Architectural Principles for Cloud Software

CLAUS PAHL, Software and Systems Engineering, Free University of Bozen-Bolzano, Italy
POOYAN JAMSHIDI, Department of Computer Science, Carnegie Mellon University, USA
OLAF ZIMMERMANN, Institute for Software, Hochschule für Technik Rapperswil, Switzerland

A cloud is a distributed Internet-based software system providing resources as tiered services. Through service-orientation and virtualization for resource provisioning, cloud applications can be deployed and managed dynamically. We discuss the building blocks of an architectural style for cloud-based software systems. We capture style-defining architectural principles and patterns for control-theoretic, model-based architectures for cloud software. While service orientation is agreed on in the form of service-oriented architecture and microservices, challenges resulting from multi-tiered, distributed and heterogeneous cloud architectures cause uncertainty that has not been sufficiently addressed. We define principles and patterns needed for effective development and operation of adaptive cloud-native systems.

CCS Concepts: • **Software and its engineering → Software infrastructure**; **Software architectures**; **Cloud computing**; *Development frameworks and environments*;

Additional Key Words and Phrases: Cloud computing, architectural style, control theory, adaptive system, software architecture, microservice, devops, model-based controller, uncertainty, cloud-native.

ACM Reference format:
Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. 2018. Architectural Principles for Cloud Software. *ACM Trans. Internet Technol.* 18, 2, Article 17 (February 2018), 23 pages.
https://doi.org/10.1145/3104028

1 INTRODUCTION

The cloud is a distributed architecture of individual cloud-native services, providing resources as services in a tiered fashion to construct a full technology stack from hardware to middleware platforms to applications. The configuration and deployment of applications and cloud platforms as interdependent adaptive systems can be managed dynamically, responding to changes in both requirements and the execution platform.

We argue that a particular software architectural style for cloud is needed to address continuous service systems development and operation. We propose a set of key principles and patterns for control-theoretic, model-based architectures. We combine established principles with emerging concerns. While aspects such as uncertainty have to some extent been investigated (Farokhi et al. 2015; Esfahani and Malek 2013; Jamshidi et al. 2014) in terms of cloud operations management, wider development challenges resulting from the cloud architecture as a multi-tiered, distributed

Authors' addresses: C. Pahl, Research Centre for Software and Systems Engineering, Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy; P. Jamshidi, Department of Computer Science, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, USA; O. Zimmermann, Institute for Software, Hochschule für Technik Rapperswil, Oberseestrasse 10, 8640 Rapperswil, Switzerland.

https://doi.org/10.1145/3104028

environment for increasingly fragmented application architectures, e.g., microservice architectures, have not been accounted for (Pahl and Jamshidi 2015). A differentiator for an architectural style for the cloud is the need to address a full provisioning stack from *resources* to *platform* to *application* and how these are operated and managed dynamically. We identify the challenges and define key elements for an architectural style with principles and patterns that are based on a control-theoretic, model-based cloud architecture in which domain-specific architectural concerns play the central role:

- *Adaptation.* Adapting systems to changing requirements is often necessary to guarantee correct and satisfying performance. Self-adaptive software systems can adjust their behaviour in response to their perception of the environment and the system itself (De Lemos et al. 2013). This has been approached as a requirements engineering problem, but the need for a software architecture perspective is recognised.
- *Dynamic Models and Control.* Requirements need to have a representation at runtime to allow self-adaptive systems to interact with the environment, i.e., "self-reflect" through models that link in the decision-making process necessary to change the underlying system (Baresi and Ghezzi 2013; Ghezzi et al. 2013; Pahl et al. 2017b). Dynamically adaptive systems deal with requirements as dynamically manageable models, enacted through a "self-prediction"-based controller that implements a control theory-based feedback loop (Filieri et al. 2015). Self-adaptiveness, self-reflectiveness, and self-predictiveness are the cornerstones of the operational side (Kounev 2011).
- *DevOps and Continuous Development.* Microservices are emerging as a variant of service-oriented architecture (SOA), aiming at realising software systems as sets of small services, each deployable on a different platform and running in its own process while communicating with each other through lightweight mechanisms without centralized control (Newman 2015; Lewis and Fowler 2014). Microservices move us towards continuous development and delivery (Fitzgerald and Stol 2014). An example is the integration of *Dev*elopment and *Op*erations that is called DevOps (Brunnert et al. 2015). Microservices require full application stacks; their infrastructure resources (e.g., data stores and networks) have to be managed adequately. DevOps provides this link from software development to technology operations and quality management; container technology and cloud-native services provide cloud-based virtualization and implementation support (Pautasso et al. 2017; Pahl et al. 2017a).

Adaptation, runtime models, continuous development, and deployment are among the principles and patterns for a cloud architectural style that have achieved a certain amount of consensus in industry and academia. We use concrete technology examples to illustrate specifically the emerging features in more detail that distinguish the cloud from other existing service-oriented contexts. The examples are for illustration only and should not be seen as part of a style definition itself or as forming a practical guide. Our target audience includes researchers (developing tools and frameworks) and architects in general (at provider and consumer side).

We address the uncertainty that impacts cloud-native software systems by making model-based adaptation and controller architectures core elements of the architectural style. This goes beyond cloud computing principles like virtualization or service-orientation, aiming to enable continuous development and deployment as key features.

In summary, our contributions are the following: We define a cloud architectural style with its principles (service-orientation, virtualization, adaptation, and uncertainty) and patterns (microservices, quality models at runtime, control loop, and controller architecture). We discuss real-world case studies to demonstrate benefits and implications for DevOps and Internetware.

We start with the review and definition of architectural styles, specifically looking at cloud concerns in Section 2. We also outline our proposed style based on principles and patterns, linking it to a general cloud architecture definition in the same section. In Section 4, we introduce the principles of the architectural style. The patterns of the style are subject of Section 5, which are detailed in individual subsections. We end with a discussion of the architectural style for cloud computing and its applicability and more related work.

2 INTRODUCTION TO THE PROPOSED CLOUD ARCHITECTURAL STYLE

An architectural style, sometimes called an architectural pattern, is a set of principles and coarse-grained patterns that provides an abstract framework for a family of systems (Microsoft 2009). An architectural style improves separation of concerns and promotes design reuse by providing solutions to frequently recurring problems (Ahmad et al. 2014). Architecture styles are sets of principles and patterns that shape an application. Garlan and Shaw define an architectural style as "a family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles) [...] or constraints related to execution semantics" (Garlan and Shaw 1994). To make principles and patterns more explicit, we follow (Zimmermann 2009): "an architectural style consists of a set of architectural principles and patterns that are aligned with each other to make designs recognizable and design activities repeatable: The principles express architectural design intent; the patterns adhere to the principles and are commonly occurring (proven) in practice." The benefits of architectural styles include a common language, allowing discussions of architecture aspects in a technology-agnostic way. This facilitates a conversation among architects, users, and developers based on patterns and principles to analyse and define architectures without the need for formal languages.

We start with an introduction to our proposed architectural style in Section 2.1 before putting this style into the context of service-oriented architecture, adaptive systems, continuous development, and DevOps in Section 2.2.

2.1 An Architectural Style for Cloud Software Systems

Cloud-native software architectures can be defined using principles and patterns of an architectural style. *Principles* capture the following influencing factors:

- service-orientation: cloud offerings are provided as services following layering, modularity, and loose coupling principles.
- virtualization: providing services for shared, virtualized resources (infrastructure virtualization) and portable application containers (platform virtualization).
- uncertainty: distribution, heterogeneity, and multi-user involvement in a changing context cause uncertainty.
- adaptivity: services including operations and management support allow for dynamic adaptation and variability management.

Patterns map principles to development and deployment platform solution templates:

- microservices: a flexible composition paradigm based on independent, self-manageable containerised components and cloud-native services.
- models at runtime: allows aspects of uncertainty to be addressed dynamically.
- controller-based feedback loop: a model-based loop (feedback, feedforward) allows controllers to adapt to and manage change.

The aim of these principles and patterns is to manage quality. The model notion in these adaptive systems plays a central role for continuous development, capturing structural and behavioural aspects of the architecture, and linking it with dynamic management of quality and variability. The architecture of a cloud system is a model of a dynamic system with a structure that takes the layers of a computing stack explicitly into account (Kounev et al. 2014). Formal languages such as discrete or continuous time models and probabilistic or fuzzy logic can help to capture the dynamic behaviour of cloud-based software systems. Models at runtime embedding this information form the core of a controller to adapt the system in uncertain environments.

Uncertainty is an important concept in this context. Uncertainty is used by many today, e.g., Esfahani and Malek (2013), to describe factors such as vagueness regarding the availability of resources, operating conditions that a system will encounter at runtime, and the emergence of new requirements while a system is operating. Control theory plays a more and more important role in software engineering to address these uncertain conditions, see, e.g., Filieri et al. (2015). Therefore, uncertainty is a concept that requires feedback loops (e.g., through controller architectures) to be implemented. Uncertainty arises here if heterogeneous cloud solutions are combined and managed across different layers. Different sources of uncertainty such as uncertainty in adaptation and its models, in the dynamic provisioning environment, in monitoring data, and in change enactment can be identified (Jamshidi et al. 2014) – detailed below.

The presented style is based on a survey of relevant literature. While we have not followed a systematic literature review protocol, we have reverse engineered commonly accepted and recently discussed concerns by looking at high-impact publications. We included documented industry use cases in the survey and validated the identified principles and patterns in concrete projects via implementation and technical action research (Wieringa and Moralı 2012). For instance, we will refer to our own experience, e.g., on auto-scaling with OpenStack and Azure, through control-theoretic solutions that we have implemented with companies such as Microsoft and Intel. Definitions such as the one from NIST (Mell and Grance 2011) identify service-orientation or shared resources for rapid provisioning, which can be enabled through virtualization. Workload and quality management are capabilities attributed to cloud systems (Fehling et al. 2014).

2.2 *Architectural Styles: SOA, Adaptive Systems, Continuous Development, DevOps*

Service-Oriented Architecture (SOA) and REpresentational State Transfer (REST) are architectural styles that are relevant to cloud software systems (Erl 2005; Perrey and Lycett 2003). A service is a logical, encapsulated, self-contained, and composable representation of a technical or business activity with specified outcome. Services as modular loosely coupled units of functionality form systems through composition mechanisms such as layering. Thus, we include SOA as a meta-principle with loose coupling, modularity, and layering as guiding principles (Zimmermann 2009). However, the lack of joint control in a shared environment requires us to also consider other principles.

Adaptive systems, managed by a control loop, are another architectural framework (De Lemos et al. 2013). The need for adaptation is recognised. Amazon states in its white paper on architecture for the cloud[1] that applications adapt themselves to fluctuating demand patterns by deploying resources instantaneously and automatically. Our definition of a cloud system entails that adaptation can happen at the infrastructure or platform side as well as in the application utilising these. Again, we have taken respective principles on board but combined them with uncertainty and SOA ones. Kratzke and Peinl (2016) propose the ClouNS reference as a six-layered stack of cloud-native resources that encompasses the traditional software-as-a-service (SaaS),

[1] J. Varia: Architecting for the Cloud: Best Practices. 2010.

platform-as-a-service (PaaS), and infrastructure-as-a-service (IaaS) layers. The addition of a container layer and a cluster management layer on top of that are important here. It aligns with the importance we assign to microservices as an architectural pattern suitable for a cloud-native architecture.

Another direction is the seamless integration of development and operations of software, known as continuous development (Fitzgerald and Stol 2014). While methodologies, such as agile development, encourage cross-functional collaboration among analysis, design, development, and quality assurance (QA) in traditional, functionally separated organisations, a cross-departmental integration of these functions with Information Technology (IT) operations and continuous delivery is lacking.

DevOps promotes processes and methods for communication and collaboration among development, QA, and IT operations. It emphasises communication, integration, automation, and cooperation between software developers and other IT professionals. DevOps sits at the intersection of development, quality assurance, and technology operations. DevOps adoption is being driven by cloud-related factors such as the wide availability of virtualized and cloud infrastructure from internal and external providers and the increased usage of data center automation and configuration management tools.

3 CLOUD ARCHITECTURE – DEFINITION AND SCENARIO

To put our architectural style definition on sound foundations, we properly introduce the notion of a cloud architecture in Section 3.1 and illustrate this through a High-Availability and Disaster Recovery (HADR) use case in Section 3.2.

3.1 Cloud Architecture Definition

The concept of cloud software architecture needs to be made more precise. Following Mell and Grance (2011), we see *a cloud system* as a collection of services provided through shared, often virtualized, resources. Furthermore, we define a *cloud architecture* as an abstract model of a distributed cloud system with the appropriate elements to represent not only application components and their interrelationships but also the resources these components are deployed on and the respective management elements.

Our view on cloud systems from an architectural perspective addresses the key shortcomings of the current discussion of control-theoretic approaches to adaptive systems, and cloud in particular. We argue for a model-based approach to controller definition. The cloud allows the distributed, tiered deployment of the full application stack. The architecture links infrastructure and platform providers with software applications running in them. Software is usually logically architected in a layered format (Kruchten 1995) but in the cloud mapped onto physical tiers and their native services. Logical layers organise code from a logical and a development perspective. Typical layers include presentation, business logic, and data management. However, this does not imply that the layers run on different computers or in different processes. Physical tiers are about the physical location of the application execution. Tiers are places where layers are deployed and where layers run. IaaS, PaaS, or SaaS (Mell and Grance 2011) realise the tiers, albeit in a virtualized form accessed through services (Figure 1). Services abstract software from underlying technologies, possibly in the form of microservices. Virtual machines and containers as virtualization units implement application components in the stack of infrastructure and platform technologies.

The cloud is a system of services with software application, platform, and infrastructure functionality. A characteristic of these cloud services is that of self-managed entities. All nodes in a cloud architecture should handle management and recovery tasks. The roles involved here from a software engineering perspective are worth noting:

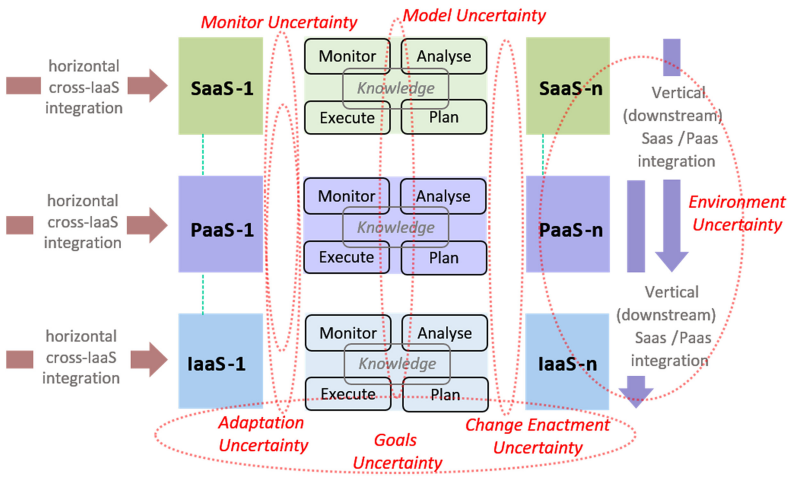

Fig. 1. A tiered and distributed cloud service architecture.

—The developer of a software system would use the PaaS layer to develop, deploy, and operate the software in a fully cloud-based scenario.
—The software user would access this as a software application at the SaaS layer.
—The cloud IaaS or PaaS provider provides platform services for development and operation, including service to monitor and manage quality.

Clouds are distributed, often federated, systems. Interaction between the layers and horizontally within the layers is often necessary, which we capture in the architectural scenario in Figure 1. Furthermore, adaptiveness based on models used at runtime requires a feedback loop (MAPE-K) (Baresi and Ghezzi 2013) to be implemented in a controller architecture. Figure 1 illustrates this with the *k*nowledge component holding the quality models that represent quality-of-service requirements, with the specific requirement in clouds to possibly having to support separate controllers for the different layers. The cloud controllers, which can be layer specific, with their *m*onitor, *a*nalysis, *p*lan, and *e*xecute functions, can be configured and customised by the consumer and application. The architecture of infrastructure IaaS_n, platform PaaS_n, and application services SaaS_n with their controllers are subject to uncertainties, which we discuss further in Section 4.

3.2 Architecture Illustration

To supplement the identified style characteristics of cloud systems, we use two sample use cases. Let us illustrate a simple cloud architecture operations problem. An infrastructure server might have the capacity to deal with 100 user applications at the same time, but the workload might temporarily reduce significantly. Load balancing would allow the system architecture to be adapted and applications relocated to one server, thus scaling down the deployment of servers. In our understanding of a cloud system as an integration of application and platform components, the architecture and/or the configuration of the deployed system change as a result of quality management activities, for instance, through load balancing in multi-cloud. Here, the system reacts to external factors—the reduced load—and adapts the configuration to reduce the costs (a non-functional requirement) while still maintaining adequate performance (also a non-functional requirement). Two key observations are as follows:

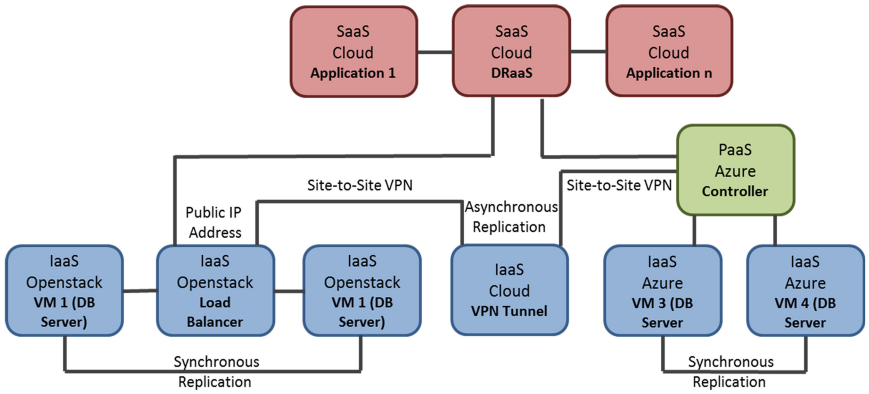

Fig. 2. A sample cloud architecture use case.

—Workload and QoS, e.g., workload bursts or performance degradations, dictate the adaptation. Cost and quality as drivers for decisions, i.e., decisions are made based on non-functional requirements.
—In the cloud as a layered architecture, user applications might run on remote third-party provided infrastructure servers. Factors that influence here down-scaling as the adaptation include (i) application performance for the user and (ii) system workload for the infrastructure provider.

Other scenarios here could involve changing goals rather than changing environmental factors. The performance requirement might need to be tightened, resulting in an up-scaling (rather than down-scaling) of the infrastructure components.

A more substantial application illustrates the need for cross-tier, distributed cloud systems with application and platform/infrastructure components in the cloud, see Figure 2. A cross-cloud HADR service is a SaaS service that is typically based on a hybrid PaaS/IaaS system (Xiong et al. 2015). A HADR service shall be introduced that enables hybrid multi-cloud cluster replication as a service. For the HADR service, we propose synchronous multi-master replication inside of a database cluster to achieve system high availability and data consistency and use asynchronous master-slave replication between primary and secondary database cluster to ensure that the backup/replication operations does not impact the normal performance of primary system. The HADR service defines an architecture pattern, with problem and context (principles) description, as well as solution and strategies (patterns). We assume two virtual server clusters to provide HADR for the databases. The primary site could host an OpenStack private cloud, which handles all client requests during normal operation. The standby site, hosted in an Azure public cloud, provides the database service when a failure happens at the primary site. We use a load balancer to manage the workload between the sites. OpenStack and Azure are used for illustration but are not part of the actual pattern.

Technically, the HADR pattern architecture uses site-to-site a Virtual Private Network (VPN) and cluster replication (synchronous and asynchronous) of the open-source relational database system MySQL. The site-to-site VPN is a private virtual cloud network that overlays the Internet to connect different cloud infrastructures, which is the networking precondition of database cluster replication. The cluster provides shared-nothing clustering and auto-sharding for the MySQL database system, which consists of multi-master synchronous replication and master-slave asynchronous replication. In terms of Figure 1, the three layers require vertical downstream integration but also horizontal cross-layering if several cloud platforms are used:

—IaaS: VM management for the server clusters and the VPN communications backbone are cross-cloud infrastructure concerns,
—PaaS: database cluster replication and management relies on platform functions,
—SaaS: HADRaaS allows user access to manage their HADR settings though a console.

Uncertainty arises here as two heterogeneous cloud solutions are combined and managed across different layers. The need for autonomic control is self-evident in a high-availability and disaster management solution. This also makes the point that high availability should be enforced through auto-management and self-healing, which requires a controller that operates on the relevant quality models. Thus, the proposed cloud architectural style concepts are applied as follows in HADR:

—Adaptive and control loop: Load balancing and scaling as adaptive measures have been applied, using a control loop to implement adaptiveness. Replication and management are further adaptive techniques in the HADR use case.
—Virtualization and composition: There are covered through virtual servers, VM and the VPN. VMs are independently deployable (if standards-compliant), but this shows the benefits of, e.g., moving to containers as a more interoperable and flexible solution.
—Uncertainty: This occurs in the HADR example through failure and the linkage of two heterogeneous cloud solutions.

4 CLOUD ARCHITECTURAL PRINCIPLES

We have identified service orientation, virtualization, uncertainty, and adaptivity as style-specific principles in Section 2. We specifically focus on *uncertainty* and *adaptation*, which are less commonly agreed on than service orientation and virtualization and require a better motivation for their inclusion in the architectural style. Service orientation is an agreed-on principle for the cloud (Mell and Grance 2011). Virtualization, though not part of some cloud definitions, is generally considered a necessity to manage a shared pool of cloud resources effectively, with optimised cost benefits and economies of scale. The four principles will be explained in terms of a common template that follows other pattern descriptions: *Motivation* reflects on the challenges, *Solution* includes the key concerns of the principle, and *Effect* covers impact and dependencies.

Service orientation. Service orientation is a grouping of principles, cf. Table 1, defining the SOA style, including layering, modularity, and loose coupling, which is a relevant organisational principle of the cloud (Jamshidi et al. 2015). The SOA principles are adopted from Zimmermann (2009).

Virtualization. The virtualization of resources in the cloud is an essential contributor to elastically manage and provide these resources as services (Antonopoulos and Gillam 2010). The virtualization principle is defined in Table 2. We can distinguish three forms: First, infrastructure virtualization separates physical infrastructures to create dedicated (virtual) resources. Second, platform virtualization through containerisation, which abstracts the platform (technology stack), achieving flexible virtualized application management. Finally, application virtualization to separate applications from the underlying platform and infrastructure so that applications may run parallel to other applications while moving across other platforms.

Adaptation. A set of model dimensions helps to frame the adaptivity problem (De Lemos et al. 2013). To contextualise the adaptation cloud principle in its wider adaptivity context, we align the general modelling dimensions to the cloud.

—*Goals*: Goals as system objectives: evolution, flexibility, multiplicity, duration, and dependency. In the cloud, multiple and interdependent goals (e.g., cost and performance) can

Table 1. Principle: Service Orientation

Principle	Explanation
Layering	
Motivation:	Service characteristics such as interface granularity and lifecycle vary, e.g., between a technical logging service and a business process.
Solution	We organize the SOA into 3++ architectural layers.
Effect	More indirections, however, require a communications infrastructure.
Modularity	
Motivation	Integrating monolithic applications such as traditional enterprise packages is hard.
Solution	Refactor into services, expose service interfaces only and encapsulate implementation details.
Effect	Services have to be located and invoked in a coordinated manner and service invocations should be free of undesired side effects (state management).
Loose Coupling	
Motivation	Once applications have been modularized, dependencies between services occur.
Solution	Couple services loosely across several dimensions. E.g., a messaging system decouples in time, location, and platform dimensions.
Effect	Each service provides a single endpoint (a message receiver). Interactions with this endpoint should be stateless. Therefore, conversational sessions require a correlation logic. Such asynchronous communication design complicates systems management.

evolve and are often expressed in flexible terms (from informal to formal, certain to uncertain). They can vary in duration.

— *Cause for adaptation*: Change captures causes of adaptation: source, type, frequency, and anticipation. In the cloud, resources provide information through monitoring but are subject to uncertainty. These can be categorized by resource and quality type. The frequency varies. Thus, anticipation using prediction techniques is essential.

— *System reaction to change*: Mechanisms to implement adaptation: type, autonomy, scope, duration, timeliness, and triggering. In the cloud, different actions are possible, for instance, scaling. These can be autonomous or manual and can cover different resource types. Some latency from when the change is triggered to when the change exists (change needs to go through several layers), the duration is typically short-termed with a need to react swiftly (timeliness), triggered through feedback loop.

— *Impact of Adaptation on System*: define adaptation impact: criticality, predictability, overhead, and resilience. In the cloud, non-operational aspects apply, covering how mission/safety critical an adaptation is or the significance of the adaptation in the context of possible failure and whether adaptation impact is durable/persistent.

A challenge is mapping requirements to the underlying architecture. We focus on the operational aspects later: cause for change and reaction to adaptation. Their application to the cloud is reflected in the principle definition in Table 3.

Uncertainty. To deal with uncertainty in cloud architectures, we need to measure at different layers and map between the different tiers in the cloud. Upper levels represent application-level

Table 2. Principle: Virtualization

Principle	Explanation
Motivation	Cloud computing with sharing of pooled resources only pays off at scale. Elasticity is required to manage changing demands and environmental conditions and deploy loads flexibly.
Solution	*Infrastructure virtualization* is software that separates physical infrastructures to create various dedicated (virtual) resources. Infrastructure virtualization software in the form of hypervisors and elastic infrastructures (Fehling et al. 2014) makes it possible to run multiple operating systems and multiple applications on the same server at the same time. It enables businesses to reduce IT costs while increasing the efficiency, utilization, and flexibility of their existing computer hardware. *Platform virtualization* is achieved through containerisation, which abstracts away the platform (technology stack) rather than the hardware. Containers build on infrastructure virtualization techniques for process isolation but enable full-stack lightweight and portable application containers (e.g., LXC or Docker) to be assembled on top of platform components (He et al. 2012; Pahl 2015). *Application virtualization* separate applications from an underlying platform or infrastructure. This enables applications to run parallel to others while moving across other platforms.
Effect	Cloud virtualization provides self-service capability, elasticity, automated management, scalability, and pay-as you go service that is not inherent in virtualization alone. Infrastructure virtualization requires process isolation to manage security concerns if resources are shared. Platform virtualization requires a container support to achieve portability at platform level.

Table 3. Principle: Adaptation

Principle	Explanation
Motivation	Systems should self-adapt with minimal human involvement. Systems must be able to cope with variable resources, system errors, and changing user characteristics, while maintaining the goals and properties envisioned by the developers and users.
Solution	Adaptation is a process in which an adaptive system adapts its behaviour based on information acquired about its user(s) and environment (Cheng et al. 2009).
Effect	Drivers of adaptivity are often requirements to maintain quality-of-service at the user end to stay within the (non-functional) limits possibly stated in a service-level agreement.

qualities, e.g., performance for different storage configurations. Lower levels are loads of infrastructure resources that run the service (processor and memory loads). Furthermore, there is a mapping of the infrastructure loads into a cost model—which can, of course, be a major driver of adaptation decisions. Mappings of metrics between layers, e.g., predicting service-level response times from infrastructure measurements (Zhang et al. 2014), adds a degree of uncertainty. Multi-cloud deployments with different monitoring mechanisms and the interference of the network in the calculation of performance metrics is equally a contributor of uncertainty.

Table 4. Principle: Uncertainty

Principle	Explanation
Motivation	Uncertainty emerges from various sources in cloud systems—as uncertainty from different interpretations and decisions in the adaptation definition process or as uncertainty arising from possible different, distributed monitoring systems resulting in partially unreliable and incomplete data.
Solution	Models also reflect how we deal with uncertainty in dynamic systems.
Effect	A system's current non-functional properties need to be aligned with non-functional requirements. Due to the layering, mapping and managing these across layers, but also within one layer, is challenging. Service Level Agreements (SLAs) become somewhat fuzzy.

Ideally, system qualities can be reliably measured. However, the cloud adds a high degree of uncertainty. This uncertainty can be captured in uncertainty levels (Sawyer et al. 2010) from general confidence about the shape of the future, with some key variables not having precise values, to the impossibility to frame possible scenarios. Different sources of uncertainty can be identified (Jamshidi et al. 2014):

— *Uncertainty in adaptation and its models.* Adaptation thresholds rely on knowledge of system behaviour and how resources are managed. The accuracy of policies remains subjective, making them prone to uncertainty. Unpredictable changes in environment or application demand may require policies to be continuously re-evaluated.
— *Uncertainty in the dynamic provisioning environment.* Acquiring or releasing resources in the cloud is not instantaneous. During this time (minutes for VMs) the application is vulnerable to workload increases, causing uncertainty.
— *Uncertainty in monitoring data.* The controller needs to continuously monitor application states as well as of resources in which applications are deployed to timely react to load variations. Monitoring involves data collected by measurement-specific probes or sensors, which are not immune to measurement deviations (sensory noise). This sensory noise is another source of uncertainty.
— *Uncertainty in change enactment.* Even the same change can take different times depending on uncertainty in underlying resources.

This is reflected in the virtualization principle in Table 4 and was applied in Figure 1, where sources of uncertainty were already identified.

5 CLOUD ARCHITECTURE PATTERNS

Architecture patterns provide solution templates that realise the principles. We can distinguish three concerns (composition from a *development* perspective and dynamic models and adaptation from an *operations* perspective) that we condense into patterns (microservices, dynamic models, control loop) as outlined in Section 2.1:

— *Composition of services: Microservices.* A mechanism is needed to decompose and map a system onto the layered, distributed architectural framework (Figure 1) as rapidly deployable, locally manageable units (Balalaie et al. 2016).
— *Models driving operation: Dynamic models for uncertainty.* A model-based solution to manage the external uncertain factors influencing the system is required.

— *Managing adaptation: Control loops.* An architecture that adapts cloud systems to changing environmental conditions is needed.

The following research directions have influenced the concerns identification and formulation as separate patterns: Resource management needs to be understood in terms of scheduling and load balancing activities in the cloud environment (Hu et al. 2010; Antonopoulos and Gillam 2010). Some approaches combine the concept of "models@runtime" with dynamic analysis of monitoring data (Cedillo et al. 2015; Heinrich et al. 2014). Maintaining SLAs in a cloud environment can be managed based on virtualization, monitoring, and adaptation features, for instance Stantchev and Schräpfer (2009), Van Hoorn et al. (2009), and Stantchev and Schräpfer (2009) discuss quality and SLA enforcement in cloud environments. Zhang et al. (2014) look further into this linking workload patterns and SLA quality requirements.

We introduce three patterns in the subsequent subsections. The patterns including *microservices, adaptivity*, and *control loops* will again be explained in terms of the previously introduced template: Motivation, Solution, and Effect.

5.1 Microservices as Composition Format

Composability of services is an underlying principle of service-oriented architectures. Recently, the notion of microservices has been discussed as a suitable unit for flexible service-based system composition in environments such as the cloud that are based on deployment and management automation (Newman 2015). A microservice architecture is a way of architecting software applications as independently deployable services. While no commonly agreed definition of microservices exists, there are some common characteristics (Zimmermann 2016; Pautasso et al. 2017):

— Organization around business capability and evolutionary design for independent replacement and upgradeability,
— Automated deployment and infrastructure automation, towards the use of cloud-native services. Lightweight containers are used to deploy services, with decentralized continuous delivery during service development,
— Intelligence in the endpoints, rather than in communication mechanism (as in enterprise service buses); with heterogeneity and decentralized control of governance and of data, with fault tolerance and design for failure in a DevOps style.

We have captured the pattern in Table 5. Microservices as a unit of service composition need to be complemented by *application packaging and orchestration mechanisms* towards a support of DevOps and continuous development. In the cloud, microservices as a pattern can be implemented through recently successful *container technologies* such as Docker and cluster management for containers such as Kubernetes, which are considered suitable to implement the architectural framework in Figure 1 (He et al. 2012; Pahl 2015). Thus microservice architecting with Docker and Kubernetes would be a *concrete instance of the pattern.* So-called *cloud-native architectures* include as far as possible platform services provided directly in the cloud rather than virtualising legacy software, which results in better control of cost and quality.

Topology definition and orchestration in and across microservices/container clusters is currently researched, with solutions focusing on orchestration engines but neglects abstraction and reuse constructs (Pahl and Lee 2015) due to lack of machine-readable contracts that composition can be based on.

Table 5. Pattern: Microservices

Principle	Explanation
Motivation	Microservice architectures have been proposed to break up application architectures into independently deployable services that can be rapidly deployed to any infrastructure resource as required to achieve speed and flexibility.
Solution	Microservices are independently deployable, usually supported by a fully automated deployment and orchestration framework. Microservices typically are deployed often and independently at arbitrary schedules, instead of requiring synchronized deployments at fixed times. Microservice deployment and orchestration across cloud vertical and horizontal dimensions are central architecture concerns. Clouds provide a management tool for flexible deployment schedules and provisioning orchestration needs, particularly, if these are to be PaaS-provisioned. For example, verbs (e.g., Checkout) or nouns (Product) of an application are good indicators to decompose. For example, product, catalogue, and checkout can be three separate microservices and then work with each other to provide a complete shopping cart experience.
Effect	Changing an independent service implementation has no impact on other services as they communicate using defined interfaces. There are several advantages of such an application, but this traditionally requires a significant effort. Functional decomposition of an application and the team is the key to building a flexible microservices architectures quickly. This achieves loose coupling and high cohesion (multiple services can compose with each other to define higher level services or application). Functional decomposition enables agility, flexibility, and scalability.

5.2 Adaptive Model-based Architectures—Quality Models at Runtime

As the uncertainty discussion indicates, both requirements (user-facing tiers) and the platform (infrastructure-facing tiers) can change dynamically. As for early sections, we review the state-of-the-art before formulating a cloud-specific patterns for this.

Model-driven engineering (MDE) is a methodology based on domain models (Stahl et al. 2006). It aims at abstract representations of the knowledge and activities that govern a particular application domain. MDE uses domain-specific modelling languages whose type systems formalize application structure, behaviour, and requirements within domains (Van Deursen et al. 2000). Domain-specific modelling languages (DSMLs) are described using metamodels, which define the relationships among concepts in a domain and their semantics and constraints. Transformation engines and generators analyse aspects of models and then generate artefacts such as executable code (Czarnecki and Helsen 2003). Automated generation aids consistency between application implementations and analysis information associated with functional and QoS requirements captured by models.

5.2.1 Dynamic Runtime Models. Domain models for dynamic quality management need to be developed and enabled for processing in self-adaption activities in the cloud. Vogel and Giese (2014) and Wätzoldt and Giese (2014) propose feedback loops for software maintenance and self-adaptation and categorise models accordingly, e.g., runtime models for monitoring and execution

Table 6. Pattern: Models at Runtime

Principle	Explanation
Motivation	A challenge is the uncertainty that arises in the interaction between models and the system architecture—the latter possibly at different tiers/layer, all interacting with one another along their interfaces, cf. Figure 1. Respective models that capture uncertainty and can map this as actions within the control loop are needed. The models need to reflect the adaptation approach and capture the non-functional properties but more significantly allow prediction and reasoning to take place in an environment prone to uncertainty.
Solution	The control loop, based on control-theory (Sawyer et al. 2010), needs to map the layering of the application architecture onto the tiered cloud. The runtime representation of requirements in the form of application requirements and cloud infrastructure models needs to provide model manipulation and access features to allow introspection and reasoning about the models (Baresi and Ghezzi 2013; Ghezzi et al. 2013).
Effect	Runtime models for adaptation can fulfil a range of purposes, including Monitoring in Maintenance, Execution in Maintenance, or Self-Adaptation.

in an evolutionary setting or for self-adaptation. Challenges in this context are (i) how to systematically derive runtime models from development models and (ii) how to find an MDE style that works for continuous development. Runtime models can be categorised based on their purpose:

- —Reflection: models describe aspects of interest from the running system and system context on a higher level of abstraction.
- —Adaptation: models contain requirements as well as the configuration space of the adaptive system and are mainly used by the 'analyse' and 'plan' activities.
- —Causal connection models: establish the causal connection to the running system and therefore primarily used during the monitoring and execution feedback loop.
- —Collaboration models: coordinated interaction of distributed feedback loops.

The pattern definition in Table 6 reflects this. The MAPE-K loop and the different controller types suggested embody reflection and adaptation, connecting the model to the cloud running system and coordinating between layers. A primary concern of models for cloud self-adaptation is to deal with uncertainty. Suitable formalisms need to represent adaptation in the presence of uncertainty, i.e., the models are reflective of uncertainty, adaptive to the cloud platform parameters, connective regarding the layers, and also collaborative across multi-cloud scenarios in terms of the above categorisation. We address the specific nature of models for uncertainty now before looking at their management by a controller architecture to enable adaptation in Section 5.3.

5.2.2 A Pattern Instance—Fuzzy Models for Uncertainty. Models here capture system state, its behaviour, and the adaptation rules to deal with uncertainty in dynamic systems. The dynamics of a system are often based on state models, describing sequences of possible actions as a protocol. In Baresi and Ghezzi (2013), a Markovian model is used, formalising specific properties in logics such as a probabilistic logics (Chan et al. 2005) to reason in uncertain spaces—in an uncertain space, the probability of the next state is included in the model. Others propose fuzzy logic (Jamshidi et al. 2014), where fuzziness is expressed as a varying, non-binary truth value. This allows the uncertainty of a system situation to be expressed through a membership functions on fuzzy sets. Also

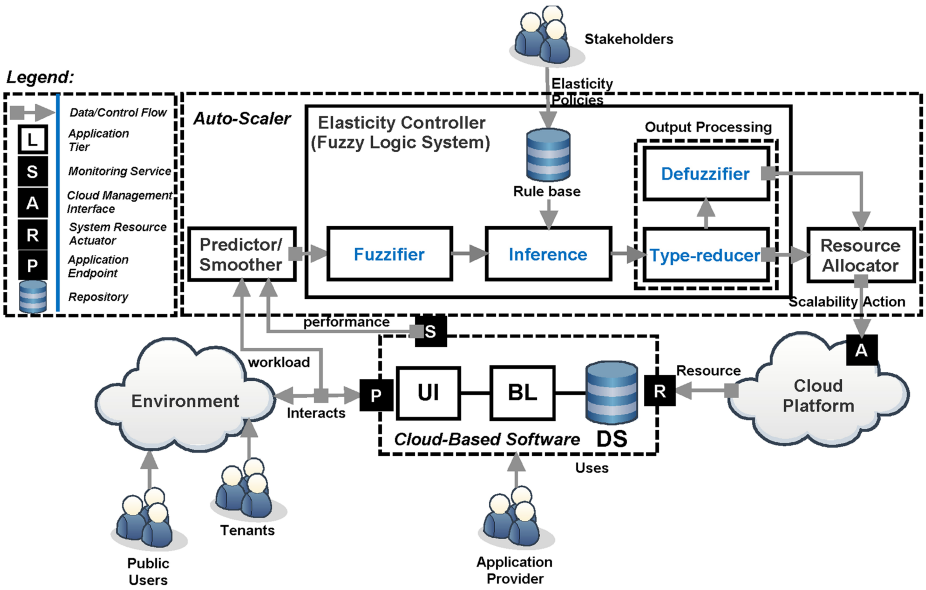

Fig. 3. Fuzzy model reasoning—example for scalability adaptation.

Bayesian approaches have been considered to handle uncertainties with respect to the changes in the assumptions at runtime.

A sample *pattern instance* is a fuzzification of adaptation rules (Jamshidi et al. 2014). Qualitative values for infrastructure workload and service performance (such as "very low (VL)" or "very high (VH)" for workload) can be presented as membership functions in a fuzzy set model, which results in smoother controller responses as we discuss in Section 5.3. Figure 3 visualises the adaptation rule model for fuzzified workloads. The fuzzy membership functions guide the adaptation rule selection (left box in Figure 3). The rule is then applied and the adaptation action as the rule consequent is determined through type reduction (using so-called type-2 fuzzy sets) and defuzzification determines a crisp output value.

5.3 Control Loop and Controller Architecture

A control loop is implemented by a controller, which in turn is driven by runtime adaptation models that were discussed in the previous section. To better understand the interaction between the patterns, we explore controllers and underlying control theory, models, and adaptation loops in more detail. We use techniques for specific pattern instances to illustrate concepts and implementation concerns but also how the principles are addressed, see Table 7. Prediction and robustness are uncertainty measures, which are part of this platform pattern to deal with adaptivity and uncertainty.

We start with prediction as the first mechanism to reduce the impact of uncertainty before looking at control theory to manage this dynamically. A utility function is the core mechanism of a controller. How to model, design, and implement a pattern is described in a dedicated controller construction section.

5.3.1 Analysis and Prediction—Cross-Tier Mapping and Uncertainty. Unreliable or incomplete data causes uncertainty, which can be alleviated to some extent by prediction. Furthermore, the delay in providing resources, as discussed above, also makes prediction a suitable approach. Two

Table 7. Pattern: Control Loop

Principle	Explanation
Motivation	Runtime models are at the core of adaptive systems, but an architectural solution is needed to manage the monitoring, analysis, and adaptation process.
Solution	Control theory and control engineering can be applied to build control loops self-adaptive systems in uncertain environments. Control theory helps to build the models and reasoning about them to inform the decision making (De Lemos et al. 2013), enacted by a controller.
Effect	Prediction is a technique to address the uncertainty in the adaptation process. This adds to the robustness of the controller, i.e., the resilience of the controller is against noise and uncertainty.

aspects emerge. First, analysing measurement data allows us to predict behaviour, reducing uncertainty and increasing the robustness of the adaptation. Second, prediction also helps to link the layers and tiers in the architecture, as, for instance, infrastructure tier metrics can be used to predict service-level quality. Prediction captures dependencies and links the tiers, cf. Figure 1.

Services across tiers can be linked through workload utilisation patterns (not to be confused with the style patterns). Their dynamic orchestration might be guided by infrastructure metrics, e.g., on workloads. Through prediction and analysis of monitored data, we can, for instance, identify stable quality utilisation patterns. We map workload patterns for CPU, storage, and network utilisation at the infrastructure tier to service-level performance patterns, thus linking layered models (here pattern-based) across tiers (Zhang et al. 2014). We can map infrastructure workload patterns $WP_i = [CPU, Storage, Network]$ to predicted application service-level performance ranges $[p_i^1, p_i^2]$ for services. The implementation of a prediction technique for workload and performance prediction as in Zhang et al. (2014) can be based on exponential smoothing to smoothen outliers and to anticipate trends. Here the aim is the robustness of the prediction and overall adaptation process against uncertainty (by looking ahead in vulnerable moments when the system is about to change).

5.3.2 Constructing a Utility Function. Decision making in the adaptation process is a multi-objective process (Sawyer et al. 2010) that can be formalised by constructing a utility function that involves all stakeholders (end-users and providers of the various tiers of the system) (Van Hoorn et al. 2009). This function is implemented by the cloud controller. The construction of utility function (the model) and controller is a process involving the following steps (Filieri et al. 2015): identify goals, identify measure, devise model and design controller, complemented by validation and verification steps.

A key property of this controller is robustness, i.e., how resilient the controller is against noise and uncertainty. Prediction is in addition to a proper calibration of the model a contributor to robustness. Prediction across layers addresses challenges arising from the tiered cloud architecture. Horizontal scaling and load balancing can deal with the distribution dimension at each tier.

5.3.3 Construction of a Controller. The quality concerns need to be managed by a control loop. Often, the MAPE-K model (Baresi and Ghezzi 2013) is utilised as the structure of a controller, cf. Figure 4: *M*onitor application and environment (in control-theoretic terms disturbances such as workload); *A*nalyse the input data and detect any possible violation.; *P*lan corrective actions in terms of adding resources or removing existing unutilized ones; and *E*xecute the plan according

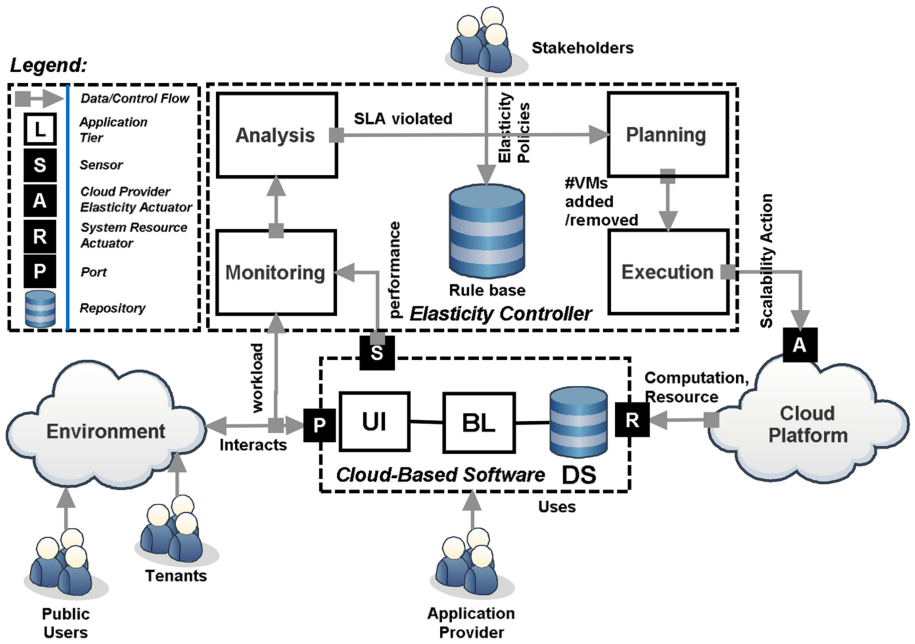

Fig. 4. MAPE-K control loop for the cloud.

to a specific platform. This utilises a shared *K*nowledge model at the centre. This knowledge component is based on the different model types discussed in Section 5.2.1.

The controller synchronises runtime adaptation models with the runtime architecture (Sawyer et al. 2010). The model part of the controller needs to be implemented and integrated with the cloud architecture to allow a model-based cloud control of quality aspects (Filieri et al. 2015). Controller construction faces challenges (Farokhi et al. 2015), including uncertainty, heterogeneity between resources and providers, unpredictable workloads and resource bottlenecks, all in multi-tier applications and multi-cloud resources environments. We have already discussed uncertainty and unpredictability in resource management and have addressed multi-tier and multi-cloud settings through our cloud architecture. The challenges emphasise the cloud as a problem from the software architecture perspective, i.e., an architecture onto which the concerns need to be projected: (i) Measurement: the controller integrates the different application and infrastructure models at the different tiers (the vertical dimension of layers and tiers). (ii) Actuating/executing: typically within a tier, across services, e.g., based on scalability actions as adaptations (the horizontal multi-cloud dimension).

5.3.4 Full Self-Adaptiveness—A Learning Loop to Replace Humans. There is a discussion on whether self-adaptiveness can be implemented by control loops purely based on predefined rules or whether a degree of (machine) learning needs to be involved to call a system self-adaptive. We therefore look here into the perspective of a second loop, i.e., a learning loop, in addition to the adaptation loop for the main controller. Uncertainty (Filieri et al. 2015) could be addressed by reducing the dependency on human stakeholders for assessing adaptation needs and defining respective rules manually.

Machine learning is another *pattern instance* that can serve to learn adaptation rules rather than relying on uncertain, possibly erroneous or inconsistent, user input or somehow externally

monitored unreliable data. Again, the software architecture perspective can clarify this. Based on Figure 4, we can add a meta-model layer on top of the MAPE-K control loop, representing the learning loop for model construction. Models can provide prediction and the feedback loop can correct it. For example, a queuing model provides how many resources are needed to guarantee an SLA. Various loop architectures with nested and interleaved control loops can allow the optimisation of the efficiency of their own activities. Cloud providers might try to minimize energy while application providers aim to minimize rent cost and satisfy SLAs.

Since models are often not precise, they can be augmented with a feedback mechanism to correct the error, called feed-forwarding (Hou and Xu 2007). This refers to a pathway within a control system which passes a controlling signal from a source in its external environment, often a command signal from an external operator, to a load elsewhere in its external environment. A control system that has only feed-forward behaviour responds to its control signal in a pre-defined way without responding to how the load reacts. This differs from a feedback system, which adjusts the output to take into account the load and how the load itself may vary unpredictably; the load is considered to belong to the external environment of the system. This short discussion signifies the importance of real self-adaptation in uncertain cloud architectures.

6 APPLICATION AND DISCUSSION

We have validated the architectural style by following a *technical action research (TAR)* (Wieringa and Moralı 2012) strategy to demonstrate the validity of the identified principles and patterns. In TAR, a researcher aims to investigate a concern about a technique by using it to solve a partner's/clients problem. We selected three use cases that we conducted with industry partners to validate the applicability and validity of the proposed style in three very different settings: the HADR architecture as a cross-layered platform service (with Microsoft), OpenStack as an infrastructure system (with Intel), and a SaaS application system (with an Irish small and medium enterprise). We deepen the earlier HADR discussion and summarize the others more briefly here.

6.1 Use Cases—Development, Management, Implementation, and Evaluation

Cloud development and operation links software requirements with cloud-native platform services, e.g., to manage performance and cost across the software lifecycle. We have already introduced the HADR system, a multi-cloud application that provides fault tolerance as a non-functional requirement by managing back-up resources in an uncertain environment. In terms of style elements, the HADR service controls fault tolerance and availability rather than performance, as most other controllers do, but still under uncertainty using virtualization *principles*. We can map functional and non-functional requirements (such as adaptive qualities like fault tolerance, availability, performance, and cost) onto *patterns*. Figure 2 showed two controllers (IaaS load balancer and Azure PaaS controller) that adapt system behaviour based on respective availability models (PaaS controller) and recovery models (IaaS controller). Applying the style principles and patterns enables evolvable and adaptive systems that respond to change and that works under uncertainty if controllers for adaptation and a microservice architecture is available for evolution (evaluated in Xiong et al. (2015)).

In addition to the HADR system that mixes infrastructure and application concerns, we also implemented two other systems according to the style guidelines that highlight the downstream integration needs between the layers:

—Linking IaaS and PaaS: We implemented an OpenStack-based controller (i.e., a platform system) that manages performance and cost through a fuzzy logic models in two variants: one

that works on a fuzzified model of expert input and one that is based on machine learning (Arabnejad et al. 2016, 2017). Both increase robustness and scalability.
—Linking PaaS and SaaS: We implemented a document management system (i.e., an application system) in the cloud in cooperation with a software vendor. We did this as a two-stage migration, first, into an IaaS system virtualization and, second, as a cloud-native PaaS system. In both cases, the system was provided as a SaaS solution.

The work on HADR and OpenStack has highlighted the importance of dynamically adaptive uncertainty management through a model-based controller to increase robustness in the quality management process. Explicit resource management was important in the software product case (e.g., through scalability testing from early stages), where the flexible microservice and native service supported architecture was central to create not only a manageable but also extensible and maintainable product.

Tools play an important role as instances of the pattern. For DevOps and microservices support, we currently use container orchestrators (Docker Swarm) (Pahl and Lee 2015). For the OpenStack platform, we even developed our own FQL4KE controller (Jamshidi et al. 2016) (but commercial tools like Amazon Cloudwatch also exist). These allow quantitative and qualitative scaling rules as adaptivity rules to be enacted.

6.2 Discussion—Benefits and Implications for DevOps and Internetware

The cloud is a distributed, multi-tiered platform onto which layered, modular software (micro) service architectures are mapped. Virtualization of cloud resources increases the adaptivity of these service-based systems that is, however, subject to uncertainty and other challenges. Our contribution is a proposal of architectural style elements from a software architecture perspective that addresses the needs of Internetware, i.e., focusing on software paradigms for Internet-centered software systems to be autonomous, situational, or evolvable. We have demonstrated its validity by investigating use cases from three very different architectural settings in a TAR approach.

The use cases, which involve development and maintenance aspects for applications but also platform technology development, demonstrate the benefits of development methods to model uncertainty and respective adaptation actions and automation, e.g., though controllers, for enacting these adaptations, specifically in the layered/tiered cloud architecture. The benefits of applying style principles and patterns is a *continuous development and maintenance of cloud systems*. For the adaption, the evaluations of the controllers have demonstrated *cost-effectiveness and performance improvements*.

The principles and patterns of our architectural style are based on accepted but also emerging concerns. Models and control theory are at the core to improve known cloud techniques such as elastic load balancing or autoscaling with the following benefits: (i) *robustness*: models for uncertainty allow prediction and enforce robustness; (ii) *multi-tier integration*: multi-tier controller manage layered builds; and (iii) *automation*: adapting the configuration and architecture itself in the cloud. There is a need for a controller framework that manages the required "watchdog" capabilities for the adaptation process but at the same time addresses the layered architecture of an application mapped onto tiered cloud resource services through a set of linked models for robust control-theoretic uncertainty management. This supports quality management in the presence of SLAs, e.g., autoscaling helps to improve performance and costs concerns as reflected in the quality-oriented models.

We have included accepted style elements such as service-orientation and virtualization with the resulting adaptiveness of clouds. However, to make this *truly self-adaptive*, more methodological and tool support needs to be added. Some of our principles and patterns have clearly already

emerged, such as uncertainty management, but others are less widely accepted, such as DevOps, microservices, and their orchestration as containers and its wider continuous development context.

Our validation suggests the need to aid DevOps practices. In DevOps, operation performance data is fed into the development. We can enable cloud systems to take care of anomalies by themselves through learning and adaptation and only feed back partial data, which cannot be taken care of automatically and require human intervention.

7 RELATED WORK—BEST PRACTICE IN INDUSTRY

While we have reviewed aspects of adaptive systems, continuous development, and service orientation under Related Work in the context of architectural styles in Section 2, we now discuss this from a wider cloud and cloud development perspective, taking in particular industry views into account. The proposed concepts align well with the current reflection of software architecture in the cloud.

The OSSM (On-demand, Self-service, Scalable, Measurable) idea[2] condenses the cloud into four key concepts: (i) On-demand: The server is setup and ready to be deployed, which we address through packaging and service orientation. (ii) Self-service: The customer chooses what he or she wants, when he or she wants it, which we address through DevOps, service orientation, and adaptiveness. (iii) Scalable: The customer can choose how much they want and ramp up if necessary, which we address through adaptation in a control loop. (iv) Measureable: Metering/reporting is available, which we address through quality-driven dynamic models. OSSM entails the need for APIs for all aspects, towards enabling full autonomy and self-adaptiveness as we introduced in our architectural style proposal.

Similarly, IBM's Cloud Computing Reference Architecture builds on the Efficiency, Lightweightness, Economies-of-scale, Genericity principles.[3] We address the first two concerns here through adaptation and application packaging through microservices, the latter via virtualization and service orientation.

The IDEAL Cloud Application Properties (Fehling et al. 2014) include Isolated State (most of the application is stateless), Distribution (applications are decomposed to use multiple cloud resources), Elasticity (applications can be scaled out and up dynamically), Automated Management (runtime tasks have to be handled quickly), and Loose Coupling (such that the influence of application components is limited in case of failure etc.). These are taken care of here through containers and microservices, a distributed architecture, adaptation and control, and service orientation. Similar are Microsoft's 24 cloud design patterns[4] that support principles and patterns such as scalability and consistency or data management and service metering. A similar set of structural architecture design patterns for AWS is also available.[5] These are design patterns at a lower granularity than our style patterns that can be used to further detail microservices architectures through different storage or communication mechanisms.

Cloud providers have their own best practice guides highlighting concerns. For intance Amazon states that "applications will adapt themselves to fluctuating demand patterns by deploying resources instantaneously and automatically." Amazon Web Services (AWS) best practice[6] highlights automated elasticity and scalability, virtual administration (self-service), fault-tolerance (self-healing), loose coupling (SOA), and infrastructure automation (auto-scaling) as they concerns—all covered here. We have, though, not included security or data management here.

[2] http://www.daveslist.com/.
[3] http://www.infoq.com/news/2011/03/IBM-Cloud-Reference-Architecture.
[4] https://msdn.microsoft.com/en-us/library/dn568099.aspx.
[5] http://en.clouddesignpattern.org/index.php/Main_Page.
[6] https://aws.amazon.com/blogs/aws/.

8 CONCLUSIONS

We have reviewed both academic and industry-based development and management proposals for the cloud. This demonstrated the practical need for a dedicated methodology for cloud systems that takes the cloud characteristics of uncertainty or virtualization into account. From a software development perspective, a number of differentiating new trends are emerging. Cloud-native and continuous development, based on software platform virtualization through containers and using microservices, forms a distinct approach to software systems development.

Our proposal for core elements of a cloud architectural style takes not only these industry input but also previous academic work into account, i.e., has, therefore, a summarizing character. It applies to cloud systems based on the three-tiered architecture, including multi-cloud configurations. Of course, the degree of uncertainty or the willingness to configure and use adaptive systems and their models might vary. The benefits of following the style guidelines includes not only technical scalability and flexibility but also enhanced cost management. Specific challenges, e.g., in the big data or edge cloud and IoT integration, are, however, not addressed. Neither have we singled out security. While important, it also applies to other multi-user, distributed, and open systems, while we have concentrated here on cloud-specific, less researched design issues.

Naturally, this investigation cannot define an architectural style that is ready to be consumed by practitioners. We restricted ourselves to identifying the core building blocks, and more work is needed to map this into practice. We have considered horizontal cross-layer integration in our reference architecture but have not included principles such as clustering. Clustering is an important concept that could be considered as a possible extension/variation of the architecture if full cloud-native container-based architectures are the target. In summary, the contribution of this article is a survey-based definition of an architectural style and its building blocks that aims at guiding further research; while it is backed by implementation work and technical action research, it does not intend to be a complete and comprehensive practical guide. To this end, a combination with methods for application building such as the 12-factor app[7] would be a suitable complement requiring further investigation.

REFERENCES

A. Ahmad, P. Jamshidi, and C. Pahl. 2014. Classification and comparison of architecture evolution reuse knowledge – A systematic review. *J. Softw.: Evol. Process* 26, 7 (2014), 654–691.

N. Antonopoulos and L. Gillam. 2010. *Cloud Computing: Principles, Systems and Applications*. Springer.

H. Arabnejad, P. Jamshidi, G. Estrada, N. El Ioini, and C. Pahl. 2016. An auto-scaling cloud controller using fuzzy Q-learning—Implementation in openstack. In *Proceedings of the European Conference on Service-Oriented and Cloud Computing (ESOCC'16)*.

H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada. 2017. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid'17)*.

A. Balalaie, A. Heydarnoori, and P. Jamshidi. 2016. Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Softw.* 33, 3 (May 2016), 42–52.

L. Baresi and C. Ghezzi. 2013. A journey through SMScom: Self-managing situational computing. *Comput. Sci.-Res. Dev.* 28, 4 (2013), 267–277.

A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kross, S. Spinner, C. Vögele, J. Walter, and A. Wert. 2015. Performance-oriented DevOps: A research agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group (RG) DevOps Performance. https://arxiv.org/abs/1508.04752.

P. Cedillo, J. Jimenez-Gomez, S. Abrahao, and E. Insfran. 2015. Towards a monitoring middleware for cloud services. In *Proceedings of the International Conference on Services Computing (SCC'15)*. 451–458.

K. Chan, I. Poernomo, H. Schmidt, and J. Jayaputera. 2005. A model-oriented framework for runtime monitoring of non-functional properties. In *Proceedings Conference on Quality of Software Architectures and Software Quality*. Lecture Notes in Computer Science, vol. 3712. Springer.

[7] http://12factor.net/.

B. H. C. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, and others. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-adaptive Systems*. Springer.

K. Czarnecki and S. Helsen. 2003. Classification of model transformation approaches. In *Workshop Generative Techniques in the Context of the Model Driven Architecture*. 1–17.

R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, and others. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32.

T. Erl. 2005. *Service-oriented Architecture: Concepts, Technology, and Design*. Pearson Education.

N. Esfahani and S. Malek. 2013. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*. Springer, 214–238.

S. Farokhi, P. Jamshidi, I. Brandic, and E. Elmroth. 2015. Self-adaptation challenges for cloud-based applications: A control theoretic perspective. In *Proceedings of the 10th International Workshop on Feedback Computing*.

C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. 2014. Cloud computing patterns. Springer-Verlag Wien. DOI: 10.1007/978-3-7091-1568-8

A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel. 2015. Software engineering meets control theory. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'2015)*.

B. Fitzgerald and K.-J. Stol. 2014. Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE'14)*.

D. Garlan and M. Shaw. 1994. *An Introduction to Software Architecture*. Vol. 1. World Scientific.

C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. 2013. Managing non-functional uncertainty via model-driven adaptivity. In *Proceedings of the 2013 International Conference on Software Engineering*. 33–42.

S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han. 2012. Elastic application container: A lightweight approach for cloud resource provisioning. In *Proceedings of the International Conference on Advances in Information Networking and Applications*.

R. Heinrich, E. Schmieders, R. Jung, K. Rostami, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl. 2014. Integrating run-time observations and design component models for cloud system analysis. In *Proceedings of the 9th Workshop on Models@run.time*. 41–46. http://ceur-ws.org/Vol-1270/.

Z.-S. Hou and J.-X. Xu. 2007. New feedback-feedforward configuration for the iterative learning control of a class of discrete-time systems. *Acta Automatica Sinica* 33, 3 (2007), 323–326. DOI: 10.1360/aas-007-0323

J. Hu, J. Gu, G. Sun, and T. Zhao. 2010. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Programming*.

P. Jamshidi, A. Ahmad, and C. Pahl. 2014. Autonomic resource provisioning for cloud-based software. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*.

P. Jamshidi, C. Pahl, S. Chinenyeze, and X. Liu. 2015. Cloud migration patterns: A multi-cloud service architecture perspective. In *Proceeings of the Service-Oriented Computing Workshops (ICSOC'14)*. 6–19.

P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada. 2016. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *Proceedings of the 2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA'16)*. IEEE, 70–79.

S. Kounev. 2011. Self-aware software and systems engineering: A vision and research roadmap. *GI Softwaretech.-Trends* 31, 4 (2011), 21–25.

S. Kounev, F. Brosig, and N. Huber. 2014. *The Descartes Modeling Language*. Department of Comp Science, University of Wuerzburg, Tech. Rep (2014).

N. Kratzke and R. Peinl. 2016. ClouNS - A cloud-native application reference model for enterprise architects. In *Proceedings of the International Enterprise Distributed Object Computing Workshop (EDOCW'16)*. 1–10.

P. B. Kruchten. 1995. The 4+1 view model of architecture. *IEEE Softw.* 12, 6 (1995), 42–50.

J. Lewis and M. Fowler. 2014. Microservices. Retrieved January 7, 2018 from http://martinfowler.com/articles/microservices.html.

P. Mell and T. Grance. 2011. The NIST definition of cloud computing. Technical Report SP 800-145. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, US.

Microsoft. 2009. Software Architecture and Design. Retrieved January 7, 2018 from https://msdn.microsoft.com/en-us/library/ee658093.aspx.

S. Newman. 2015. *Building Microservices*. O'Reilly.

C. Pahl. 2015. Containerization and the PaaS cloud. *IEEE Cloud Comput.* 2, 3 (2015), 24–31.

C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. 2017a. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*. Published online at https://ieeexplore.ieee.org/document/7922500/.

C. Pahl and P. Jamshidi. 2015. Software architecture for the cloud - A roadmap towards control-theoretic, model-based cloud architecture. In *Proceedings of the European Conference on Software Architecture (ECSA'15)*.

C. Pahl, P. Jamshidi, and D. Weyns. 2017b. Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. *J. Softw.: Evol. Process* 29, 2 (2017).

C. Pahl and B. Lee. 2015. Containers and clusters for edge cloud architectures - A technology review. In *Proceedings of the 3rd International Conference on Future Internet of Things and Cloud (FiCloud'15)*. IEEE.

C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. 2017. Microservices in practice, part 1: Reality check and service design. *IEEE Softw.* 34, 1 (2017), 91–98.

R. Perrey and M. Lycett. 2003. Service-oriented architecture. In *Proceedings of the Symposium on Applications and the Internet Workshops 2003*. 116–119. DOI: http://dx.doi.org/10.1109/SAINTW.2003.1210138

P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. 2010. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *Proceedings 18th IEEE International Requirements Engineering Conference (RE)*. 95–103.

T. Stahl, M. Voelter, and K. Czarnecki. 2006. *Model-driven Software Development: Technology, Engineering, Management*. Wiley & Sons.

V. Stantchev and C. Schräpfer. 2009. Negotiating and enforcing QoS and SLAs in grid and cloud computing. In *Proceedings of the Advances in Grid and Pervasive Computing*. Vol. 5529. 25–35.

A. Van Deursen, P. Klint, and J. Visser. 2000. Domain-specific languages: An annotated bibliography. *Sigplan Not.* 35, 6 (2000), 26–36.

A. Van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. 2009. An adaptation framework enabling resource-efficient operation of software systems. In *Proceedings of the Warm Up Workshop for the ACM/IEEE International Conference on Software Engineering (ICSE'10)*.

J. Varia. 2011. Architecting for the cloud: Best practices. Technical Report. Amazon Web Services. https://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf.

T. Vogel and H. Giese. 2014. Model-driven engineering of self-adaptive software with EUREMA. *ACM Trans. Auton. Adapt. Syst.* 8, 4 (2014), 18.

S. Wätzoldt and H. Giese. 2014. Classifying distributed self-* Systems based on runtime models and their coupling. In *Proceedings of the 9th Workshop on Models@run.time*. 11–20. http://ceur-ws.org/Vol-1270/.

R. Wieringa and A. Moralı. 2012. Technical action research as a validation method in information systems design science. In *Proceedings of the International Conference on Design Science Research in Information Systems*. 220–238.

H. Xiong, F. Fowley, and C. Pahl. 2015. An architecture pattern for multi-cloud high availability and disaster recovery. In *Proceedings of the Workshop on Federated Cloud Networking (FedCloudNet'15)*.

L. Zhang, Y. Zhang, P. Jamshidi, L. Xu, and C. Pahl. 2014. Workload patterns for quality-driven dynamic cloud service configuration and auto-scaling. In *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing (UCC'14)*. 156–165.

O. Zimmermann. 2009. *An Architectural Decision Modeling Framework for Service Oriented Architecture Design*. Ph.D. Dissertation. Universität Stuttgart.

O. Zimmermann. 2016. Mircroservices tenets: Agile approach to service development and deployment. In *Proceedings of the Symposium/Summer School on Service-Oriented Computing*.

Received September 2016; revised May 2017; accepted May 2017