud products, existing cloud applications, existing architectural patterns from other domains, and Daimler-internal architecture guidelines. To be significant, concepts had to fulfill one of the following characteristics:

- The concept is referred to by a term that is reoccurring in multiple sources of information.
- The concept is important to the application development in terms of application management, availability, scalability, elasticity, and pay-per-use.
- The concept describes a solution to a cloud-specific problem, for example, how to share application components between multiple users while isolating them from another (multi-tenancy).
- The concept distinguishes the solution from other similar solutions. For example, the privacy guaranteed by a cloud offering and the displayed accessibility were used to describe clouds as being ”public” or “private”.

Step 3: Identification of irrelevant details – since the reviewed information was often provider-specific or use-case-specific, it contained aspects that were only relevant in that scope. Irrelevant information was identified if it fell into one of the following categories:

- Description of provider supplied interfaces and details of their invocation, for example, the specification of message formats and transfer protocols.
- Description of the functionality that an application shall display in a concrete use-case.
- Background information on general cloud use that serves as the introduction to a concrete use-case or architectural guideline.

Step 4: Abstract description of significant concepts – based on the filtered information sources, abstract descriptions of the significant concepts were created. They were used to state the essential information left in information sources after the significant concepts were summarized and the irrelevant ones were omitted. For example, an abstracted description based on information obtained from Amazon’s white papers [44] [43] was that cloud applications using Amazon’s Cloud Offerings (Amzon Web Service, AWS) [1] had to be componentized, so that component could be scaled out individually.

Step 5: Pattern Creation and Classification – reoccurring abstracted descriptions obtained from different information sources were compiled into patterns based on the defined pattern format. We classified patterns into four classes: *cloud service models* – concepts describing the style in which different IT resources are offered in clouds; *cloud types* – descriptions of properties and behavior of different clouds; *cloud offerings*– description of the functionality and behavior of different cloud offerings used for computation, communication, and storage; *cloud application architectures* – concepts how applications can be built on top of cloud offerings.

Step 6: Iterative Improvement – during the last three steps, we had multiple discussions with Daimler AG employees regarding the readability and the usability of patterns in application development projects. Further, we identified how an overview of the patterns can be given to application developers.

3. PATTERNS OF CLOUD-BASED APPLICATIONS

Based on the described research design and the pattern format, we have identified and described cloud types, cloud service models, cloud offerings and their behavior, as well as cloud application architectures that are built on top of such offerings. Through description of the provider side (cloud types, cloud service modes and cloud offerings) in the same form and catalog, the identification of application architectural patterns relevant for a developer is simplified, because patterns are set into perspective using relations among them. A uniform way to describe information also eases perception [35]. In the following, we cover the used pattern format, introduce the new *cloud component gateway pattern*, and give an overview of the other patterns that we have discovered so far [13].

3.1 Pattern Format

The pattern format comprises the following sections. It has been developed during Step 2 described in Section 2.4.

Name – patterns are identified by a unique name that specifies the purpose of the pattern or the entity in the application architecture that is described by the pattern.

Icon – each pattern is also identified by a graphical icon to be used in architectural diagrams. Icons were designed to resemble equally sized boxes that contain minimal graphical elements from which the essence of the pattern description may be obtained as good as possible. Also, the icons may be used in the description of more complex patterns that are composed of multiple other patterns.

Driving Question – at the beginning of the detailed pattern description, the problem solved by the pattern is given in form of a short question. Since cloud application developers use the pattern catalog to search for solutions to questions at hand, this form eases the identification of patterns that fit their current problems.

Context – for each pattern the conditions under which the described problem may arise are given. References to other patterns may be used to describe the context. Especially, the pattern descriptions of cloud types and cloud offerings can be referred to in this section. Therefore, their description in a pattern form significantly eased the description of the environment in which an application architectural pattern can be applied and vice versa, it eased describing the requirements a pattern has on the environment.

Challenges – while the driving question allows a quick perception of the essential problem solved by the pattern, a detailed description of all challenges is given in this section.

Solution – this section gives instructions on how to address the challenges using the pattern. Instructions are given in the form of short steps to follow.

Sketch – explanation of the solution employed by the pattern is guided by a graphical sketch. It depicts the fundamental architectural components of the solution. This sketch may also contain icons of other patterns that the described pattern uses in its solution.

Result – in addition to the brief solution statement, the result section describes in greater detail how the required steps can be implemented and what the outcome will be. Additional challenges that arise under these new conditions may also be covered here.

Relations to other patterns – the pattern may be used in the context of other patterns, which can be referenced here. Also, patterns having similar challenges or context can be pointed to using this section.

Patterns outside of the catalog can also be referenced here on which the described pattern is based or has been derived from. Especially, these can be established patterns whose concepts have been transferred to the area of cloud computing.

Variations – often, a pattern can be applied in slightly different forms. If the differences of these forms are not significant enough to justify their description as completely separate patterns, they are covered in this section.

Known Uses – concrete applications, use cases, cloud offerings, and documents are referenced here from which the pattern has been abstracted.

Annotations – for the purpose of extensibility, patterns may be annotated with additional artifacts related to their instantiation on concrete platforms, their behavior, monitoring of the state of contained components etc. We propose the annotations of runtime-specific and cloud environment-specific artifacts to patterns in the catalog. Especially, annotations may be used to guide developers during the configuration of the runtime environment which serves an implemented pattern as runtime. Because this information is cloud environment-specific, we do not organize it in the same catalog as the pattern descriptions.

Not all types of annotation are applicable to all patterns in the same way. For example, a pattern describing a cloud type can be annotated with names and services levels of providers offering that type of cloud. Patterns describing cloud offerings can be annotated with process models, sequence diagrams or code snippets describing how to use them in applications. Additionally, standardized monitoring models, interface descriptions, and event specifications can be annotated to patterns. This can be used to ensure a standardized extraction of monitoring information given a custom implementation of a pattern.

3.2 Cloud Component Gateway Pattern

Based on the pattern format, we described the discovered patterns. The *cloud component gateway pattern* was discovered during the analysis of the Windows Azure App Fabric [27] offered by Microsoft and the WSO2 Enterprise Service Bus [47], a product of WSO2. Provider web sites, development guidelines, life product presentations and discussions with technical consultants were used as information sources for these cloud offerings of respective companies.

To obtain the underlying concepts, we omitted irrelevant details, such as exemplary implementation code or specific transport protocols. From the remaining relevant information, we abstracted to obtain the common underlying concepts. We found that both providers offered functionality to enable communication between environments for which communication was restricted. The common abstract concepts are (i) two different environments are bridged, (ii) inaccessible functionality is mocked, and (iii) access to the mocked functions are relayed using unrestricted communication channels that originally were not intended for this communication. We compared these concepts to existing patterns from other domains and found a relation to the proxy pattern [5] and the façade pattern [5] used in standalone applications.

The abstracted concepts and information obtained from existing patterns was distilled into the pattern format followed by collaborative iterations and reviews to identify challenges arising after the application of the pattern. During these iterations, the relations of the following *cloud component gateway pattern* to other patterns have also been identified.

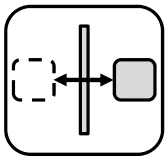


Figure 1: Icon of the Cloud Component Gateway Pattern

Name: Cloud Component Gateway

Icon: The icon is depicted in Figure 1.

Driving Question: How can an application component be made accessible in one cloud or datacenter, when it is hosted in a different cloud or datacenter and the communication between these environments is restricted?

Context: Multiple applications or their components are distributed among different clouds or datacenters. The synchronous or asynchronous communication between these environments is restricted, for example, through the use of firewalls. In most applications of this pattern, the inbound communication from an off-premise environment to an on-premise environment is restricted:

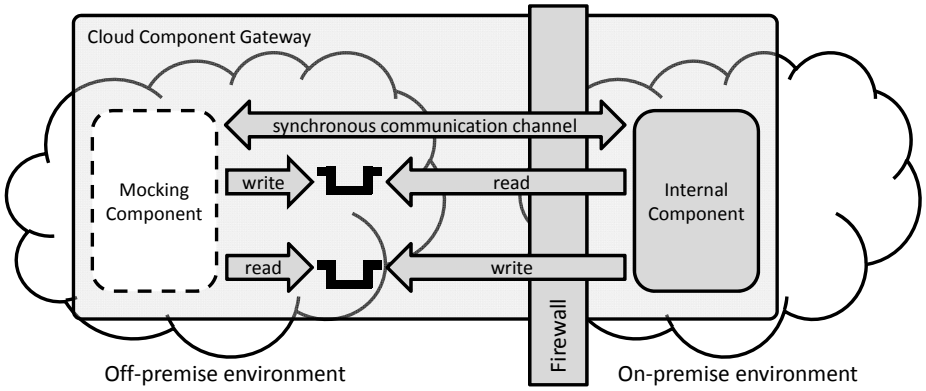


Figure 2: Sketch of the Cloud Component Gateway Pattern

while the internal components residing in the on-premise environment may gain access to external components, no external access to internal components is allowed.

Challenges: Available communication channels between different computing environments used by a company are often restricted. While outbound communication is possible, inbound communication from an off-premise environment may be restricted completely. However, application components hosted off-premise may need to access data and functionality of internal application components that are hosted on-premise, but direct access to these components is impossible.

Solution: Duplicate the interface of the internal component in the off-premise environment. This duplicate mocks the internal component's interface and forwards accesses to the internal component.

Sketch: The sketch is depicted in Figure 2.

Result: Since direct access to the internal component is avoided, its data and functions can be accessed through its duplicate. Communication channels may be synchronous or asynchronous and are always established and maintained by the internal component due to the access restrictions.

In case of synchronous access, a communication channel between the internal component and its external interface is initiated by the internal component. Such an access is similar to that to an external web server, which also has to send information back into on-premise environments. In contrast to direct access to the internal component from the outside, the synchronous communication channel must be maintained at all times by the internal component, because it cannot be triggered from the off-premise environment.

Asynchronous access is enabled through the use of message queues residing in the off-premise environment. Accesses to the external component are put into a queue from which the internal component may obtain them. It writes its output to another queue, where the external component may access it. Again, access to the internal component cannot be triggered from the outside. Therefore, a critical design challenges is to determine how often the internal component polls new messages.

Relations to other patterns: The *cloud component gateway* can be used in *hybrid clouds*[2] to address communication restrictions between computing environments that are integrated. This applies to the integration of multiple applications as well as to the integration of application components comprising a *composite application*[3]. The asynchronous communication may follow the *reliable messaging pattern*[4] and can guarantee *exactly-once delivery*[5] or *at-least-once delivery*[6].

Further, the cloud component gateway is based on concepts of the proxy pattern [5] and the façade pattern [5], which address similar challenges in standalone applications.

Variations: In principle, this pattern describes an integration between different environments. Therefore, it may also be used to integrate two off-premise environments, for example, if the communication between these environments is restricted in a similar fashion.

Also, the queues required to realize the asynchronous communication between the internal component and its interface component may be hosted by a different provider, a so called Enterprise Application Integration (EAI) as a Service [37] provider, or in a specially controlled network segment in the on-premise environment, a demilitarized zone (DMZ) [22].

Known Uses: Microsoft is offering an implementation of this pattern using a synchronous communication channel and services as application components to be integrated. It is part of Windows Azure AppFabric [27]. If the internal service is developed using Microsoft development tools, the mocking component can be generated automatically. In case other internal services need to be made available off-premise, an additional on-premise service has to be developed that accesses the existing service.

WSO2 offers asynchronous access to internal components as part of the WSO2 enterprise service bus [47]. If this enterprise service bus (ESB) is installed on-premise and in an off-premise environment, so-called "service gateways" may be used to make services accessible in both environments.

Annotations: Use of the Windows Azure implementation is described in [28]. Documentation for the WSO2 service gateways is given by [46].

3.3 List of Patterns in the Catalog

We give an overview of previously discovered patters in the form of a list containing pattern names, their icons, and driving

[2] http://cloudcomputingpatterns.org/?page_id=106

[3] http://cloudcomputingpatterns.org/?page_id=240

[4] http://cloudcomputingpatterns.org/?page_id=195

[5] http://cloudcomputingpatterns.org/?page_id=199

[6] http://cloudcomputingpatterns.org/?page_id=204

questions. The list is divided into separate sections for the different classes of patterns describing cloud service models, cloud types, cloud offerings, and cloud application architectures.

This representation was inspired by [17]. During the cooperation with Daimler AG, we found that it enabled an easy access to patterns, because developers could identify patterns based on the graphical icon and the question they were trying to solve.

We make no claim to provide a complete list of patterns for any scenario, because cloud computing is still a very new research area. Also note that the catalog contains existing patterns that have been transformed into the pattern format used in the catalog. For these patterns a reference to the original source is given after their name.

Cloud Service Models

Infrastructure as a Service: How can IT infrastructure be offered dynamically over a network?

Platform as a Service: How can IT platforms be offered dynamically over a network?

Software as a Service: How can software be offered dynamically over a network?

Composite as a Service: How can composite application be offered dynamically over a network?

Cloud Types

Public Cloud: How can elastic IT services be offered concurrently to different companies?

Private Cloud: How can elastic IT services be offered exclusively for internal use of one company?

Hybrid Cloud: How can elastic IT services be offered to multiple companies, whilst some services are used exclusively by one company and may even be provided by it?

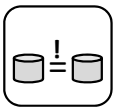
Community Cloud: How can elastic IT services be offered concurrently to a certain set of companies?

Cloud Offerings

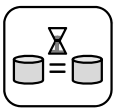
Elastic Infrastructure: How can IT resources by offered dynamically and on-demand?

Low-available Compute Node: How can compute services be offered at low costs if their availability is relaxed?

High-available Compute Node: How can compute services be offered if their availability is of vital importance?

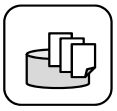


Strict Consistency: How can the availability of a storage solution be increased while consistency is ensured at all times?



Eventual Consistency: How can the availability and performance of a distributed storage solution be increased if the requirement on consistency is loosened?

Relational Data Store: How can data elements be stored so that relations between their attributes and those of other elements can be expressed and complex queries based on these attributes are possible?



Blob Storage: How can large data elements be stored and organized centrally and made available over a network?

Block Storage: How can central storage be accessed similar to local hard drives?

NoSQL Storage: How can a database support extreme scale-out and a flexible data structure?

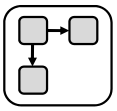
Message Oriented Middleware [17]: How can applications (or application components) communicate remotely via messages while being loosely coupled regarding their location and message format?

Reliable Messaging [17]: How can messages be exchanged while guaranteeing that messages are not lost in case of system or communication failures?

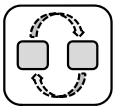
Exactly-once Delivery [17]: How can a message oriented middleware assure that a message send through it is delivered only once to a receiver?

At-least-once Delivery [17]: How can the performance of a messaging system be increased if duplicate messages are acceptable?

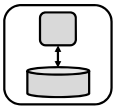
Cloud Application Architectures



Composite Application: How can application functionality be distributed and composed from various sources?

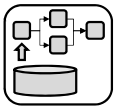


Loose Coupling: How can the dependencies between applications and their components be reduced?



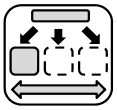
Stateless Component: How can data loss be avoided if a component of an application fails or is removed from the application?

Idempotent Component [17]: How can a component receiving messages handle duplicate messages?

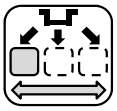


Map-Reduce: How can the performance of complex queries on large data sets be increased if the used storage solution does not support such queries natively?

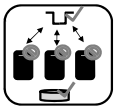
Elastic Component: How can the number of application components, that are scaled-out, be adjusted automatically based on system utilization?



Elastic Load Balancer: How can the number of application components, that are scaled-out, be adjusted automatically based on the number of requests?

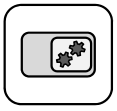


Elastic Queue: How can the number of application components, that are scaled-out, be adjusted automatically based on the number of asynchronous requests in an optimized fashion?

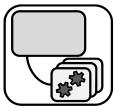


Watchdog: How can a high available application be realized using unreliable compute nodes?

Update Transition: How can a componentized application be updated seamlessly, when new versions of application components or the used middleware, operating system etc. become available?



Single Instance Component [32]: How can an application component be shared between multiple tenants, if individual configuration is not required?



Single Configurable Instance Component [32]: How can an application component be shared between multiple tenants if individual configuration is required?

Multiple Instance Component [32]: How can an application component be provided to multiple tenants who configure it, if sharing is unfeasible?

4. PATTERN-BASED APPLICATION DEVELOPMENT

In the following, we introduce a method to obtain a set of applicable patterns from given requirements and propose the use of annotations on these patterns to assist their instantiation.

4.1 Populating a Decision Recommendation Table

The basic assumption for the proceeding we propose is the availability of a decision recommendation table containing pattern interrelations of a given catalog. Here, we use the catalog of patterns described in Section 3 as basis. The resulting decision recommendation table is depicted in Table 1. It contains pattern interrelations of three different types.

1. The **strong cohesion** relation (+) states that one pattern is likely to be combined with the related pattern. For example, in a public cloud, where individual resources are likely to display a low availability, patterns to distribute application functionality among multiple resources and monitor these resources have a strong cohesion to the *public cloud pattern*[7].
2. The **exclusion** relation (-) states that two patterns cannot be combined. For example, the use of high available compute nodes renders the use of patterns handling component failure on the application level unnecessary. Therefore, such patterns have an exclusion relation to the *high availability compute node pattern*[8].
3. The **undetermined** relation (o) states that neither "strong cohesion" nor "exclusion" exists between two patterns. For example, patterns addressing the availability of applications are likely to be unrelated to patterns addressing security, because these requirements are mainly orthogonal.

To populate the decision recommendation table, the relations between all patterns were initially set to being undetermined. Then, relations were obtained from the pattern descriptions themselves. For example, if a pattern uses another pattern to describe its context, it is strongly related to it. This is the case for the *hybrid cloud pattern* setting the context for the *cloud component gateway pattern*. In the same fashion, a pattern may explicitly state that it cannot be combined with another pattern. Further exclusion relations were determined based on the experience of the authors. Many of these relations were straight forward, because the problem solved by one pattern does not occur in the context of another pattern. For example, the *cloud component gateway pattern* addresses challenges arising during the integration of different environments. It therefore has an exclusion relation to all patterns describing homogeneous *cloud types* (public, private, community). In these cloud types, the *cloud component gateway pattern* is not useful. Additionally, the information sources described in Section 2 were analyzed to obtain pattern relations. If the set of information sources from which a pattern was obtained displayed a large overlap with the set of information sources of another patterns, it was investigated if these patterns are strongly related.

After the decision recommendation table has been populated, relations between patterns should be bi-directional. Therefore, the table should be symmetric regarding its diagonal. If this is not the case, references between patterns may be missing in the pattern descriptions. As a side effect, the decision recommendation table can, thus, be used during the iterative improvement of pattern descriptions performed in Step 6 described in Section 2.4.

4.2 Identification of Applicable Patterns

The inclusion of patterns describing cloud types, cloud service models, and cloud offerings guides the developer during the identification of applicable cloud application architecture patterns as follows: the developer can start by selecting patterns that describe the environment in which the developed application will be deployed. Based on these selections, cloud application architecture patterns are then recommended to him or her for implementation. The developer selects patterns in the decision recommendation table that shall be used and omits those that cannot be used. Conflicting selections can then be detected. The developer has to resolve these conflicts manually by deciding which patterns are more important to him or her.

[7] http://cloudcomputingpatterns.org/?page_id=90

[8] http://cloudcomputingpatterns.org/?page_id=156

+ : strong cohesion - : exclusion o : undetermined		Cloud Service Models				Cloud Types					Cloud Offerings													Cloud Application Architectures																			
		Infrastructure as a Service	Platform as a Service	Software as a Service	Composite as a Service	Public Cloud	Private Cloud	Hybrid Cloud	Community Cloud	Elastic Infrastructure	Low-available Compute Node	High-available Compute Node	Strict Consistency	Eventual Consistency	Relational Data Store	Blob Storage	Block Storage	NoSQL Storage	Message Oriented Middleware	Reliable Messaging	Exactly-once Delivery	At-least-once Delivery	Composite Application	Loose Coupling	Stateless Component	Idempotent Component	Map-Reduce	Elastic Component	Elastic Load Balancer	Elastic Queue	Watchdog	Update Transition	Single Instance Component	Single Configurable Instance Component	Multiple Instance Component	Cloud Component Gateway
Cloud Service Models	Infrastructure as a Service		o	o	o	+	+	+	+	+	+	+	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	+	o	o	o	o	o	o	o	o
	Platform as a Service	o		o	o	+	+	+	+	o	-	-	+	+	+	+	+	+	+	+	+	+	o	o	o	o	+	o	+	+	+	o	o	o	o	o
	Software as a Service	o	o		o	+	+	+	+	o	-	-	o	o	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	o	o	o	o	o
	Composite as a Service	o	o	o		+	+	+	+	o	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	o	o	o	o	o	o	+	+	o	o
Cloud Types	Public Cloud	+	+	+	+		-	-	-	+	+	o	o	+	o	o	o	o	o	o	o	o	+	+	o	o	o	+	+	+	+	o	o	o	-
	Private Cloud	+	+	+	+	-		-	-	+	o	+	o	+	o	o	o	o	o	o	o	o	+	o	o	o	o	+	+	+	o	+	o	o	-
	Hybrid Cloud	+	+	+	+	-	-		-	+	+	+	o	+	o	o	o	o	o	o	o	o	+	+	o	o	o	+	+	+	+	+	+	+	+
	Community Cloud	+	+	+	+	-	-	-		+	+	o	o	+	o	o	o	o	o	o	o	o	+	+	o	o	o	+	+	+	+	+	o	o	-
Cloud Offerings	Elastic Infrastructure	+	o	o	o	+	+	+	+		+	+	o	o	o	o	o	o	o	o	o	o	+	+	o	o	o	o	o	o	o	+	o	o	o
	Low-available Compute Node	+	-	-	+	+	o	+	+	+		-	o	o	o	o	o	o	o	o	o	o	+	+	+	o	o	o	o	o	+	o	o	o	o
	High-available Compute Node	+	-	-	+	o	+	+	o	+	-		+	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	-	o	o	o	o
	Strict Consistency	o	+	o	+	o	o	o	o	o	o	+		-	+	+	+	-	+	o	+	o	+	o	o	o	o	o	o	o	+	o	o	o	o
	Eventual Consistency	o	+	o	+	+	+	+	+	o	o	o	-		-	+	+	+	+	-	-	+	+	+	o	o	o	o	o	o	+	o	o	o	o
	Relational Data Store	o	+	-	+	o	o	o	o	o	o	o	+	-		o	o	o	o	o	o	o	+	+	+	o	o	+	+	+	o	o	o	o	o
	Blob Storage	o	+	-	+	o	o	o	o	o	o	o	+	+	o		+	o	o	o	o	o	+	o	+	o	+	+	+	+	o	+	o	o	o
	Block Storage	o	+	-	+	o	o	o	o	o	o	o	+	+	o	+		o	o	o	o	o	+	+	+	o	o	+	+	+	o	+	o	o	o
	NoSQL Storage	o	+	-	+	o	o	o	o	o	o	o	-	+	o	o	o		o	o	o	o	+	+	+	+	+	o	o	o	o	o	o	o	o
	Message Oriented Middleware	o	+	-	+	o	o	o	o	o	o	o	+	+	o	o	o	o		+	+	+	o	+	+	+	+	o	o	+	o	+	o	o	+
	Reliable Messaging	o	+	-	+	o	o	o	o	o	o	o	-	o	o	o	o	o	+		+	-	o	o	o	-	o	-	-	+	o	o	o	o	+
	Exactly-once Delivery	o	+	-	+	o	o	o	o	o	o	o	+	-	o	o	o	o	+	+		-	o	o	+	+	o	-	-	+	o	o	o	o	+
	At-least-once Delivery	o	+	-	+	o	o	o	o	o	o	o	+	+	o	o	o	o	+	-	-		+	+	+	+	o	-	-	+	o	o	o	o	+
Cloud Application Architectures	Composite Application	o	o	-	+	+	+	+	+	+	+	o	+	+	+	+	+	+	o	o	o	+		+	+	+	+	+	+	+	+	+	+	+	+
	Loose Coupling	o	o	-	+	+	o	+	+	+	+	o	o	+	+	+	+	+	+	o	o	+	+		+	+	o	o	o	o	+	+	o	o	+
	Stateless Component	o	o	-	+	o	o	o	o	o	+	o	o	o	+	+	+	+	+	o	+	+	+	+		+	o	o	o	o	+	+	o	o	+
	Idempotent Component	o	o	-	+	o	o	o	o	o	o	o	o	o	o	o	o	+	+	-	+	+	+	+	+		o	o	o	+	o	o	o	o	+
	Map-Reduce	o	+	-	o	o	o	o	o	o	o	o	o	o	o	+	o	+	+	o	o	o	+	o	o	o		+	+	+	o	o	o	o	o
	Elastic Component	+	o	-	o	+	+	+	+	o	o	o	o	o	+	+	+	o	o	-	-	-	+	o	o	o	o		o	o	o	o	+	+	+	o
	Elastic Load Balancer	o	+	-	o	+	+	+	+	o	o	o	o	o	+	+	+	o	o	-	-	-	+	o	o	o	o	+		o	o	o	+	+	+	o
	Elastic Queue	o	+	-	o	+	+	+	+	o	o	o	o	o	+	+	+	o	+	+	+	+	+	o	o	+	+	o	o		o	+	+	+	+	o
	Watchdog	o	+	-	o	+	o	+	o	o	+	-	+	o	o	o	o	o	o	o	o	o	+	+	+	o	o	o	o	o		o	+	+	+	o
	Update Transition	o	+	+	o	+	+	+	+	+	o	o	o	+	o	+	+	o	+	o	o	o	+	+	+	o	o	o	+	+	o		o	o	o	o
	Single Instance Component	o	+	+	+	o	o	+	+	o	o	o	o	o	o	o	o	o	o	o	o	o	+	o	o	o	o	+	+	+	+	o		o	o	o
	Single Configurable Instance Component	o	+	+	+	o	o	+	+	o	o	o	o	o	o	o	o	o	o	o	o	o	+	o	o	o	o	+	+	+	+	o	o		o	o
	Multiple Instance Component	o	+	+	+	o	o	+	+	o	o	o	o	o	o	o	o	o	o	o	o	o	+	o	o	o	o	+	+	+	+	o	o	o		o
	Cloud Component Gateway	o	o	o	o	-	-	+	-	o	o	o	o	o	o	o	o	o	+	+	+	+	+	+	+	+	o	o	o	o	o	o	o	o	o	

Table 1: Decision Recommendation Table

As a result of a selection, four different sets of patterns can be discriminated: (i) patterns chosen explicitly by the developer; (ii) patterns that are likely to be applicable as well, thus, they are interrelated to the set of chosen patterns via the "strong cohesion"; (iii) patterns of which the applicability is undetermined; (iv) patterns that cannot be used. After each selection, these sets can be visualized, for example, through highlighting entries in the decision table with different colors. The list of applicable patterns is then refined iteratively by the developer to obtain the effective set of patterns applicable in the concrete use case.

For example, a developer starts by selecting the *hybrid cloud pattern*, because he or she knows that two different types of clouds shall be used as an integrated runtime environment. Further, the developer selects *NoSQL*[9] to be used as storage. Based on this selection of cloud type and cloud offering patterns, cloud architectural patterns are recommended for implementation. The *cloud component gateway pattern* is recommended due to a strong cohesion to the *hybrid cloud pattern* and the *map-reduce pattern*[10] is recommended, because it has a strong cohesion to the *NoSQL pattern*. If the developer also selects recommended patterns, the strong cohesion relations of those patterns are again evaluated to recommend further patterns. Iterative execution of these steps leads to a user-driven refinement of the set of patterns to be used in a concrete solution.

[9] http://cloudcomputingpatterns.org/?page_id=186

[10] http://cloudcomputingpatterns.org/?page_id=260

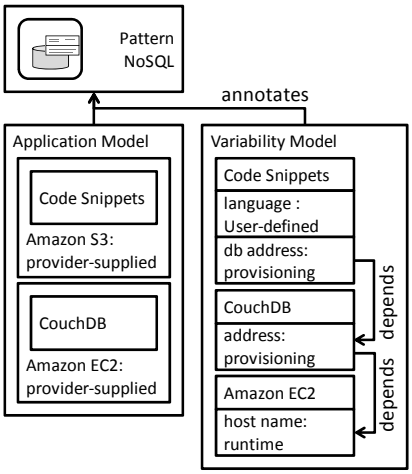


Figure 3: Example of Pattern Annotations

4.3 Instantiation of Applicable Patterns

After the application developer has identified the set of patterns applicable to his or her problem, annotations are used to assist the developer also during the setup of the runtime environment and the creation of the cloud application. Since the structure of annotations is arbitrary, each pattern can have individual artifacts associated with it that assist the developer during its instantiation. We suggest the following artifacts and their use during the instantiation of patterns: *cloud types* and *cloud service models* – patterns of this class can be associated with information on concrete cloud providers, their service levels, and pricing information. *Cloud offerings* – this class of patterns can be annotated with a list of concrete offerings of cloud providers. Beyond human-readable information on how to use these concrete offerings, we further suggest the annotation of *application models* and *variability models* [31] containing machine-readable information that can be used to provision the offering. These models also describe how additionally annotated code snippets can be adjusted for the provisioned offering. They can then be used in the application created by the developer. This customization of annotated models is enabled by interdependencies. The code snippets associated with an architectural pattern may for example have dependencies on the address of a used cloud offering and security keys required to access it. This information is only available after the cloud offering has been provisioned for the application developer, who is then provided with an individually adjusted version of the code snippets that he or she can use directly to access the offering.

We will now give an example for concrete annotations of application models and variability models. Also, their use in a proposed framework to organize, search and instantiate patterns will be covered. The *NoSQL pattern* describes a cloud offering. Therefore, cloud providers are annotated to it, along with multiple application models and variability models used for the provisioning of the offering at a provider. An example for one of such annotations is depicted in Figure 3. It contains artifacts for the use of Apache CouchDB [2] on an Amazon EC2 [1] instance. The application model describes how the application components are deployed on other application components. The code snippets are deployed to Amazon S3 [1] from where they can be downloaded by the developer. The CouchDB installation is uploaded to an EC2 virtual machine. Amazon S3 and EC2 are modeled as being *provider-supplied* meaning that during the provisioning provider-supplied management interfaces will be accessed. In the variability model, *variability points* and their dependencies are modeled that need to be bound during this provisioning. Each variability point is associated with a certain phase. During *customization*, the user selects from a list of available implementation languages specified in the language variability point of code snippet.

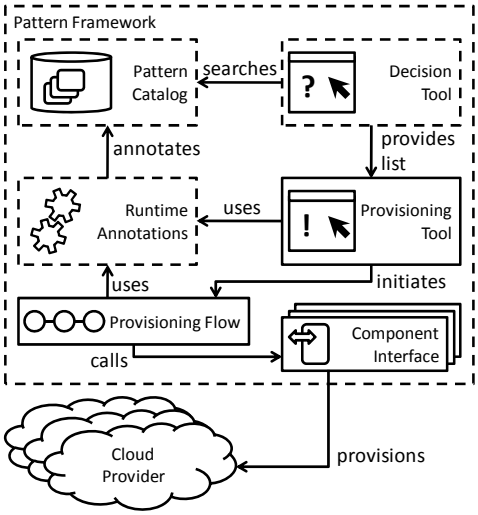


Figure 4: Cloud Pattern Framework

The variability point for the database address used in the code must be bound prior to its *provisioning*. This means that the code snippet must be adjusted prior to uploading it to Amazon S3 where it can be accessed by the application developer. The database address in the code snippet depends on the address of the CouchDB. The corresponding variability point of the CouchDB component depends on the host name of the EC2 virtual machine that becomes available during its *runtime*. Based on these dependencies among variability points, the user is guided through the customization and the order in which components must be provisioned can be determined [31].

4.4 Integration of the Pattern Catalog and Decision Table into a Provisioning System

We envision the necessary tasks for pattern discovery and instantiation to be supported by a framework depicted in Figure 4. Components that are not implemented yet have dashed borders in the Figure. For these components, we have presented the concepts on which they will rely. The pattern format and the annotation of arbitrary information to patterns will form the basis for the *pattern catalog component* and the *runtime annotations component* of the framework. The decision recommendation table will be used in the *decision tool component*.

We further propose the following use of the framework. The application developer first accesses the *decision tool*, which is currently in the form of the decision recommendation table (see Table 1). After the required patterns have been identified, the *provisioning tool* is accessed to instantiate the required runtime and to provide code snippets as a basis for the custom implementation. We will use the existing provisioning tool Cafe [33], which provides the *provisioning tool*, the *provisioning flow* and *component interfaces* to the framework. The *provisioning tool* guides the developer during the customization and provisioning of pattern implementations and their runtime infrastructures.

It accesses the application models and variability models annotated to used patterns. Variability points that require user decisions are obtained from the user. When all of these variability points have been bound, the customization tool passes the models to the *provisioning flow*. This flow analyses the other variability point dependencies and provisions components in the respective order. It does so by accessing a set of component interfaces, each encapsulating the functionality required to setup components.

In the above example, the framework would therefore contain at least three of these component interfaces for the instantiation of Amazon EC2 machines, the deployment of Apache CouchDB on top of a running machine, and the upload of code snippets to Amazon S3. After the provisioning flow has been executed, it returns an entry point in the form of an URL to the provisioning tool. The application developer uses this URL to access the code snippets that were customized for his or her runtime environment.

5. SUMMARY AND OUTLOOK

As cloud computing is a new field of research and inventions are mainly industry driven, discussions we had with other researchers and industry partners alike were often hindered by unclear definitions of used terms and unclear cloud-specific principles and concepts. We experienced that the use of a pattern format beyond the description of good solutions eased these discussions and helped developers unfamiliar with cloud computing to gain a quicker access to the field. In this scope, a pattern-based description also unified architectural guidelines of different providers laying the basis for a structured teaching of principles and concepts of cloud computing.

We have presented an approach how to describe the required architectural structures of cloud applications as patterns. The same pattern format has also been used to describe cloud types, cloud service models, and cloud offerings to enable a guided identification of patterns applicable in a concrete cloud environment. Further, we have shown how the patterns can be annotated with provider-specific provisioning information and code snippets. The proposed approach respects the cloud-specific dynamicity and flexibility during the development process of cloud applications by bringing architectural patterns and their instances closer together. The degree to which this impacts the development of applications will be investigated further during the ongoing collaboration with Daimler AG.

Identification and description of cloud architectural patterns related to legal regulations, security, auditing, trust, and billing are ongoing. With a growing catalog, the usability of the introduced decision recommendation table will be reduced for humans. Therefore, our future work is also to develop additional methods and tools that use the information contained in the proposed decision recommendation table to organize patterns, make them searchable in a comfortable form, and recommend them to application developers.

6. ACKNOWLEDGMENTS

Many thanks go to Ernst Oberortner for his insightful comments and constructive feedback during the review of this paper.

Christoph Fehling, Frank Leymann, Ralph Retter and David Schumm would like to thank the Daimler AG for the collaboration.

Additionally, David Schumm would like to thank the German Research Foundation (DFG) for financial support within the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart.

7. REFERENCES

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

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

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

[4] Brewer E.A.: Towards Robust Distributed Systems. PODC Keynote, 2000. http://www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[5] Buschmann F., Henney K., Schmidt D.C.: Pattern-oriented Software Architecture: On Patterns and Pattern Languages. Wiley, 2007.

[6] Chong F., Carraro G.: Architecture Strategies for Catching the Long Tail. MSDN Library, Microsoft Corporation, 2006. http://www.cistratech.com/whitepapers/MS_longtailsaas.pdf

[7] Ciurana E.: Developing with Google AppEngine. Springer, 2009.

[8] DaimlerChrysler TSS GmbH: MDA Success Story ePEP successful with Model Driven Architecture, 2005. http://www.omg.org/mda/mda_files/SuccesStory_DC_TSS_MDO_English.pdf

[9] Daimler AG, car2go, 2010. http://www.car2go.com

[10] Daimler FleetBoard GmbH, FleedBoard, 2011. http://www.fleetboard.com

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

[12] Fehling C., Konrad R., Leymann F., Mietzner R., Pauly M., Schumm D.: Flexible Process-based Applications in Hybrid Clouds. Proceedings of the 2011 IEEE International Conference on Cloud Computing, 2011.

[13] Fehling C., Mietzner R., Leymann F.: A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-based Application Architectures. Technical Report, 2011. http://www.iaas.uni-stuttgart.de/institut/mitarbeiter/fehling/TR-2011-05-Patterns_for_Cloud_Computing.pdf

[14] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1995.

[15] Google: AppEngine, 2011. http://code.google.com/appengine/

[16] Hanmer R. S., Kocan K. F.: Documenting Architectures with Patterns. Bell Labs Technical Journal, Volume 9, Number 1, 2004.

[17] Hohpe G., Wolf B.: Enterprise Integration Patterns: Designing, Building, and Deploying. Addison-Wesley, 2004.

[18] IBM, Corporate Strategy Analysis of IDC data, 2007.

[19] IEEE: Cloud Profiles Working Group (CPWG), 2011. http://standards.ieee.org/develop/wg/CPWG-2301_WG.html

[20] IEEE: Intercloud Working Group (ICWG), 2011. http://standards.ieee.org/develop/wg/ICWG-2302_WG.html

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

[22] Malik S.: Network Security Principles and Practices. Cisco Press, 2003.

[23] Martin R.C.: Design Principles and Design Patterns. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[24] Mell P., Grance T.: The NIST Definition of Cloud Computing. NIST, 2009.

[25] Meszaros G., Doble J.: A Pattern Language for Pattern Writing.. Pattern Languages of Program Design, 1998. http://hillside.net/index.php/a-pattern-language-for-pattern-writing

[26] Microsoft Case Studies: Daimler: Auto Maker Uses Scalable Cloud Solution to Deliver Internet Services for Electric Car, 2011. http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?CaseStudyID=4000008606

[27] Microsoft: Windows Azure AppFabric Overview, 2011. http://www.microsoft.com/windowsazure/appfabric/

[28] Microsoft: Windows Azure AppFabric Service Bus Tutorial, 2011. http://msdn.microsoft.com/en-us/library/ee706736.aspx

[29] Microsoft: Windows Azure, 2011. http://www.microsoft.com/windowsazure/

[30] Mietzner R., Leymann F.: A Self-service Portal for Service-based Applications. Service-Oriented Computing and Applications (SOCA), 2010.

[31] Mietzner R., Unger T., Leymann F.: Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE, 2009.

[32] Mietzner R., Unger T., Titze R., Leymann F.: Combining different multi-tenancy patterns in service-oriented applications. Proceedings of the 13th IEEE International Conference on Enterprise Distributed Object Computing, 2009.

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

[34] Naghshineh M., Ratnaparkhi R., Dillenberger D., Doran J. R., Dorai C., Anderson L., Pacifici G., Snowdon J. L., Azagury A., VanderWiele M., Wolfsthal Y.: IBM Research Division cloud computing initiative. IBM Journal of Research and Development, 2009.

[35] Petre M.: Why Looking isn't Always Seeing. Communications of the ACM, 1995.

[36] Riempp G., Gieffers-Ankel S.: Application portfolio management: a decision-oriented view of enterprise architecture. Information Systems and e-Business Management, 2007.

[37] Scheibler T., Mietzner R., Leymann F.: EAI as a Service - Combining the Power of Executable EAI Patterns and SaaS. 12th International IEEE Enterprise Distributed Object Computing Conference, 2008.

[38] Staten J., Yates S., Gillett F. E., Saleh W., Dines R. A.: Is Cloud Computing Ready For The Enterprise? Forrester Research, 2008.

[39] 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

[40] Tanenbaum A.S., Van Steen M.: Distributed Systems: Principles and Paradigms. Prentice-Hall, 2003.

[41] T-Systems: Dynamic Services for Infrastructure, 2011. http://www.telekom.com/dtag/cms/content/dt/en/596392?archivArticleID=953552

[42] Use Case Discussion Group: Cloud Computing Use Cases White Paper, Version 2.0, 2009. http://www.opencloudmanifesto.org/Cloud_Computing_Use_Cases_Whitepaper-2_0.pdf

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

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

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

[46] WSO2: WSO2 Enterprise Service Bus Documentation, 2011. http://wso2.org/project/esb/java/3.0.1/docs/

[47] WSO2: WSO2 Enterprise Service Bus, 2011. http://wso2.com/products/enterprise-service-bus/

All links were last followed on 23rd September 2011.