Future Internet **2012**, *4*, 110-141; doi:10.3390/fi4010110

OPEN ACCESS

future internet

ISSN 1999-5903

www.mdpi.com/journal/futureinternet

Article

Pattern-Based Development and Management of Cloud Applications

Christoph Fehling [1],*, Frank Leymann [1], Jochen Rütschlin [2] and David Schumm [1]

[1] Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, Stuttgart 70569, Germany; E-Mails: leymann@iaas.uni-stuttgart.de (F.L.); schumm@iaas.uni-stuttgart.de (D.S.)

[2] Daimler AG, Epplestraße 225, Stuttgart 70546, Germany; E-Mail: jochen.ruetschlin@daimler.com

* Author to whom correspondence should be addressed; E-Mail: fehling@iaas.uni-stuttgart.de; Tel.: +49-711-685-88486; Fax: +49-711-685-88472.

Received: 28 November 2011; in revised form: 18 January 2012 / Accepted: 3 February 2012 / Published: 15 February 2012

Abstract: Cloud-based applications require a high degree of automation regarding their IT resource management, for example, to handle scalability or resource failures. This automation is enabled by cloud providers offering management interfaces accessed by applications without human interaction. The properties of clouds, especially pay-per-use billing and low availability of individual resources, demand such a timely system management. We call the automated steps to perform one of these management tasks a “management flow”. Because the emerging behavior of the overall system is comprised of many such management flows and is often hard to predict, we propose defining abstract management flows, describing common steps handling the management tasks. These abstract management flows may then be refined for each individual use case. We cover abstract management flows describing how to make an application elastic, resilient regarding IT resource failure, and how to move application components between different runtime environments. The requirements of these management flows for handled applications are expressed using architectural patterns that have to be implemented by the applications. These dependencies result in abstract management flows being interrelated with architectural patterns in a uniform pattern catalog. We propose a method by use of a catalog to guide application managers during the refinement of abstract management flows at the design stage of an application. Following this method, runtime-specific management functionality and management interfaces are used to obtain automated management flows for a developed application.

Keywords: cloud computing; distributed application; systems management

1. Introduction

Cloud computing has drastically changed the way in which we consume IT resources. Unlike traditional data centers, clouds offer *elasticity*—the ability to reserve and free resources flexibly (often within minutes); *pay-as-you-go*—users only pay for the resources they actually consume; and *standardization*—through the use of virtualization and the offering of runtime platforms by providers, application middleware and hardware stacks are commoditized. Because of these new environmental properties, the architecture and management of applications have to be adjusted as well. However, the fast and mostly industry-driven evolution of cloud offerings often obfuscates the common concepts of offered solutions. Due to different requirements of applications and the components they consist of, offerings of different cloud providers have to be integrated in many cases. Using only one cloud provider for the complete application landscape of a company is, therefore, often impossible.

We consider cloud applications to be divided into loosely coupled application components so that individual application components can use different cloud offerings which best fit their requirements. Applications components are developed specifically for one cloud offering and requirements are often not explicitly specified. Exchanging cloud offerings after the initial development of the application, therefore, remains a challenge. To describe requirements of application components in a more generic form, we have abstracted the development guidelines for specific cloud providers. The obtained common architectural principles were compiled into a uniform *pattern format* in [1] and are available online (http://cloudcomputingpatterns.org). These patterns describe good solutions to common problems in the area of cloud computing as well as clouds and their offerings. Especially, they allow the classification of cloud providers regarding the patterns they support and, thus, ease application development and requirements management.

Building on these patterns, describing architectural concepts of cloud application development, we now focus on the runtime management of cloud applications. The runtime management of applications is often comprised of manual tasks that are performed according to implicit knowledge of application managers. Such tasks are, for example, adequate resource adjustments to changing workloads or reactions to failing system resources. Cloud computing has introduced additional challenges to the runtime management of such applications: (i) its use is often most beneficial, if application management is automated, for example, to exploit pay-as-you-go pricing models more efficiently; (ii) to address the often low availability of individual cloud resources, an automation of resiliency management is required to address resource failure [2]; and (iii) vendor lock-ins can be avoided if application components can be moved automatically between different environments.

The remainder of this article is structured as follows. In Section 2, we align the new management challenges of cloud applications with existing IT management approaches and give examples of how today's cloud providers address and automate them. Existing work on the offering process of cloud application is also covered in this section. We extend this process to incorporate the definition of automated management flows as an integral part of application development. In Section 3, we then give an overview of existing patterns that we organized in a catalog. Section 4 introduces a new pattern class for cloud application management patterns to be included in this catalog. Just as architectural patterns are used to guide the implementation of applications, management patterns will guide the creation of automated management processes. Section 5 covers annotations of implementation artifacts to patterns to guide application developers and application managers. For example, we propose to annotate information about management interfaces to be used during the creation of automated management flows. We argue that this respects the flexibility and speed that cloud computing has introduced to systems management tasks, such as provisioning, deprovisioning, and scaling of resources. Further, it enables application managers to address future challenges, for example, to automate the replacement of a cloud provider with a different one or with an in-house data center during the design time. Such annotations can also help to standardize pattern implementation which enables companies to control the allowed runtime environments. The otherwise deployment of applications on arbitrary hard- and middleware increases the management complexity drastically [3]. In Section 6, we employ the proposed approach in an end-to-end example describing the pattern-based development of a web shop application. Especially, we show how patterns can be used to express requirements of application components as part of an architecture diagram. We cover how management flows can be created in a standardized fashion by refining abstract management flows to executable ones. Finally, Section 7 covers limitations of the approach and gives an outlook on the future research that will address them.

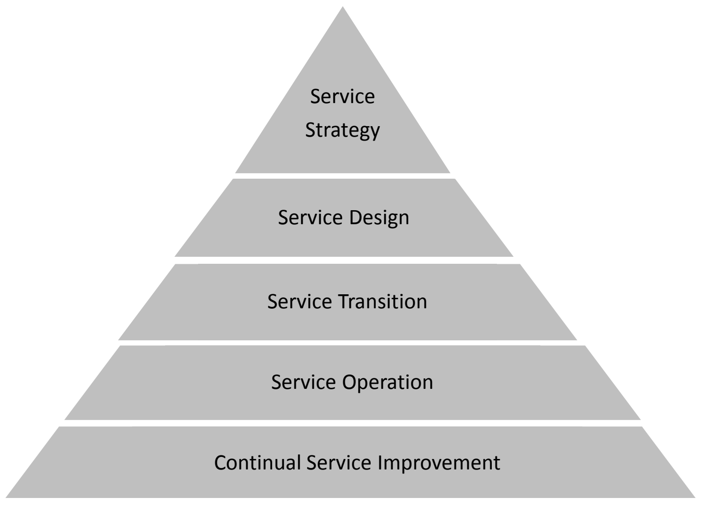
2. Offering and Managing Cloud Applications

In this section, we align the proposed abstract management flows with other management efforts for IT systems. Further, we extend an existing *cloud application offering process* with management considerations and introduce a set of management challenges which arise in today’s cloud applications.

2.1. ITIL-Based Management of IT Systems

The IT Infrastructure Library (ITIL)-Based Management of IT Systems [4] standardizes IT management efforts. In this library a set of common practices and processes are described to strategically plan, design, develop, operate and improve IT services. These different aspects of IT service management and their relations are depicted in Figure 1. On the very top is the *service strategy* describing how to strategically plan IT services, their abstract specification and how to address financial management aspects. The *service design* of IT services concretizes strategies into service specifications and a general architecture of the IT system. During the *service transition* this general architecture and specification is realized in form of an implemented IT service that can be offered to other companies or be used internally. After this development of concrete services, required operational processes are specified by the *service operation* that can be used, for example, to ensure the required availability and performance of managed IT services. Finally, *continual service improvement* considers methods to evaluate existing IT services and improve their behavior. Such improvements may consider cost, performance, ecological footprints, *etc*. To put the architectural patterns and management flows presented in this paper into perspective with the ITIL standard, the architectural patterns and management flows here can be considered as a technical concretization of the abstract processes described in ITIL. The presented pattern catalog and an exemplary set of management flows aim at guiding application developers during the automation of some of the ITIL management aspects regarding *service design*, *service transition*, and *service operation*. During *service design*, the architectural patterns provide a graphical notation for each pattern to be used in architectural diagrams to ease architecture discussions. These discussions are additionally supported by patterns providing a common vocabulary to focus on concepts rather than on products. The patterns may further be used during *service transition* to express requirements and desired service behavior. Since the staff implementing an application often differs from those designing them, patterns ease a clear specification of requirements. Finally, the management patterns introduced in this paper, directly address best practices during *service operation* and guide the automation of this management phase.

Figure 1. IT Infrastructure Library (ITIL) Management Aspects (adapted from [4]).



2.2. Common Management Challenges in Cloud Applications

Because the quality of service and self-management capabilities of today’s computer systems are both very high, human error has become the most significant reason for system downtime in large distributed systems [5,6]. The management flexibility introduced by cloud computing has additionally increased this effect: management tasks, such as the provisioning of new virtual machines or the deprovisioning of unused ones is literally at the systems managers’ fingertips and can be handled completely via management interfaces of cloud providers, such as [7] and [8]. Due to this evolution of management interfaces, human errors are more likely to occur, since there is hardly any notion of physical machines or an underlying physical connection network and management tasks can be performed much easier and quicker. Significant effort is required to structure the abstract view on managed resources in management interfaces [9].

However, the most effective way to avoid human errors during systems management is the automation of management tasks leaving only the decision when to trigger them up to human application managers. Regarding the scaling of cloud applications and to enable their resiliency towards failing cloud resources, significant automation is being introduced by cloud providers and cloud application developers [2,10]. Automation of these tasks is fundamental to the successful use of cloud resources, because it enables cloud applications to benefit best from pay-per-use pricing models and automation also addresses the often low availability of individual cloud resources. Handling these tasks manually would make cloud application too expensive to operate and it would most likely result in unacceptable availability assurances. Other management tasks, such as the update of applications or the migration of applications to new environments are, however, not automated to a similar degree.

We now describe how elasticity, resiliency, and the move of applications may be handled in cloud applications and motivate further why an automation of these management tasks would be beneficial. We cover the current state of the industry regarding these automation efforts.

Amazon AWS offers monitoring of running virtual machines to analyze central processing unit (CPU) usage, memory usage, request processing time *etc*. Additionally, running virtual machines may publish customized monitoring information and metrics [11]. Based on this information, triggers may be defined to provision new virtual machines or reduce their number. This functionality may be handled for elasticity to scale-out cloud resources. The identification of failing resources, however, may require custom monitoring on the application level, because in many cases not all application failures can be identified correctly on the system level. Additionally, the application developer has to carefully configure the behavior of the systems. Amazon's elasticity management, Cloud Watch, covers the configuration of: (i) the minimum number of virtual machines of a certain type; (ii) the maximum number; (iii) the number of machines to be provisioned and deprovisioned in case a condition occurs; and (iv) how often to check for such conditions. The actual behavior, and especially the success of systems management using this functionality, is highly dependent on this configuration. Consider, for example, a web shop that should scale-out automatically. The application developer therefore monitors the average CPU utilization of virtual machines every five minutes and specifies a trigger that provisions a new instance when the CPU utilization is above 80%. This configuration may be sufficient to scale the application most of the time, but fails when the load of the application increases drastically within very short time frames. In such a case, more machines should be provisioned to adequately react to a workload peak. Only provisioning one new instance every five minutes as specified in the example configuration simply takes too long. Even if a cloud provider offers advanced scaling and resiliency functionality, the adequate configuration of this functionality is still a management challenge.

Microsoft Windows Azure [12] offers a monitoring functionality similar to that of Amazon. Therefore, systems level behavior may be monitored in a similar fashion. How to react to the monitored information has to be programmatically implemented by the application developer, whereas the behavior is merely configured when using Amazon.

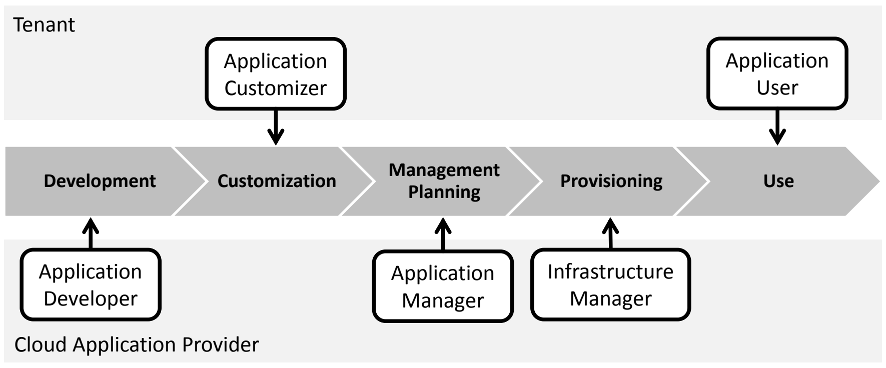
Moving applications from one cloud provider to another or back into a traditional data center can be an even more challenging management task. Even though significant efforts are made by the industry to standardize cloud interfaces and used formats [13–16], cloud providers are not generally interchangeable. However, the need to switch providers may arise under many circumstances. Changes in the pricing model of a provider making other providers a better fit economically is only one factor in this scope. Changes to requirements or adherence to new regulations and laws also have to be handled. The scenario in which a used cloud provider goes out of business or terminates a used service also has to be considered. Migration may also be an issue if different computing environments are to be used during the lifecycle of an application. For example, Amazon may be used for application development. Afterwards, the application shall be hosted in a more secured internal environment.

While these management tasks may also arise in non-cloud applications, the faster provisioning and deprovisioning capabilities of cloud computing should be respected. Therefore, the speed at which applications and their administrators can react to environmental changes should be increased to match the flexibility of clouds. We argue that this can be achieved by standardization of management processes and their capturing in automated management flows during the design time of application.

2.3. Offering Customizable Multi-Tenant Cloud Applications

In [17] we introduced a *cloud application offering process*. Here, we extend this process by stakeholders and activities handling the automation of management tasks of cloud applications. The new cloud application offering process is depicted in Figure 2. We assume that a cloud application provider offers an application to multiple tenants. In [17] we described a development style for such applications enforcing application componentization and a specific component format to be used during the development phase of such applications. This development style enabled the customization of the application to a tenant's specific needs. In detail, we addressed the following application variability: *user interface variability*—the look and feel of the application can be customized; *functional variability*—tenants may alter the processes supported by the application; *data variability*—tenants may define custom data objects and queries in the application; *provisioning variability*—the application can be deployed on a variable set of hard- and middleware and on different clouds. After the customization phase, the application's functionality and the desired runtime environment has been specified by the application customizer. Prior to the provisioning phase, in which the infrastructure manager deploys the application to the desired runtime environments, we added an additional management planning phase. During this phase, an application manager defines automated management processes to handle management tasks arising while the application is used. The management tasks covered in this paper and the creation of automated management flows are addressed during this phase of the cloud application offering process. While the process considers cloud applications to be offered to multiple tenants, it may also be employed for applications that are only deployed once. In this case, the customization phase may be omitted, since the application only supports a fixed set of functions and is deployed only into one environment, *etc*. The remaining phases of the cloud application offering process are still applicable in the same form.

Figure 2. Extended Cloud Application Offering Process.



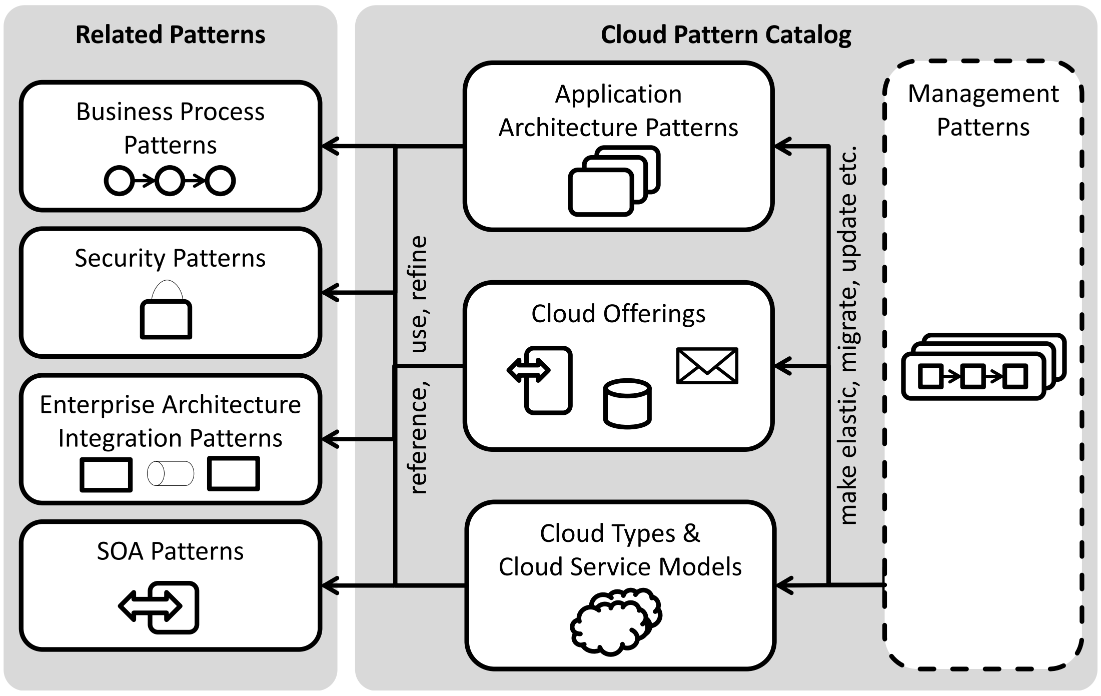
3. Organization of Cloud Patterns in a Catalog

In the following, we give an overview of the cloud architectural patterns we identified and the interrelations between the different pattern classes for cloud types, cloud service models, cloud offerings, and cloud architectures. We cover the format used to describe patterns and show how the newly introduced management pattern class fits into the catalog. Further, we describe how other existing cloud architectural patterns and those targeting other domains may be incorporated to form a homogeneous presentation in the catalog.

3.1. Overview of the Pattern Catalog

In [1] and [18], we introduced a catalog of architectural patterns to guide developers during the design of cloud applications. We also used the uniform pattern format to describe different cloud types and their offerings. Especially, we identified existing patterns from other domains, such as message-based application [19] or standalone application [20] that are useful in the area of cloud computing. These patterns were altered if necessary to respect the specifics of cloud environments and were expressed in the same pattern format as the rest of the catalog. Such a homogenization of information in a common format eased perception for application developers [21]. Additionally, the environment in which a pattern may be applied can be specified more easily, because pattern descriptions for cloud types, cloud offerings, and cloud service models may be used to set its context. Other existing architecture patterns that target the domain of cloud computing or other related domains were referenced with patterns contained in the catalog. The catalog structure depicted in Figure 3is divided into four sections: *cloud types*, *cloud service models*, *cloud offerings*, *cloud application architectural patterns*, and the newly introduced class for *cloud management patterns* (dashed lines). This new class contains patterns that describe how cloud applications developed according to the other patterns may be managed after their deployment. Existing architectural patterns identified by others are referenced from patterns contained in the catalog, if they describe good solutions to problems arising during the application of a pattern. For example, many of identified cloud architectural patterns face security challenges. Most of these are equivalent to security challenges in non-cloud applications which have been expressed as patterns [22]. To provide a linkage between these existing security patterns and the cloud patterns in the catalog informal references are made. Other security issues arise specifically due to the use of cloud computing mainly due to the sharing of cloud resources with other cloud users. For example, there have been security issues in the management interfaces of Amazon AWS [23] allowing other users to hijack other user accounts [24]. In this scope, legal implications are also quite different due to cloud computing, because providers may be legally responsible for employees but not for other users. Since significant work already exists on patterns describing the misuse of cloud computing [25], we did not compile this information into the used pattern format but referenced them in the cloud patterns catalog. Further domains for which we found existing patterns related to or used in cloud computing are also depicted in Figure 3. Messaging patterns as defined by [19] are often used to enable asynchronous communication in the cloud to loosely couple application components. This componentization is also a fundamental concept of the service-oriented architecture (SOA) patterns described by [26]. Similar componentization can be found in object-oriented programming defined by patterns in [20]. The management patterns introduced in this paper contain a management flow that may be modeled according to business process patterns described in [27].

Figure 3. Pattern Classes Comprising the Pattern Catalog and their Relations to Existing Patterns.


Related Patterns
Cloud Pattern Catalog
Business Process Patterns
Security Patterns
Enterprise Architecture Integration Patterns
SOA Patterns
Application Architecture Patterns
Cloud Offerings
Cloud Types & Cloud Service Models
Management Patterns
reference, use, refine
make elastic, migrate, update etc.

We propose to use this catalog of patterns to coordinate the work of the different stakeholders during the cloud application offering process described in Section 2.3. Using a decision recommendation table, introduced in [18], the application developer selects patterns describing the environment for which he wishes to create an application. By evaluating relations among the patterns, architectural patterns describing development guidelines are recommended to him for implementation. He selects the ones he wishes to use and implements application components. Based on the implemented patterns, the application manager is provided with a set of recommendations for management patterns describing abstract management flows to automate the management of the application components.

In the following sections, we will introduce the different pattern classes used by application developers and application managers during the cloud application offering process. We give an overview of existing patterns we described for clouds and architectural guidelines used by application developers and cover management patterns addressing the cloud-specific management challenges introduced in Section 2.2.

3.2. Pattern Format Used in the Catalog

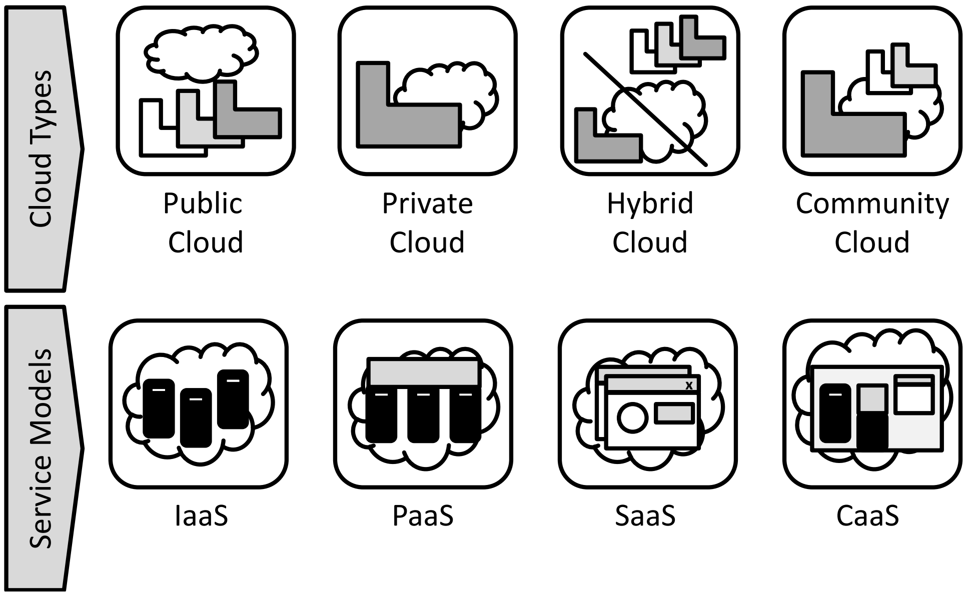
The patterns in the catalog follow a uniform format to increase readability and comprehensibility. This format is often used by the pattern community and is inspired by [19,20,26]. Each pattern is identified by a small *icon* depicting the essence of the pattern. This icon can also be used in composite patterns to make a graphical reference to the used pattern. The pattern description is then introduced by a *driving question*, which is answered by the pattern. This allows application developers to quickly identify patterns which solve the questions at hand. The *context* section describes the environment in which the pattern can be applied in greater detail. Uniform descriptions of cloud types, cloud service models, and cloud offerings in the same pattern format was especially motivated so they can be used in the context section to set the environment cloud architectural patterns. The context section is followed by a detailed description of *challenges* that are addressed by the pattern. How this is done is briefly stated in the following *solution* section, giving developers steps to follow. The solution is supported by a *sketch* that depicts an overview of the architecture described by the pattern. In this sketch, the icons of other patterns may be used to describe a pattern composing other patterns. In the following *result* section, the outcome after application of the pattern is discussed as well as new challenges that might have to be addressed. *Variations* of the pattern are discussed next. These variations are slightly different applications of the pattern, but changes are not significant enough to justify their description in an independent pattern. Then, *references to other patterns* are given that solve similar problems, or are likely to be combined with the discussed pattern, *etc*. Finally, *known uses* of the pattern are referenced.

3.3. Existing Patterns for Cloud Types and Cloud Service Models

To describe the different environments in which cloud applications may be hosted, we described different cloud types in the pattern format. Figure 4 depicts the icons of defined cloud types. Resources of *public clouds* are available to everyone, which often raises concerns regarding privacy, security, and trust. *Private clouds* on the other hand are hosted exclusively for a company. *Community clouds* are in between these two extremes offering resources only to a certain group of companies, for example, to a car manufacturer and its suppliers. Finally, a *hybrid cloud* is formed by an integration of at least two clouds of the other types. This is mostly done if one cloud alone cannot handle all requirements of a company.

Clouds offer resources in very different styles leaving more or less control of the hard- and middleware to the application developer. For each of these cloud service models, we gave a description in the pattern format: *Infrastructure as a Service (IaaS)*—the offering of (virtual) servers to customers; *Platform as a Service (PaaS)*—the offering of a provider-controlled middleware that can be used by customers to host their applications; *Software as a Service (SaaS)*—providing complete applications that can only be customized by users; and *Composition as a Service (CaaS)*—the offering of a configurable composition of cloud offerings of different cloud service models.

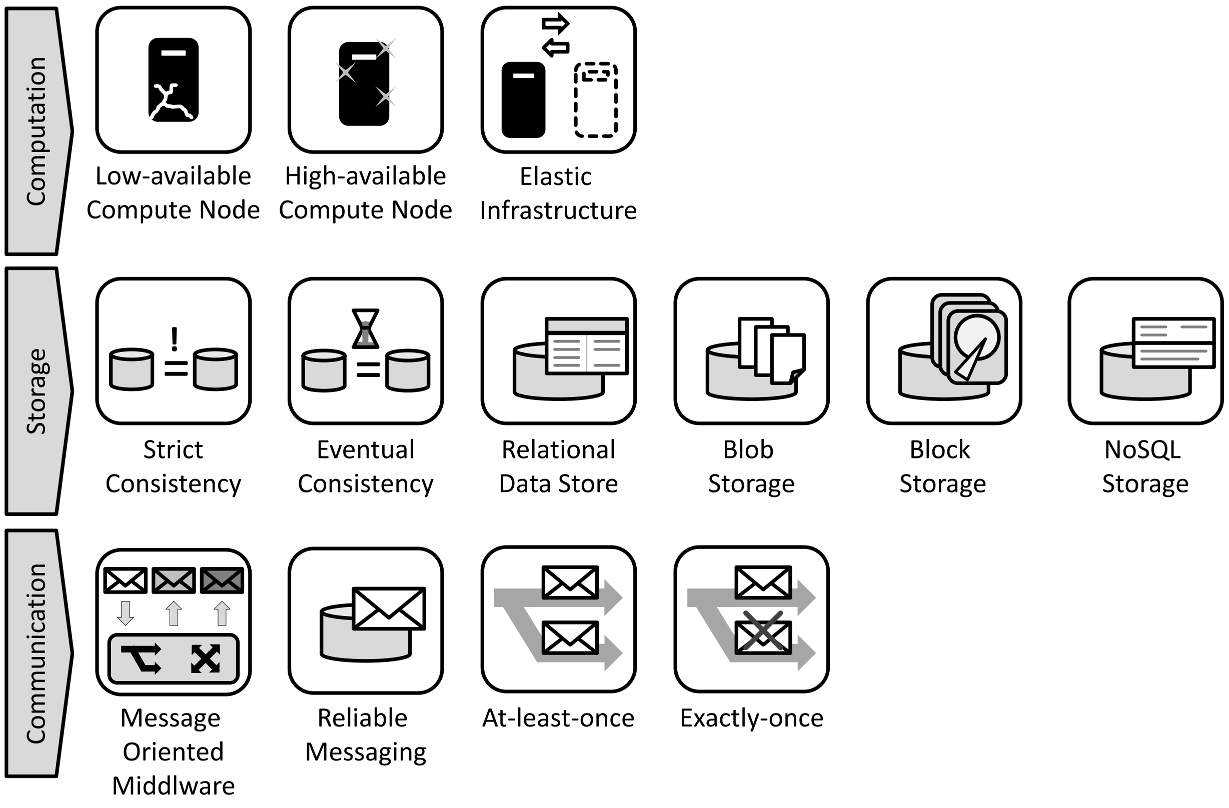
Figure 4. Icons of the Cloud Types and Cloud Service Model Patterns (adapted from [1]).



3.4. Existing Patterns for Cloud Offerings

While cloud types and cloud service models describe the cloud environments, cloud offerings describe the behavior of services available in them. We differentiated these offerings into three subclasses as depicted in Figure 5.

Figure 5. Icons of Cloud Offerings (adapted from [1]).



Cloud compute offerings provide resources to handle the actual computation workload of applications. They are often hosted in an *elastic infrastructure* that allows them to be flexibly scaled. Further, we differentiated between *low-available* and *high-available compute nodes* since this resource property significantly affects how applications using these resources have to be built.

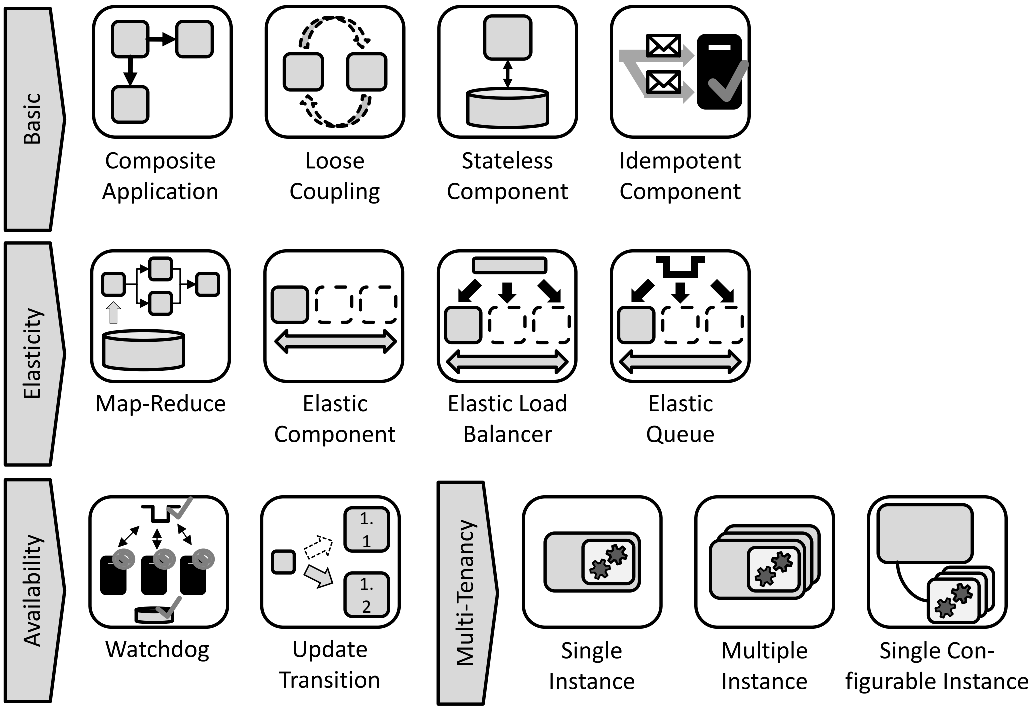
The second class of cloud offerings are *cloud storage offerings* that can be used to store data in the cloud. We described *relational data stores*—a table-based store that ensures a defined data schema [28] as well as a certain data structure, *blob storage*—a data store for large unrelated files, *block storage*—a cloud storage service that can be used similar to physical hard drives; and *NoSQL storage*—another table-based data store relaxing data consistency and schema regulations. Consistency behavior of these storage services, which can be *strict* or *eventual*, was also described in a pattern-form.

The third cloud offering class consists of *cloud communication offerings*, which can be used to exchange data between distributed applications or their individual application components. We described messaging services as well as their behavior that has a significant impact on the application using messaging. We included existing patterns for *reliable messaging* [19], also called transactional messaging and *messaging systems* [19] not offering transactional behavior. Further, we included patterns for the delivery behavior, which may be either *exactly-once* [19] or *at-least-once* [19] referring to delivering a sent message once and only once or possibly multiple times respectively.

3.5. Existing Patterns for Cloud Application Architectures

So far, the pattern classes have described the environment offered by cloud providers and the offerings available therein. Cloud application architecture patterns guide application developers during the design and implementation of applications components that use cloud offerings and are deployed to different cloud types. Again, we introduced several sub-classes in the catalog as depicted in Figure 6.

Figure 6. Icons of Cloud Application Architecture Patterns (adapted from [1]).



Basic architectural patterns give general advice to follow when designing applications for the cloud. The concepts of dividing an application into a *componentized application* and avoiding dependencies between these components resulting in *loose coupling* has been introduced by service oriented computing [29]. The pattern to avoid having an internal state in these components to make them *stateless* and better manageable is also based on this computing paradigm. Duplicate messages of used cloud offerings providing at-least-once delivery can generally be handled by implementing *idempotent components* adapted from [19].

Elasticity patterns address the new challenges of cloud applications to "breathe" depending on the current workload. We covered different styles to measure this workload based on the load experienced by individual *components*, *load balancers*, or *queues* to determine and provision the number of required resources. Further, we described *map-reduce*, introduced by [30], in the pattern format. Map-reduce is a divide-and-conquer approach for distributed computing and is frequently used for large-scale data analysis.

The insufficient availability of cloud resources can be addressed by implementing *availability patterns*. These describe how resources may be monitored by a *watchdog* [31] and how components may be designed to be *transitioned* to a new *updated* version. The last sub-class describes how application components may be shared by multiple tenants. This so-called multi-tenancy [32] can either be realized in a *single instance* of an application component. Alternatively, component instances may also be *configurable* for different tenant or *multiple instances* may be hosted to serve each tenant separately.

4. Cloud Application Management Patterns

Previously discovered patterns for cloud architectures, cloud offerings, cloud types, and cloud service models have been described in Section 3. In this section, we introduce a new pattern class, *cloud application management patterns*. While cloud architecture patterns mentioned in Section 3.5 describe how application components shall be designed and interconnected, patterns of this class describe how application components should be managed. Therefore, the patterns of this class describe cross-cutting concerns how to manage cloud applications.

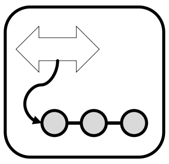
The management patterns follow the same pattern format described in Section 3.2 to seamlessly integrate into the catalog. The only difference is that the sketch does not depict an abstract architectural diagram but an *abstract management flow* using Business Process Model and Notation (BPMN) [33]. This is due to the fact that management patterns are not implemented as application components but in the form of automated management flows.

In some cases, a specific system architecture is required for a management pattern to be applicable. For example, the following *elasticity management* pattern is likely to be combined with the application architecture patterns *elastic component*, *elastic load balancer*, or *elastic queue*. This is expressed through interrelations between the patterns rather than including duplicate descriptions of management flows in every architectural pattern. Another advantage of this separation is that the creation of application components and management processes performed by different roles of the cloud application offering process described in Section 2.3. In the following, we present management patterns we discovered to handle the cloud-specific management challenges mentioned in Section 2.2.

4.1. Elasticity Management Pattern

Icon: the icon of the *elasticity management pattern* is depicted in Figure 7.

Figure 7. Icon of the Elasticity Management Pattern.



Driving Question: *How can the number of resources to which application components are scaled-out be adjusted efficiently to the currently experienced workload?*

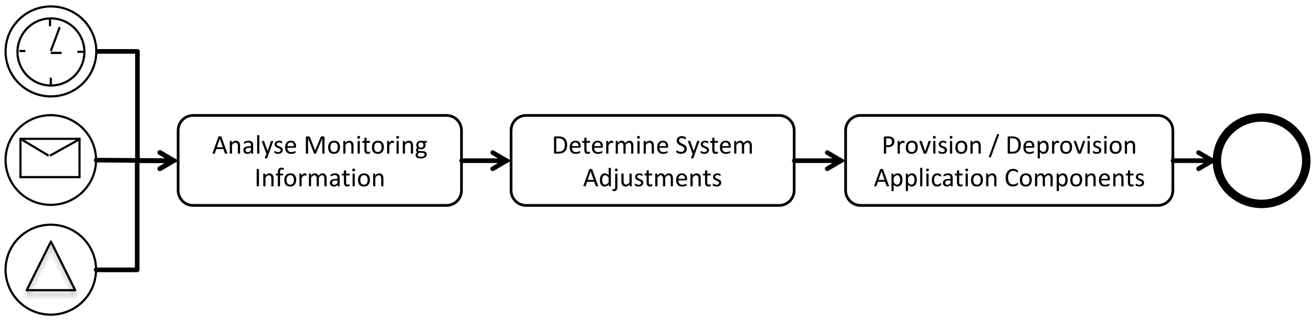
Context: a *componentized application* is hosted on an *elastic infrastructure* and comprised of *elastic components*, *elastic queues*, or *elastic load balancers*.

Challenges: the dynamicity of clouds demands automatic scaling of cloud resources. To perform this task, the current resource demand has to be determined and has to be reflected in provisioning and deprovisioning of system resources.

Solution: analyze the current utilization of system resources after certain time intervals, when a user requests it, or based on monitored system events to determine the current workload and adjust resource numbers accordingly.

Result: the elasticity management flow can be triggered periodically (every day, month *etc*.) depicted by the timer event in Figure 8. Alternatively, a user of the application can trigger it, for example, because he knows that a number of new employees will start using the application at a certain date. This case is reflected by a message event passed to the management flow. As a third option, the flow can be triggered by signals originating from monitoring. For example, the size of a queue may exceed a certain threshold if this management pattern is combined with the *elastic queue pattern*.

Figure 8. Abstract Management Flow of the Elasticity Management Pattern.


Analyse Monitoring Information
Determine System Adjustments
Provision / Deprovision Application Components

When the elasticity management flow is triggered, resource utilization is optimized and resource numbers are adjusted to correctly reflect the workload. Critical design decisions in this scope are:

1. Time interval at which system utilization is evaluated (in case of time-based triggers): if this interval is too large, the system may be underutilized or overloaded without notice.

2. Reactions to workload analysis: the reaction must be appropriate, for example, if too few resources are added to the system, it takes too long until an increased workload can be handled by the application.

Both design decisions mainly depend on heuristics and prior experiences. When determining the time interval at which utilization is measured, it has to be considered how quickly utilization has changed in the past. Also, the time it takes to provision new resources has to be considered here. This is also an important factor when determining how a change in the utilization should be addressed. Historic information, such as user behavior during holidays may be used to adjust these variables [34].

Relations to other patterns: the elasticity management pattern can be combined with *elastic components, elastic queues,* or *elastic load balancers*. These patterns form the architectural basis for the elasticity management flow by providing the required monitoring information that has to be extracted from the managed application components and their runtime environment.

Variations: manual triggering of the introduced management flow may not only originate from application users. Who is responsible for this task mainly depends on the employed *cloud service model*. In case of *IaaS*, the user of the cloud likely performs this task. However, in case of *PaaS* and *SaaS*, the configuration of elasticity management may be hidden from the user. In this case, the task instead may be performed by the cloud provider.

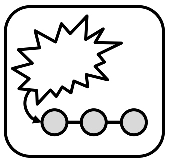
Known Uses: [2] describes how to scale Amazon AWS Resources. Especially, it covers different events and conditions when the elasticity management should be executed. [35] evaluates the scaling capabilities of Windows Azure, for which [36] offers a scaling software as a service. The concept to scale-out applications automatically is, however, not only seen in cloud computing and has been used by the industry for quite some time [31].

Self-adaptive autonomous systems perform a similar management task to adjust their size, structure, communication channels, *etc*. regarding environmental conditions. They undergo a so-called monitor-analyze-plan-execute (MAPE) loop [37] comprised of similar steps as the elasticity management flow.

4.2. Resiliency Management Pattern

Icon: the icon of the *resiliency management pattern* is depicted in Figure 9.

Figure 9. Icon of the Resiliency Management Pattern.



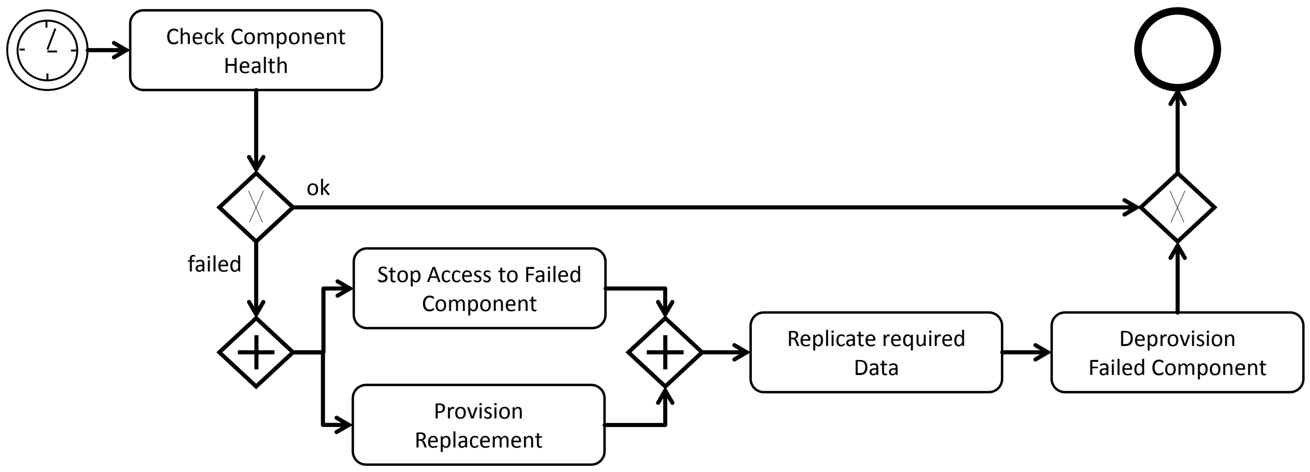
Driving Question: *How can the availability of a composite application be ensured even if individual components fail?*

Context: a *composite application* that distributes *application components* among different *low-availability compute nodes* offered by an *elastic infrastructure*.

Challenges: if an application depends on the availability of many individual resources, the overall availability of the application is may be reduced drastically. This is due to the fact that the chance that at least one of the resources fails is higher, the more resources are used. Therefore, it has to be ensured that individual resources may fail without affecting the availability of the overall application.

Solution: scale-out application components among multiple resources. Ensure that a sufficient threshold of component instances is provisioned exceeding what the currently experienced workload requires, monitor and react to component instance failures according to the abstract management flow depicted in Figure 10.

Figure 10. Abstract Management Flow of the Resiliency Management Pattern.



Results: the failure of component instances can be detected and automatic reaction is enabled. Critical design decisions in this scope are:

1. Time intervals at which the states of component instances are checked.
2. Method for failure detection.

The time intervals mainly depend on the required recovery time and, therefore, have to respect provisioning time of resources. Often, cloud providers do not make assurances for these times and heuristics have to be employed to predict them. To detect a component failure, providers may offer monitoring of network availability, CPU or memory utilization, for example [11]. The application manager, however, is obligated to interpret these values and deduct information about application component availability from them. Further, none of this information can assure that the component functions correctly on the application level. Such tests have to be implemented manually by the application developer.

Variations: sometimes, failures are hard to detect on the application level. Especially, testing how application components behave after a very long runtime can be challenging. To address these challenges, application components can be randomly treated as failed after certain time intervals and are then replaced by newly provisioned instances.

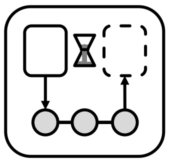
Relations to other patterns: the components themselves should be implemented as *stateless components* to simplify their management within the scope of this pattern. The *resiliency management pattern* may be combined with the *elasticity management pattern* if elasticity is also an issue.

Known uses: The concept has been used to enable high availability of processes running on a single machine. In this scope, the system component handling the above described management flow is called a watchdog to which system components notify their availability [38]. Amazon suggests a similar approach to assure fault tolerance in applications using their Elastic Beanstalk service [39].

4.3. Move/Update Management with Downtime Pattern

Icon: the icon of the *move/update management with downtime pattern* is depicted in Figure 11.

Figure 11. Icon of the Move/Update Management with Downtime Pattern.



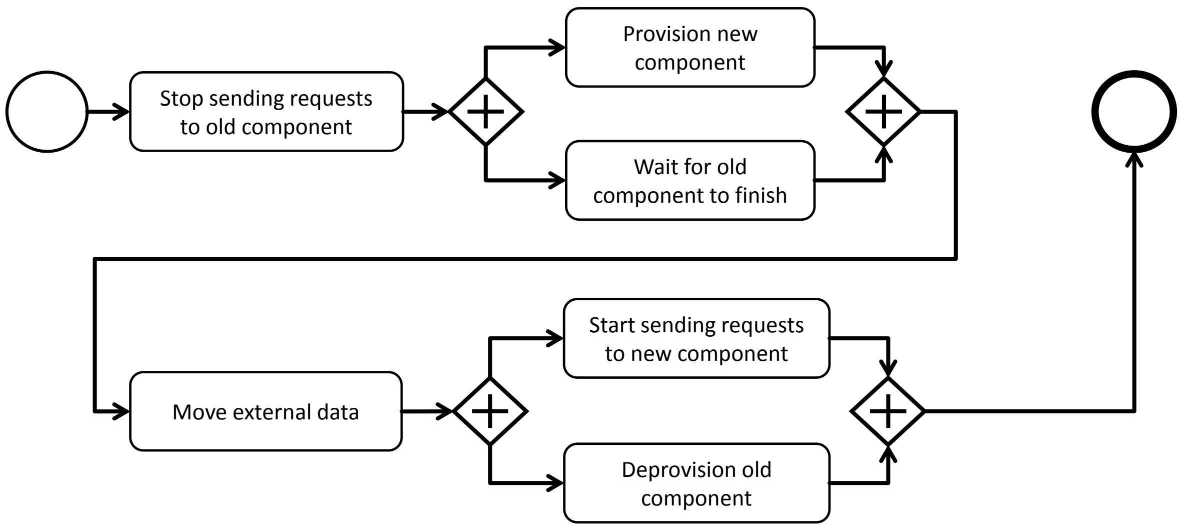
Driving Question: *How can an application component be moved to a different environment or updated to a new software, middleware, or hardware version, if downtime is acceptable?*

Context: a *stateless application component*, which has no internal state but, relies on external data and is part of a *composite application*. This component may be unavailable for the timeframe in which it is updated/moved.

Challenges: during the transition it has to be ensured that application components are idle prior to their deprovisioning. Further, accesses either have to be stopped on the application level or have to be queued during the downtime.

Solution: as depicted in Figure 12, ensure that no new requests are sent to the component to be moved or updated. Then, provision a new component instance with the new version (update) or in the new runtime environment (move). Send requests to the new component when it becomes available and deprovision the old one once it has finished processing requests.

Figure 12. Abstract Management Flow of the Move/Update Management with Downtime Pattern.


Stop sending requests to old component
Provision new component
Wait for old component to finish
Move external data
Start sending requests to new component
Deprovision old component

Result: the provisioning of the new component and the deprovisioning of the old component can partly be parallelized. Problems may arise if the time required for the migration/update is hard to estimate or depends on external factors, such as the behavior of the used cloud provider. This is due to the fact that the acceptable downtime is often fixed, for example, it could be limited to night hours.

Variations: if the stateless component does not rely on external data, concurrently provisioning both versions of the component is likely to be possible ensuring a very timely switchover.

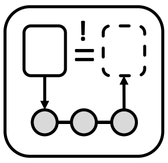
Relations to other patterns: a migration/update can also be realized without downtime as described by the *migration/update management without downtime pattern*. The managed components are often accessed via messaging queues [19].

Know Uses: Peecho [40], offering printing as a service, is based completely on Amazon AWS. An overview how it handles updates with minimal downtime is given by [41]. Kununu [42] also automated this management flow based on Amazon AWS [43].

4.4. Move/Update Management Without Downtime Pattern

Icon: the icon of the move/update management without downtime pattern is depicted in Figure 13.

Figure 13. Icon of the Move/Update Management without Downtime Pattern.



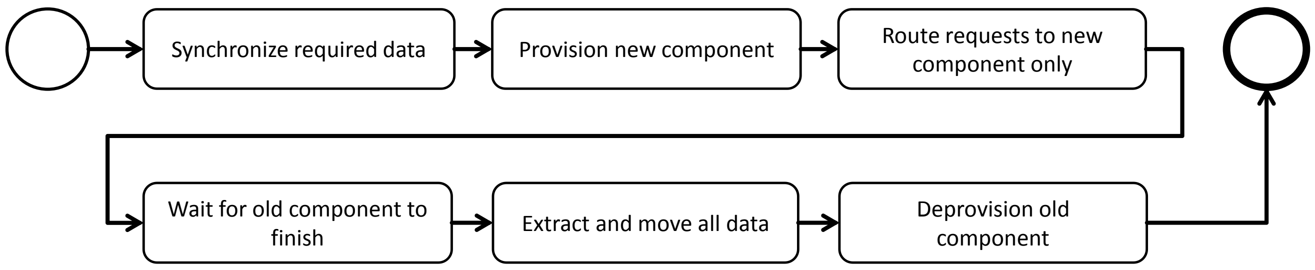
Driving Question: *How can an application component be moved to a different environment or updated to a new software, middleware, or hardware version, if downtime is inacceptable?*

Context: A *stateless application component*, which has no internal state, but relies on external data and is part of a *composite application*. The component must not be unavailable for the timeframe in which it is updated/moved.

Challenges: due to the provisioning time of the new component and the time it takes the old component to finish processing request, both components must be operated during the switch from one environment/version to the other. This is especially challenging, if the application component depends on external data that has to be shared by the old and the new components.

Solution: as depicted in Figure 14, replicate and synchronize required data to be accessed by the old and the new component and provision and operate both versions concurrently. Then, instantly switch from the old to the new version of the component.

Figure 14. Abstract Management Flow of the Move/Update without Downtime Pattern.


Synchronize required data
Provision new component
Route requests to new component only
Wait for old component to finish
Extract and move all data
Deprovision old component

Result: because the data accessed by both components is synchronized, a consistent component behavior is ensured. The switch between both versions is done instantly. The old component is given time to finish any processing that may have been assigned to it prior to the switch. Then, additional data that was not needed directly in the beginning, for example, log information or archived data, is moved. Finally, the old component is no longer needed and can be deprovisioned.

Variations: if the managed component does not access external data, the corresponding activities in the management flow to replicate and synchronize the data accessed by the managed application component can be omitted.

Known uses: Significant efforts are being made to migrate virtual machines between multiple runtime environments without downtime [44,45]. [46] gives an overview of the tasks that have to be covered and their order to achieve this.

5. Cloud Patterns and Implementation Artifacts

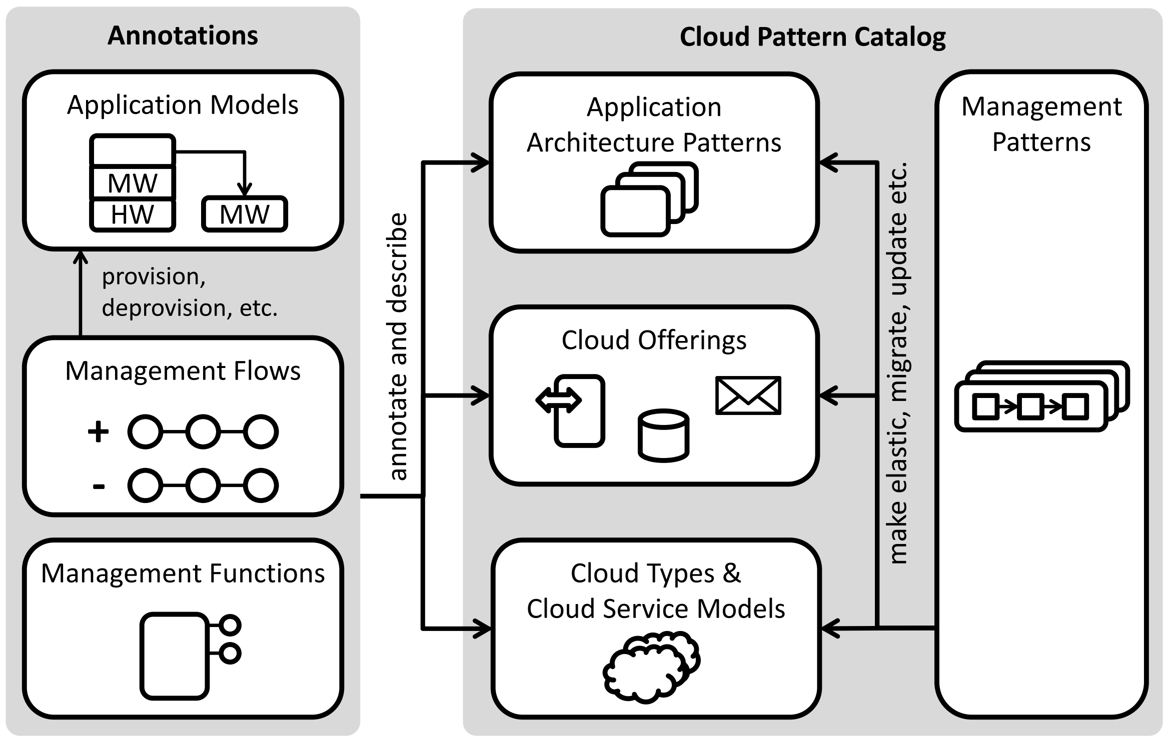
Patterns describe good solutions to a developer, but often they have to be implemented individually for each cloud environment used in an application. In [18], we covered conceptually how a list of implemented application components, concrete products, and their configuration options can be annotated to patterns in order to assist and standardize these efforts. These annotations can be used to ensure a homogenization of the runtime environments. This reduces the management effort, which is one of the major cost factors of IT systems, significantly [47].

In a similar form as architectural patterns are refined to a set of implemented software artifacts that are hosted on a standardized and restricted infrastructure, we now propose that the abstract management flows shall be refined for implemented application components. In Section 4, we gave examples how to abstractly make an application component elastic, how to make it resilient regarding system failures, and how to move it to a different environment or update it. In this section, we show how these abstract management flows may be refined in a structured fashion through the annotation of application component implementations, middleware, hardware, and descriptions of management interfaces. We argue that this annotation will lead to a similar standardization of application management efforts as other patterns have introduced to cloud applications. Figure 15 depicts the structure of the presented cloud pattern catalog together with proposed annotations. The new pattern class of management patterns covered in Section 4 abstractly describes how implementations of the other pattern classes can be managed. In the following, we first describe the annotated artifacts in greater detail and then give an end-to-end example how they can guide the implementation of management patterns. We described the implementation of other patterns classes is in greater detail in [18].

Implementation artifacts subsume application models, management functions, and management flows. Depending on the pattern class to which these artifacts are annotated, they assist the pattern instantiation to application components, cloud offering subscriptions, or management flows. Application models have been introduced by [48,49]. They describe the deployment dependencies among application components and used middleware. Here, we will use them to describe the dependencies among application components and cloud offerings provided by certain clouds. For example, an application component may be implemented as a Business Process Execution Language (BPEL) [50]

Process. This BPEL Process is then an entity in the application model that has a deployment dependency on a BPEL engine component. This component may then again specify a dependency on a virtual machine with an installation of Linux. Components in this model which are *provider supplied* have no dependency on other components but are provided, for example, in the form of a cloud offering. [48,49] consider each of these components to have a standardized *component interface* offering functionality to instantiate components and, in case of middleware components, to deploy other components on them. This standardized interface is used to automate the provisioning of applications modeled in the application diagram. Here, we generalize this approach and allow custom management interfaces for annotated to application components. For example, a cloud provider may offer a management interface to add users, create security credentials *etc*. Management flows may then orchestrate said management functionality to automate management tasks, *i.e.*, those described in Section 2.2.

Figure 15. Organization of Cloud Patterns in the Catalog and Annotated Implementation Artifacts.

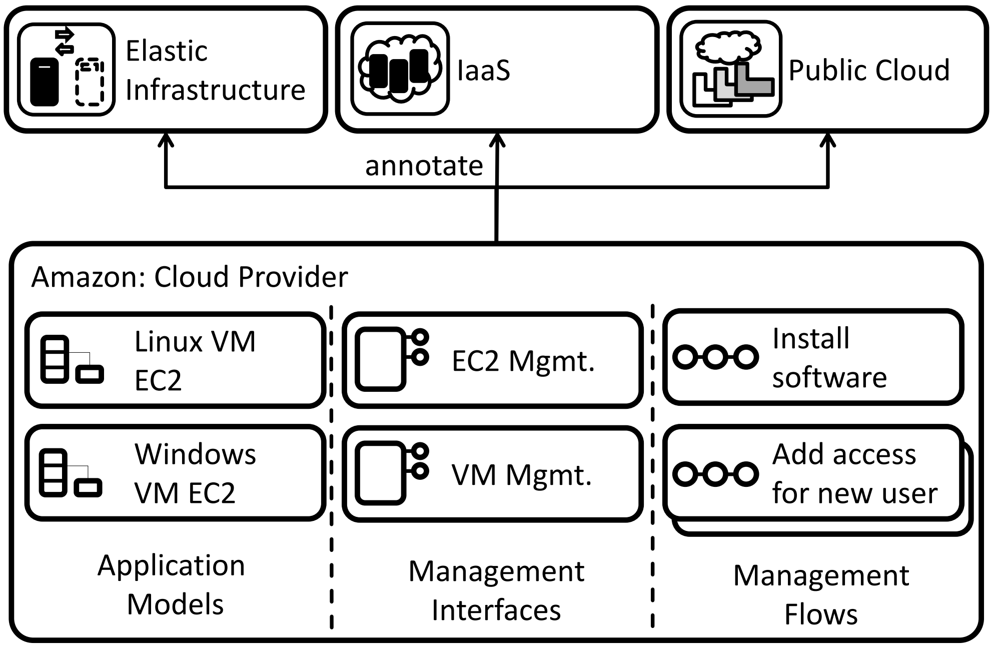


We now give examples for annotations to a cloud type, a cloud offering, and an application component as well as show how they can be combined during the application development process.

5.1. Exemplary Annotations of a Cloud Provider

In the exemplary annotation depicted in Figure 16, the users of the catalog wants to use Amazon EC2 as a cloud provider for the hosting of pre-configured Linux and Windows virtual machines. Further, the management functionality of these virtual machines to install software packages is made available via Web services. The provider is, therefore, described by two application models, one for the Linux and one for the Windows virtual machine. As management interfaces, the Web service interface of Amazon EC2 may be used to start and stop the virtual machines. Further, a VM management interface is provided that is offered by a Web service running on the custom virtual machine images. It offers functionality to install and uninstall software packages on instances of the virtual machines, start and stop a service, upload configuration files, *etc*. The functionality of these management interfaces is used in annotated management flows to install software via the VM management interface or add access for a new user to the virtual machine.

Figure 16. Implementations Artifacts of Amazon EC2.



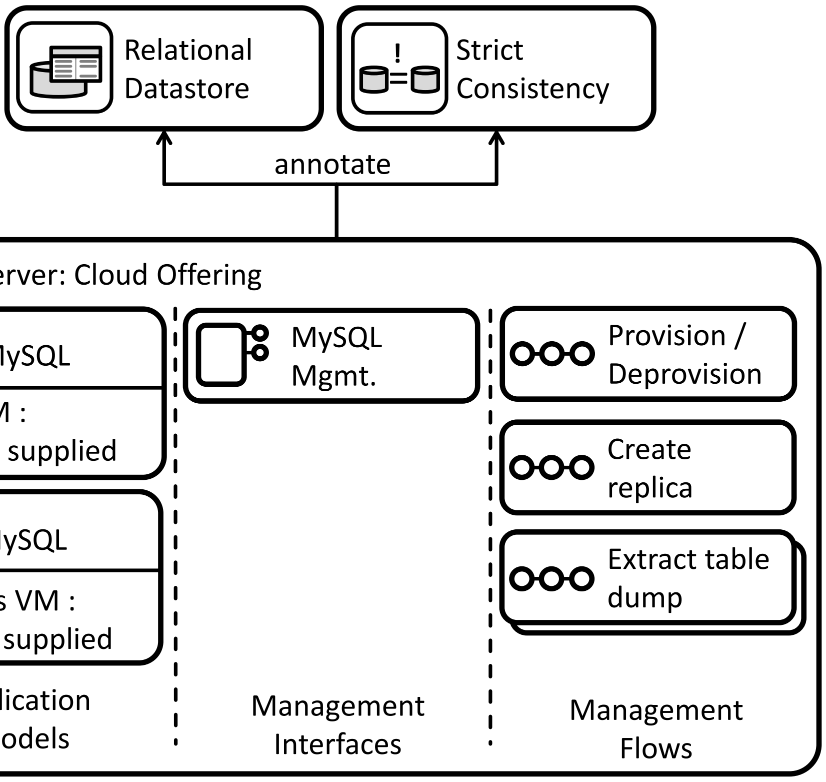
The set of implementation artifacts associated with Amazon EC2 are annotated to the patterns provided by the EC2 cloud offering. These are: (i) the elastic infrastructure pattern, because virtual machines may be added flexibly; (ii) the Infrastructure as a Service pattern, because it is the cloud service model that Amazon follows; and (iii) the public cloud pattern as services offered by Amazon are available to everyone. Whenever a developer selects one of these annotated patterns from the catalog, these annotations are, thus, recommended to him.

5.2. Exemplary Annotations of a Cloud Offering

A cloud offering for a MySQL server shall be described by annotated implementation artifacts as depicted in Figure 17. Again, multiple application models are available. In difference to the cloud provider, these application models cannot be instantiated independently but include a dependency on either a Linux or a Windows virtual machine. The MySQL offering therefore requires a virtual machine of either type providing a management interface to deploy the MySQL installation. To automate this functionality, it offers a management flow for its provisioning/deprovisioning that accesses the management interface of a running virtual machine in order to install/uninstall the MySQL server. The management functionality of MySQL itself is again encapsulated behind a management interface. This interface is further used in annotated management flows to create a replica of the MySQL server, extract a table dump, *etc*.

Similar to the cloud provider, the implementation artifacts describing this cloud offering are annotated to the patterns they implement. These are (i) the relational data store pattern, which describes how the MySQL offering behaves during the access of the database; and (ii) the strict consistency pattern, which describes the data consistency model followed by the MySQL offering.

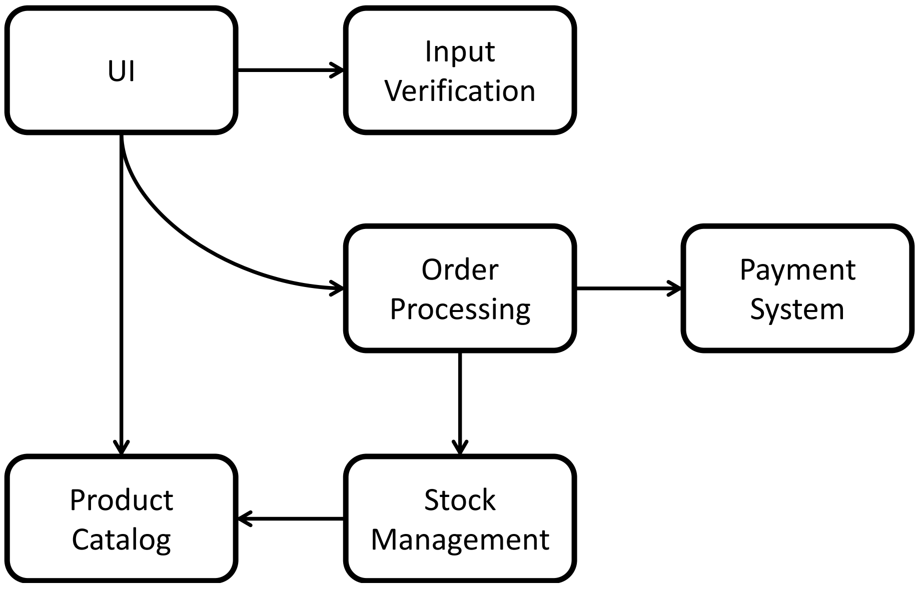
Figure 17. Implementation Artifacts of a MySQL Cloud Offering.



6. End-to-End Example for Pattern-Based Application Development and Management.

Consider the architecture diagram of a componentized application depicted in Figure 18. It is created by an application architect and shall now be passed to an application developer to undergo the *cloud application offering process* described in Section 2.3. The used notation is similar to that of UML Component Diagrams [51], thus, each box depicts a component and arrows show which other components are used by it. The application realizes a Web shop offering arbitrary products. Customers access the application through a web-based user interface to browse products in a product catalog. This catalog is periodically updated from a stock management component to respect item availability. Once customers decide to order an item, they are asked to input their contact and payment information. Input is first verified for consistency *i.e.*, to check if the zip code fits the street and city. Then, the ordering information is passed to the order processing component handling the billing and shipping process. For these tasks, the component accesses functionality of a payment system and a stock management system.

Figure 18. Components of Web Shop Applications and their Dependencies.

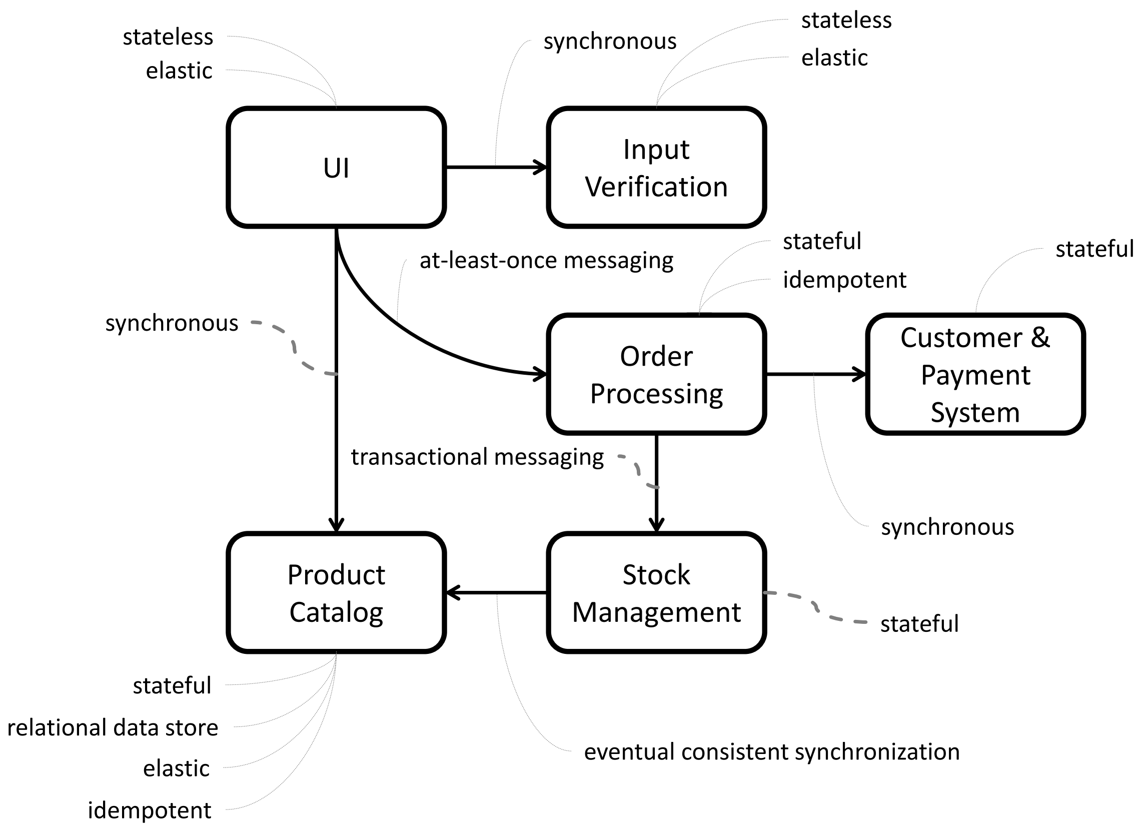


In the following, we show how the requirements of application components (box) and usage dependencies (arrow) on each other and on the runtime environment can be expressed through enriching the architecture diagram by pattern annotations. Based on these annotations, we discuss how application developers concretize the application components during the *development phase* by creating application models for each component. Finally, the annotated information is used to identify suitable management patterns during the *management planning phase* that are concretized by application managers using annotated implementation artifacts. In this example, we assume that the application is only developed and deployed once. Therefore, the optional *application customization phase* of the *cloud application offering process* is omitted. We show how cloud patterns may be used for expressing the requirements of the example application and the desired behavior and management of implemented application components. The architecture diagram is enriched using these patterns to guide the communication between application architects, application developers and application managers.

6.1. Requirements Management Through Pattern-Based Diagram Enrichment

To express the behavior expected from application components and communication channels, pattern names are annotated to entities in the architecture diagram as depicted in Figure 19. This enables application developers to identify suitable cloud offerings and middleware components that the implemented application components may use as runtime. Application components holding an internal state are characterized as implementing the *stateful component pattern*, while those relying completely on external states are annotated with the *stateless component pattern*. Further, the order processing component is required to be *idempotent*, which means it has to be able to handle duplicate messages as it is accessed via *at-least-once-messaging*. To address changing workload on the application, the application components that are needed for synchronous user interaction, UI, input verification, and product catalog, are required to be *elastic*. Once the order is issued to the order processing component timely responsiveness of the application is less critical with respect to the time it requires to ship the ordered products. Finally, the product catalog reflects the items handled by the stock management component and provides it to the user interface. No real-time information is required, thus, the availability of products does not have to be reflected directly in the product catalog. The synchronization between the stock management and the product catalog is, therefore, characterized as *eventual consistent*.

Figure 19. Enriched Diagram of the Web Shop Application.



6.2. Concretization of Architecture Diagrams During the Development Phase

Based on the annotated pattern information, an application developer may identify potential cloud offerings and cloud providers to be used during the implementation of the application components. The pattern-based expression of requirements and desired component behavior thus enables the application architect and the application developer to use a common language and diagram format. Lengthy textual descriptions with unclear semantics are reduced. We argue that this will lead to fewer communication errors regarding requirements specification. Here, we discuss this selection of implementation artifacts and the resulting concretization of the application architecture diagram for the order processing and the product catalog component.

To implement the process ordering component, the developer decides to manage the state of individual orders in a BPEL process ensuring the implementation of the *stateful component pattern*. The resulting application model is depicted in the right of Figure 20. To address duplicate messages, he decides to implement a message filter as a Web service as suggested by the *idempotent component pattern*. This message filter accepts messages send to the process ordering components and forwards it to the BPEL process. For these implementation components the application developer then requires to find an appropriate runtime environment. He identifies an application model with the Apache ODE BPEL process engine [52], and the Apache Tomcat application server [53], because it is annotated to both, the idempotent component and the stateful component patterns, as possible runtime. The application model then expects the availability of a Linux virtual machine. Since the order component handles business critical data, the developer decides to host it in a *private cloud*. Annotated with this cloud type, he finds an application model providing a Linux virtual machine in a private cloud based on VMware [54]. He combines this application model with the application components and the cloud offering application model to complete the application stack.

Figure 20. Concretization of the Order Processing Component.

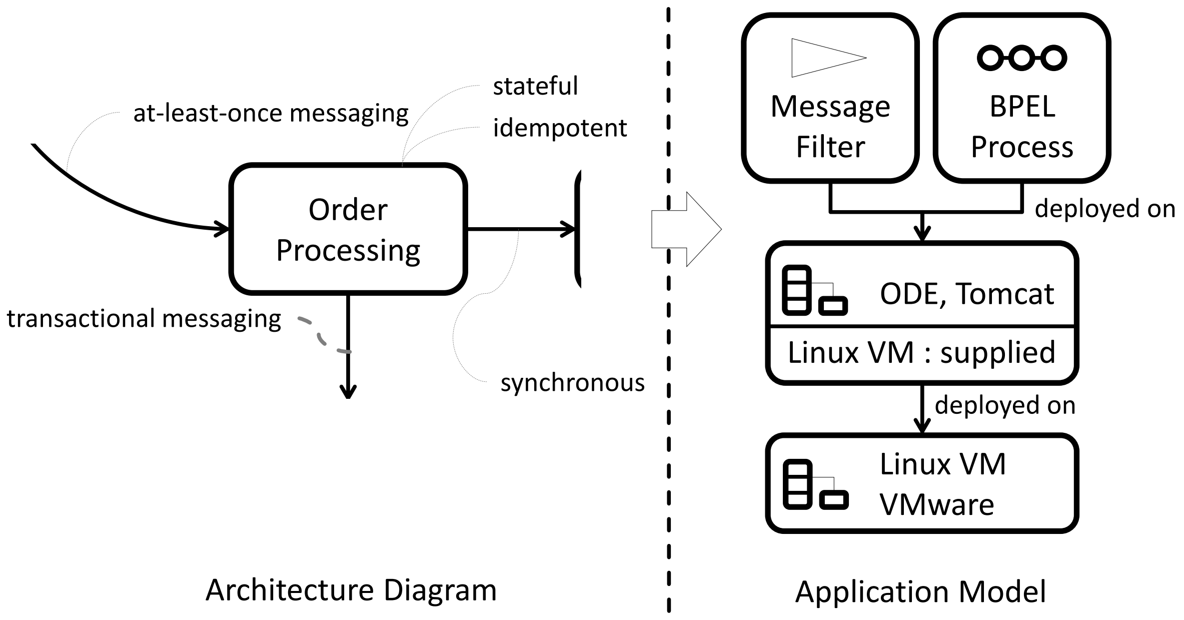
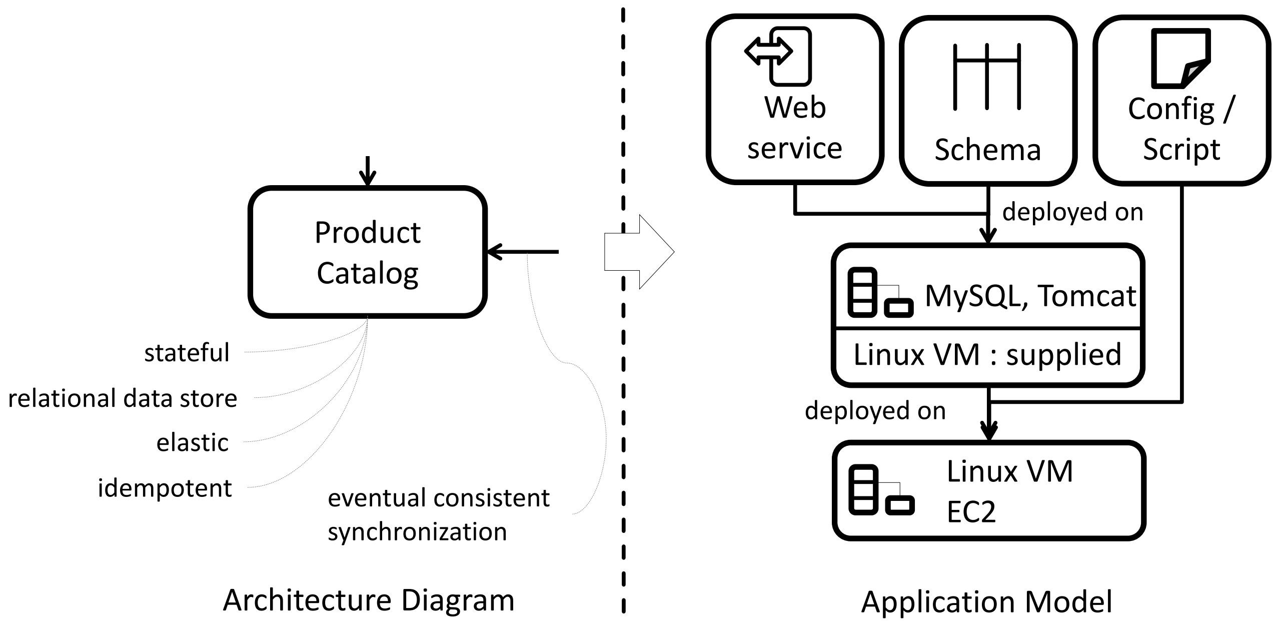


Figure 21 depicts the concretization of the product catalog component from the architecture diagram to the application model. To implement this component, the developer decides to realize it as Java Web service [55] that accesses a MySQL database [56]. He finds an application model for a MySQL server and an Apache Tomcat application server, a runtime for Web services, annotated to the *relational data store pattern* and the *stateful component pattern.* He decides to use this application model as runtime for the Web service and data base schema. As the product catalog component has to be elastic and hosts data that is publicly available, the developer further identifies a *public cloud* as suitable hosting environment and finds an annotated application model for hosting the required Linux virtual machine on Amazon EC2. To implement the desired eventual consistent synchronization with the stock management component, the developer creates a shell script and configuration for the Linux virtual machine. This script is executed after certain time intervals as specified in the configuration file. It accesses the local MySQL server and issues a poll of catalog data from the stock management component. Because this synchronization is only performed after certain intervals, the desired eventual consistent synchronization is guaranteed.

Figure 21. Concretization of the Product Catalog Component.



6.3. Pattern-Based Modeling of Management Flows During the Management Planning Phase

Through the interrelation of abstract management flows with pattern-based descriptions of application architecture patterns, cloud types, cloud service models, and cloud offerings, application mangers can identify management patterns that are applicable to the created application models. Now, we briefly discuss the provisioning of application components, because extensive work exists on this subject. Further, we give detailed examples how the management patterns for component migrations described in Section 4, can be concretized for the order processing component and the product catalog component. Again, this task is performed based on the enriched application diagram and the annotated implementation artifacts.

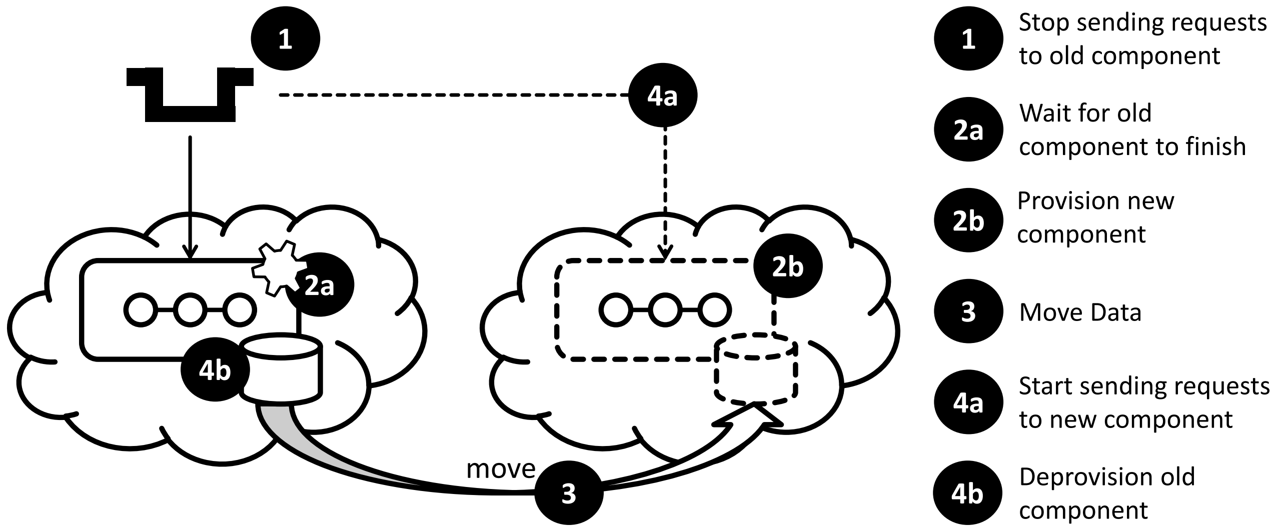
Provisioning of application components

Application components are provisioned through execution of provisioning management flows that are associated with them. The order in which individual provisioning flows must be executed can be obtained from dependencies between the application components. In [57], a method is described how this order may be computed automatically. Alternatively, complete provisioning flows may be generated from the application model if an additional variability model [49] describing component interdependencies in greater detail is also provided.

Moving the Order Processing Component with Downtime

Figure 22 depicts the deployment of the ordering processing component that is accessed through a messaging queue on the very left. The queue provides asynchronous access to the order processing component. Its application model has been identified for use in this setup through an annotation to the *at-least-once messaging pattern*. We now cover the modeling of an automated management flow to move this component to a different cloud environment depicted in the center of the figure. On the right, the steps of the abstract management flow handling the move of application components with downtime are listed. The complete flow can be obtained from Figure 12 part of Section 4.3. The first activity in the abstract management flow is firstly to stop sending requests to the old process management component. To perform this activity in the management flow, the application manager refines this first activity to access the management interface of the messaging queue and to stop the delivery of messages. Since no more messages are processed now, the process ordering component is unavailable to the rest of the application. The second activity of the management flow (2a) waits for the running process ordering component to finish handling requests that were already assigned to it. The application manager refines this activity to periodically poll the management interface of the process engine to check the number of running process instances. Simultaneously, (2b) the new component is provisioned. This activity is handled by refining it to execute the corresponding provisioning flows for the used BPEL engine and then accessing its management interface to install the process model. In a similar fashion, the Web service handling the message filtering can be automatically setup in the new environment. When there are no more active ordering process instances running on the old process engine, the (3) process data, *i.e.*, information about completed instances is moved to the new ordering process component. The application manager enables this by refining this abstract activity to access the management functionality offered of the process engine in order to extract the data about the completed instances. Then, the functionality of the same interface at the new process engine is accessed to store the extracted data. When the new order processing component is ready to handle requests and all data has been moved, the next management flow activity (4a) ensures that requests are started to be send to the new component. This is enabled by accessing the management interface of the queue and starting message delivery again. At the same time, the deprovisioning management flow of the old order processing component is called to implement the abstract activity (4b) handling the deprovisioning of the old component.

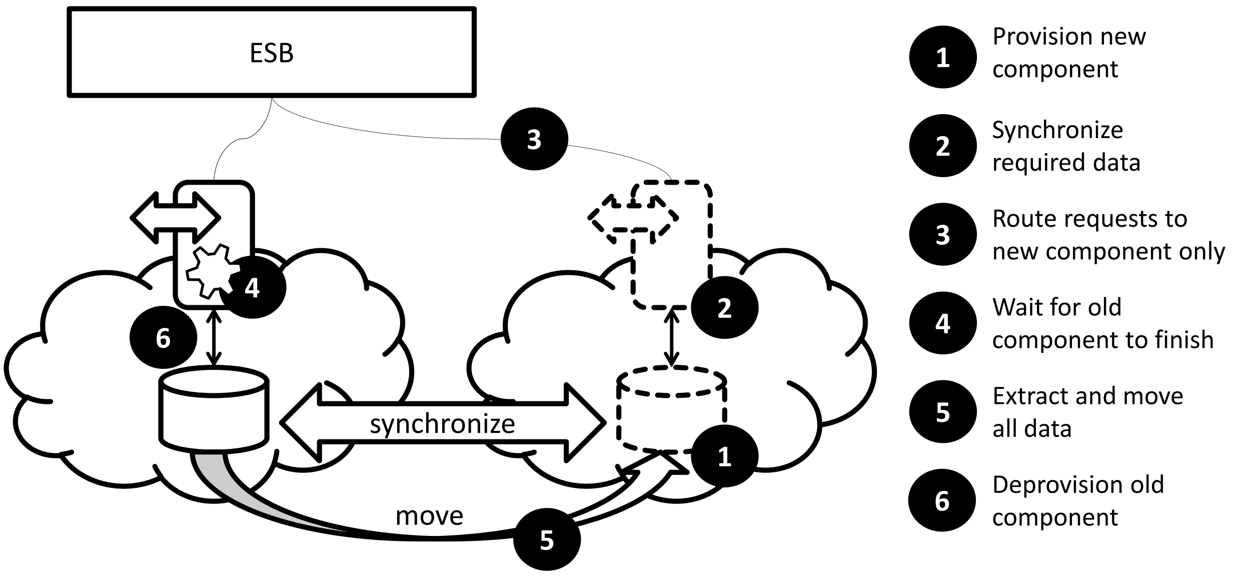
Figure 22. Steps to Move the Order Processing Component.


1
4a
2a
2b
4b
move
3
1 Stop sending requests to old component
2a Wait for old component to finish
2b Provision new component
3 Move Data
4a Start sending requests to new component
4b Deprovision old component

Moving the Product Catalog Component Without Downtime

The move of the product catalog component implemented as the Web service depicted in the left of Figure 17 shall also be automated using a management flow. The component is accessed synchronously through an enterprise service bus (ESB) [58]. Because the product catalog is required by the user interface for application users, its migration management flow shall ensure that there is no downtime. The steps of the abstract management flow introduced in Figure 14 of Section 4.4 shown on the very right of Figure 23 must therefore be concretized by the application manager. First, (1) the product catalog component is provisioned in the new environment by using its provisioning flow. Then, the application manager has to specify how (2) the required data shall be synchronized. He does so by accessing the management interface of the database residing in the old cloud environment to obtain a table dump and also models its replay to the new database component using its management interface. Because the data synchronization between the stock management component and the product catalog is eventually consistent and is also handled by a script running on the new product catalog component, this initial data replication is sufficient. The developer now refines the next abstract activity (3) to route requests to the new component by changing the configuration of the ESB using its management interface. The management flow then (4) has to wait for the old component to stop processing requests. However, the product catalog does not offer a management interface to obtain this information. The application manager, thus, refines this abstract activity to use management functionality of the cloud provider to check CPU utilization of the underlying virtual machine periodically. The following step (5) to extract and move all data can be omitted in this example, because during the remaining processing of the old component data is not created or altered. It can therefore be (6) deprovisioned by including its corresponding management flow.

Figure 23. Steps to Move the Product Catalog Component.


ESB
3
4
6
2
synchronize
1
move
5
1 Provision new component
2 Synchronize required data
3 Route requests to new component only
4 Wait for old component to finish
5 Extract and move all data
6 Deprovision old component

6.4. Triggering of Automated Management Flows

In [59], we present a framework handling the event-based execution of provisioning and deprovisioning flows annotated to application components. Based on application model events can be specified, which are generated by each component or by the user of the application. Additionally, for each application component it can be specified if it should be provisioned or deprovisioned if a specific event was observed. For example, a queue can throw an event when a certain threshold of messages stored in it is exceeded. The necessary provisioning and deprovisioning flows are then triggered by the framework through iteration of the application components in the application model. Those, whose provisioning or deprovisioning flows are triggered by this event are then added or removed from the application respectively. In scope of the queue example, the framework can, therefore, ensure that additional components are provisioned to handle messages in case the queue is filling up. The refined automated management flows, proposed by this paper, seamlessly integrate into this framework through their correlation to events generated by users and application components to specify under which conditions they shall be executed.

7. Summary and Outlook

We have seen how the pattern catalog may guide application developers during the creation of application components. The patterns implemented by individual application components also guided the application managers during the identification of management patterns that he could use to manage the application component. We have defined such abstract management flows in a pattern format to handle elasticity, resiliency, and for moving application components between different environments or updating them to a new version. Using the proposed pattern catalog and annotated implementation artifacts, we have further shown how these abstract management flows could be refined for different components of an exemplary web shop application. Future use of patterns in architecture diagrams will have to be further refined to handle a larger catalog of patterns and have to be integrated into architecture modeling tools. An extension of architecture modeling languages, such xADL [60], could be one solution. The general challenge that needs to be addressed is the creation of a composition language to connect the icons of architectural patterns in a structured and well-defined manner. To extend tool support, patterns should further be associated with properties of the application that they enable, for example, high-availability, privacy, *etc*. With an increasing size of the catalog, such annotations provide an easier accessibility of the patterns.

A limitation of the presented approach is that the management flows currently consider isolated application components. While such isolated management of application components can be helpful during the runtime management of the application, considering the complete application would be even more powerful. For example, based on the architectural diagram of the example Web shop application seen in Figure 18 of Section 6 management tasks may be automated for the overall application as well. Conceptually, this would result in a management flow that executes the individual management flows of application components in a particular order. This order can likely be determined automatically by a graph-analysis of the dependencies expressed in the application architectural diagram. Therefore, future research will investigate how the dependencies between application components can be evaluated automatically to compose individual management flows of application components into management flows handling the complete application. In [57] such computations have already been investigated based on similar architecture diagrams and descriptions of variability. However, these models only consider deployment dependencies of application components on middleware and hardware to compute the order of provisioning tasks. To incorporate the links of architecture diagrams, these models and corresponding algorithms need to be extended respectively.

Another limitation is that individual components are refined to use separate middleware and hardware stacks, thus, each application component is individually combined with cloud offerings by the application developer. However, in practice, application components often share middleware and hardware due to performance and licensing issues. In [61], we proposed a method, how application architecture diagrams can be used to optimize the assignment of application components to different middleware installations, virtual machines, and clouds. The above mentioned management flows considering the complete application, therefore, have to respect the impact of shared middleware, hardware, and cloud offerings as well. The challenge to merge redundant middleware and hardware also arises if outsourced IT resources are moved back into a company or two companies merge. Therefore, approaches targeting these challenges may be applicable in the domain of pattern-based development and management of cloud applications. Recently, [62] started to capture these concepts.

Acknowledgments

This work was supported by the German Research Foundation (DFG) within the funding programme Open Access Publishing.

References

1. Fehling, C.; Leymann, F.; Mietzner, R.; Schupeck, W. A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-based Application Architectures. Technical Report No. 2011/05; University of Stuttgart: Stuttgart, Germany, 2011.
2. Varia, J. Architecting for the Cloud: Best Practices. Technical Report, Amazon, 2010. Available online: http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf (accessed on 15 January 2012).
3. DaimlerChrysler TSS GmbH: MDA Success Story ePEP successful with Model Driven Architecture, 2005. Available online: http://www.omg.org/mda/mda_files/ SuccesStory_DC_TSS_MDO_English.pdf (accessed on 15 January 2012).
4. Malone, T.; Blokdijk, G.; Wedemeyer, M. *ITIL V3 Foundation Complete Certification Kit*; Emereo Pty Ltd.: Brisbane, Australia, 2008.
5. Brown, A.B.; Patterson D.A. To Err is Human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependability (EASY'01)*, Göteborg, Sweden, July 2001.
6. Kuhn, D.R. Sources of failure in the public switched telephone network. *Computer* **1997**, *6*, 31–36.
7. Amazon. AWS Management Console. Available online: http://aws.amazon.com/console/ (accessed on 15 January 2012).
8. Microsoft. The New Management Portal. Available online: http://msdn.microsoft.com/ en-us/library/gg441576.aspx (accessed on 15 January 2012).
9. Mitchell, R. Managing virtual machines. Computerworld, 2006. Available online: http://features.techworld.com/operating-systems/2569/managing-virtual-machines/ (accessed on 15 January 2012).

10. Lagar-Cavilla, H.A.; Whitney, J.A.; Scannell, A.M.; Patchin, P.; Rumble, S.M.; De Lara, E.; Brudno, M.; Satyanarayanan, M. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, Nuremberg, Germany, April 2009.
11. Amazon. CloudWatch. Available online: http://aws.amazon.com/cloudwatch/ (accessed on 15 January 2012).
12. Microsoft. Windows Azure. Available online: http://www.microsoft.com/windowsazure/ (accessed on 15 January 2012).
13. Distributed Management Taskforce (DMTF): Interoperable Clouds Whitepaper, 2011.
14. IEEE. Intercloud Working Group (ICWG), 2011. Available online: http://standards.ieee.org/develop/wg/ICWG-2302_WG.html (accessed on 15 January 2012).
15. IEEE. Cloud Profiles Working Group (CPWG), 2011. Available online: http://standards.ieee.org/develop/wg/CPWG-2301_WG.html (accessed on 15 January 2012).
16. Storage Networking Industry Association (SNIA): Cloud Data Management Interface (CDMI) Whitepaper, 2010.
17. Fehling, C.; Konrad, R.; Leymann, F.; Mietzner, R.; Pauly, M.; Schumm, D. Flexible Process-based Applications in Hybrid Clouds. In *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD)*, Washington, DC, USA, July 2011.
18. Fehling, C.; Leymann, F.; Retter, R.; Schumm, D.; Schupeck, W. An Architectural Pattern Language of Cloud-based Applications. In *Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP 2011)*, 21–23 October 2011.
19. Hohpe G.; Wolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying*; Addison-Wesley: Reading, MA, USA, 2004.
20. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*; Addison-Wesley: Reading, MA, USA, 1995.
21. Petre M.: Why Looking isn't Always Seeing. *Commun ACM* **1995**, *38*, doi:10.1145/203241.203251.
22. Schumacher, M.; Fernandez-Buglioni, E. Security Patterns: Integrating Security and Systems Engineering; John Wiley & Sons: Hoboken, NJ, USA, 2005.
23. Amazon. Amazon Web Services. Available online: http://aws.amazon.com/ (accessed on 15 January 2012).
24. Somorovsky, J.; Heiderich, M.; Jensen, M.; Schwenk, J.; Gruschka, N.; Lo Iacono, L. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCSW)*, Chicago, IL, USA, 17–21 October 2011.
25. Hashizume, K.; Yoshioka, N.; Fernandez, E.B. Misuse Patterns for Cloud Computing. In *Proceedings of the Asian Conference on Pattern Languages of Programs (AsianPLoP)*, Tokyo, Japan, 17–19 March 2011.
26. Erl, T. *SOA Design Patterns*; Prentice Hall: Upper Saddle River, NJ, USA, 2009.
27. van Der Aalst, W.M.; Ter Hofstede, A.H.; Kiepuszewski, B.; Barros, A.P. *Workflow Patterns, Distributed and Parallel Databases*; Springer: Berlin, Germany, 2003.
28. Date, C.J. *An Introduction to Database Systems*; Addison-Wesley: Reading, MA, USA, 2000.

29. Erl, T. *SOA Principles of Service Design*; Prentice Hall: Upper Saddle River, NJ, USA, 2007.
30. Dean J.; Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. Google Whitepaper, 2004. Available online: http://labs.google.com/papers/mapreduce.html (accessed on 15 January 2012).
31. Leymann, F.; Roller, D. Workload Balancing in Clustered Application Servers. U.S. Patent *6681251 B1*, 20 January 2004.
32. Chong, F.; Carraro, G. Architecture Strategies for Catching the Long Tail. Microsoft Whitepaper, 2006. Available online: http://msdn.microsoft.com/en-us/library/aa479069.aspx (accessed on 10 February 2012).
33. Object Management Group (OMG). BPMN 2.0 Specification Document, 2011. Available online: http://www.omg.org/spec/BPMN/2.0/PDF/ (accessed on 15 January 2012).
34. Allspaw, J. *The Art of Capacity Planning*; O’Reilly: Sebastopol, CA, USA, 2008.
35. Hill, Z.; Li, J.; Mao, M.; Ruiz-Alvarez, A.; Humphrey, M. Early Observations on the Performance of Windows Azure. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, Chicago, IL, USA, 21–25 June 2010.
36. Paraleap Technologies. Azurewatch: Elasticity-as-a-Service for Windows Azure, 2011. Available online: http://www.paraleap.com/ (accessed on 15 January 2012).
37. Tanenbaum, A.S.; van Steen, M. *Distributed Systems: Principles and Paradigms*; Prentice Hall: Upper Saddle River, NJ, USA, 2007.
38. Leymann, F.; Roller, D. *Production Workflow: Concepts and Techniques*; Prentice Hall: Upper Saddle River, NJ, USA, 2000.
39. Amazon. *Elastic Beanstalk Developer Guide*; Amazon Web Service: Seattle, WA, USA, 2010.
40. Peecho. Print as a Service, 2011. Available online: http://peecho.com (accessed on 15 January 2012).
41. Peecho. Minimizing downtime on Amazon AWS, 2011. Available online: http://www.peecho.com/blog/minimizing-downtime-on-amazon-aws.html (accessed on 15 January 2012).
42. Kununu. Job rating site, 2011. Available online: http://kununu.com (accessed on 15 January 2012).
43. Amazon. AWS Case Study: kununu.com, 2011. Availableonline: http://aws.amazon.com/ solutions/case-studies/kununu/ (accessed on 15 January 2012).
44. Red Hat. Enterprise Virtualization: Live Migration. Available online: http://www.redhat.com/f/pdf/rhev/DOC054-RHEV-Live-Migration.pdf (access on 15 February 2012).
45. VMware: vMotion, 2011. Available online: http://www.vmware.com/products/vmotion/ (accessed on 15 January 2012).
46. Clark, C.; Fraser, K.; Hand, S.; Hansen, J.G.; Jul, E.; Limpach, C.; Pratt, I.; Warfield, A. Live Migration of Virtual Machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, Berkeley, CA, USA, 02–04 May 2005.
47. Riempp, G.; Gieffers-Ankel, S. Application Portfolio Management: A Decision-Oriented View of Enterprise Architecture. *Information Systems and e-Business Management* 2007, *5*, 359–378.
48. Mietzner, R.; Leymann, F. A Self-Service Portal for Service-Based Applications. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 13–15 December, Perth, Australia, 2010.

49. Mietzner, R.; Unger, T.; Leymann, F. Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE,* Crete, Greece, 17–21 October 2009.
50. OASIS: Web Services Business Process Execution Language Version 2.0, 2007. Available Online: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (accessed on 15 January 2012).
51. Object Management Group (OMG). Unified Modeling Language (UML), 2010. Available online: http://www.omg.org/spec/UML/2.3 (accessed on 15 January 2012).
52. Apache Software Foundation: Apache ODE, 2011. Available online: http://ode.apache.org/ (accessed on 15 January 2012).
53. Apache Software Foundation: Apache Tomcat, 2011. Available online: http://tomcat.apache.org/ (accessed on 15 January 2012).
54. VMware: vCenter, 2011. Available online: http://www.vmware.com/de/products/ datacenter-virtualization/vcenter (accessed on 15 January 2012).
55. Oracle. Java Web Services Overview, 2011. Available online: http://www.oracle.com/ technetwork/java/index-jsp-137004.html (accessed on 15 January 2012).
56. Oracle: MySQL. Available online: http://www.mysql.com (accessed on 15 January 2012).
57. Mietzner, R.; Leymann, F. Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors, In *Proceedings of the IEEE International Conference on Services Computing (SCC),* Hawaii, HI, USA, 8–11 July 2008.
58. Chappell, D.A. *Enterprise Service Bus*; O’Reilly: Sebastopol, CA, USA, 2004.
59. Fehling, C.; Retter, R. Composite as a Service: Cloud Application Structures, Provisioning, and Management. *IT Inf. Technol.* **2011**, *53*, 188–194.
60. University of California. Highly-extendable Architecture Description Language for Software and Systems, 2003. Available online: http://www.isr.uci.edu/projects/xarchuci/ (accessed on 15 January 2012).
61. Leymann, F.; Fehling, C.; Mietzner, R.; Nowak, A.; Dustdar, S. Moving Applications to the Cloud: An Approach Based on Application Model Enrichment. *Int. J. Coop. Inf. Syst.* **2011**, *20*, 307–356.
62. Binz, T.; Leymann, F.; Schumm, D. CMotion: A Framework for Migration of Applications into and between Clouds, In *Proceedings of IEEE International Conference on Service Oriented Computing & Applications (SOCA),* Irvine, CA, USA, 12–14 December 2011.