# Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications*

Robert Heinrich
Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
robert.heinrich@kit.edu

## ABSTRACT

Building software systems by composing third-party cloud services promises many benefits such as flexibility and scalability. Yet at the same time, it leads to major challenges like limited control of third party infrastructures and runtime changes which mostly cannot be foreseen during development. While previous research focused on automated adaptation, increased complexity and heterogeneity of cloud services as well as their limited observability, makes evident that we need to allow operators (humans) to engage in the adaptation process. Models are useful for involving humans and conducting analysis, e.g. for performance and privacy. During operation the systems often drifts away from its design-time models. Run-time models are kept insync with the underlying system. However, typical run-time models are close to an implementation level of abstraction which impedes understandability for humans.

In this vision paper, we present the iObserve approach to target aforementioned challenges while considering operation-level adaptation and development-level evolution as two mutual interwoven processes. Central to this perception is an architectural run-time model that is usable for automatized adaptation and is simultaneously comprehensible for humans during evolution. The run-time model builds upon a technology-independent monitoring approach. A correspondence model maintains the semantic relationships between monitoring outcomes and architecture models. As an umbrella a megamodel integrates design-time models, code generation, monitoring, and run-time model update. Currently, iObserve covers the monitoring and analysis phases of the MAPE control loop. We come up with a roadmap to include planning and execution activities in iObserve.

## Keywords

Architectural Run-time Model, Performance Model, Usage Profile, Palladio Component Model, Privacy

## 1. INTRODUCTION

Two major trends continue to shape the way software engineers build and maintain future long-living software systems [24]: (I) Software systems will be built by selecting, configuring, and composing third-party software-defined services, providing access to application software, cloud compute and storage facilities, internet-connected things, as well as data. Software-defined services separate ownership, maintenance and operation from usage of software. Service users do not need to deploy and run software on their own. They use software executed by third parties that can be remotely accessed through service interfaces [12]. (II) Software will be deployed on virtualized, remote compute infrastructures, such as cloud infrastructures, software-defined network devices, as well as Internet-of-Things devices. These deployment models result in software that will increasingly be deployed on hardware resources and on top of middleware that is owned and operated by third parties. The use of third party services and cloud infrastructures promises many benefits, such as flexibility, scalability, reusability and economic use of resources. Yet at the same time, it leads to major challenges (e.g., see [42]) such as those described hereafter.

Third party services are subject to changes at run-time in their execution environment not under the control of software engineers and, in most cases, cannot be foreseen during development. Examples include service changes, such as new versions, un-provisioning of existing services and fluctuations in quality characteristics like performance.

As a consequence of using software and infrastructure that is owned, hosted and maintained by third-parties, software engineers will have limited visibility. In most cases services can only be observed through scattered interfaces offered by the service provider. The architecture or code of the software service will not be known by engineers at the side of the service consumers and integrators. Also, the utilization and load distribution of the infrastructure in realistic application settings will not be published by the provider.

In addition to decentralizing functionality, future software system will also distribute and decentralize their data. As an example, data-intensive applications may require deploying the data analytics tasks to several compute nodes. As those nodes may be dynamically migrated and replicated, this can lead to privacy concerns when distributing data across different legal systems (i.e. geographical locations) with different regulations for privacy and data proliferation (cf. [55]). In the cloud context performance and privacy are closely interrelated. The application usage impacts on the application's performance. Continuously appraised elasticity rules trigger the migration and replication of cloud application's software components among geographically distributed data centers. Both, migration and replication, may increase performance yet lead to violation of privacy policies and increasing costs.

Increased complexity and heterogeneity of cloud services as well as their limited observability, will bring fully automatic adaptation to its limits [24]. For instance, business decisions, such as sales campaigns, lead to an intensive application usage which cannot be anticipated by the system by monitoring data alone. Fully capturing and formalizing the knowledge required to take autonomous adaptation decisions may just become infeasible or face prohibitive costs.

While, in recent years, the research focus for adaptation was on driving further its automation (e.g., see [2]), it becomes evident that we need to allow operators (humans) to engage in the adaptation process. An open question is how to facilitate such operator-in-the-loop adaptation.

While changes in cloud application requirements (such as supporting a new feature) provoke human software evolution activities, changes in cloud infrastructures (such as virtual machine migration, data replication) and variations in the application workloads may be addressed by the application in a self-adaptive way (e.g., see [8]). In our work, we understand evolution as a longer sequence of modifications to a software system over its life-time applied manually by software engineers (cf. [38]), while we understand (self-) adaptation to be single or a few related modifications by the system performed in an automated way (cf. [41, 49]).

This vision paper takes a model-based stance by proposing architectural run-time models as a means for combining automated adaptation as well as the human inspection of cloud services. While run-time models have shown their effectiveness for self-adaptation, using run-time models during software evolution and to put the operator in the loop has been neglected so far [26, 24]. As commonly observed, design-time models often drift away from the actual system [46]. In contrast, run-time models are kept in-sync with the underlying system. Thus run-time models may serve as valuable basis for evolution activities. However, typical run-time models are close to an implementation level of abstraction [61]. While being useful for self-adaptation, such low level of abstraction impedes understandability for humans. In addition, owing to various modifications during the life-time of a system, run-time models may grow in detail or become unnecessarily complex, which severely limits understandability of this kind of run-time models for humans during software evolution (e.g., see [62]).

The iObserve[1] approach targets aforementioned challenges of long-living, cloud-based software systems by the notion of architectural run-time models to facilitate the automated analysis, as well as the human inspection to detect, in particular, performance and privacy anomalies. Architectural run-time models are kept up-to-date via dynamic processing of monitoring data [21]. As an umbrella to integrate design-time models, code generation, monitoring, analysis, and run-time model update, iObserve introduces a concise megamodel. The megamodel reflects the relationships of models, meta-models and transformations [15]. Various preprocessed monitoring data are used to update the architectural run-time model based on relationships specified in the megamodel. Subsequently the updated model takes part in existing analysis of performance (e.g., [50]) and privacy (e.g., [55]) which may trigger adaptation or evolution activities. Contributions of iObserve so far focus on monitoring and analyzing cloud-based software applications. Next we list these contributions reported in the paper.

- Architectural run-time model, which reflects updates of component structures, deployments, and application usage caused by changes in the cloud application and its environment at run-time. In contrast to related work this architectural run-time model targets to be usable for automatized adaptation and is simultaneously comprehensible for humans during evolution.
- Instrumentation model, which facilitates model-driven instrumentation and monitoring of cloud applications and infrastructures. In contrast to related work this model provides a record structure that is independent of a specific monitoring framework and allows for the injection of monitoring probes in an application or infrastructure independent of its specific technology.
- Correspondence model, which is a novel approach to define the transition between low-level monitoring data and component-based architecture models. Thereby it maintains the semantic relationships between the system and the run-time models while keeping the models understandable for humans.
- Megamodel, which serves as an umbrella to integrate design-time architecture models, generator models, instrumentation models, correspondence models and run-time architecture models.
- Further, we propose a roadmap for extending iObserve to planning and execution of adaptation activities.

The remainder of the paper is structured as follows. Sec. 2 gives an overview of the iObserve approach. Sec. 3 discusses changes to cloud applications at run-time before we present our approach to run-time architecture modeling in Sec. 4. Sec. 5 exemplifies the application of iObserve. We describe the roadmap to further extensions of iObserve in Sec. 6. Sec. 7 discusses relate work. The paper concludes in Sec. 8.

## 2. OVERVIEW OF IOBSERVE

iObserve [26, 22] addresses the challenges of distributed cloud-based software systems by following the widely adopted MAPE (Monitor, Analyze, Plan, Execute) control loop model. MAPE is a feedback cycle for managing system adaptation [66]. iObserve extends the MAPE loop with shared models to ease the transition between adaptation and evolution. For adaptation, the state of the software system is determined by monitoring, analyzed to detect anomalies and predict deviations, used as input for planning the adaptation to mitigate anomalies, and finally to execute. Fig. 1 gives an overview of iObserve. The figure is inspired by Oreizy et al. [48]. Parts in the focus of this paper are marked blue. The cloud application life-cycle, underlying the iObserve approach, considers evolution and adaptation as two mutual, interwoven processes that influence each other [22]. The evolution activities are conducted by human developers, while the adaption activities are performed fully automatically by predefined strategies.

Central to this perception is an architectural run-time model that is usable for automatized adaptation and is simultaneously comprehensible for humans during evolution. iObserve builds upon a model driven engineering approach [22] that models the software architecture and deployment

[1] https://sdqweb.ipd.kit.edu/wiki/iObserve

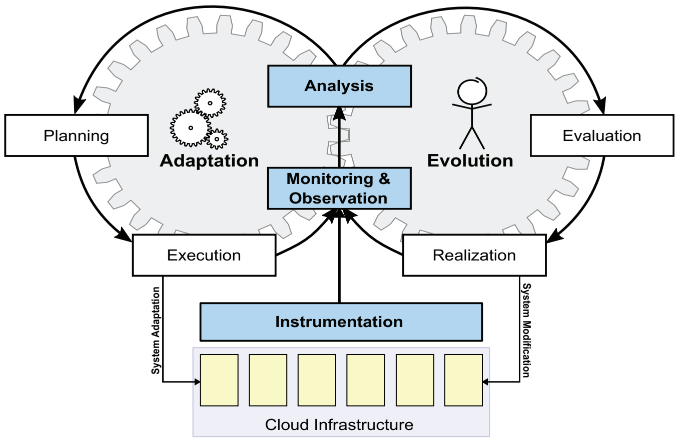

Figure 1: iObserve cloud application life-cycle: Considering adaptation and evolution as two interwoven processes.

in a component-oriented fashion and generates the artifacts to be executed during run-time. The cloud application is instrumented with monitoring probes to keep the run-time model causally connected with application and infrastructure. The run-time model is analyzed to determine problems and thus triggering adaptation. When adaptation cannot be done fully automatically, the human operators or developers will be involved.

## 3. CHANGES AT RUN-TIME

Cloud-based software systems are subject to a wide range of *run-time changes (C)*. This section lists run-time changes to software applications and their environment taken from literature [65, 18, 59, 6] and discusses how to observe them in a cloud context.

*Workload characterization changes (C1)*: The workload intensity faced by the application and the user behavior may change which may affect the system performance (cf. [33]). The amount of users concurrently using the software application (closed workload [50]), the users' arrival rate at the system (open workload [50]), and the invoked services are contained in observable user sessions [59].

The deployment of a software systems may change, e.g. to address performance issues by migration or replication of components, which, however, may cause violations to privacy constraints. These changes may either be triggered actively by an adaptation mechanism or can be performed autonomously by the observed system.

*Migration (C2)*: moves a software component from one execution context, e.g. an application server, to another. While there have been first attempts to move running code, in practice the transfer is realized by undeploying the component on one execution context, and creating a new instance of the same component type on another context [65]. If necessary internal state is also transfered. In consequence migration is in essence the combination of one undeployment and one deployment operation of a component of the same type on different execution contexts.

*Replication (C3)*: duplicates a running component instance in a way that the workload can be distributed among the deployed instances. At heart this operation is based on the deployment operation. If necessary some state must be copied or shared between the components. Replication can only be performed if the architecture allows for distributing requests among components.

*Dereplication (C4)*: is the inverse operation to C3. It is in essence the undeployment of a component instance which has been replicated before.

The changes C2 to C4 all rely on deployment and undeployment operations. However, solely based on the observation of deployment and undeployment events, it is not possible to detect this operations with total accuracy without further information. To be able to distinguish these types of changes, component instances must have unique identifiers. Based on these identifiers, a sequence of deployment and undeployment operations of the same component type with the same instance identifier can be clearly identified as migration. While replication creates a new instance with a new identifier of the same component type.

*(De)-allocation (C5/C6)*: appears when execution contexts become available (allocation), or disappear (de-allocation) [65]. The observation of both is highly technology dependent. However, three ways to observe (de)-allocation can be distinguished. First, the system controlling the allocation and de-allocation provides information to the monitoring system directly. Second, the allocation and de-allocation do not create observable events, e.g., when the cloud infrastructure service controller only accepts requests, but does not issue events. In that case the monitoring system must actively poll the controller for allocation and de-allocation information. And third, in systems where deployment and undeployment of component instances is handled autonomously, the allocation and de-allocation is often performed implicitly without creating distinct events. However, as a deployment always requires an existing execution context, such deployment events imply the necessary allocation event. The same applies to the undeployment of components.

## 4. RUN-TIME ARCHITECTURE MODELING IN IOBSERVE

In contrast to related run-time modeling approaches (discussed in Sec. 7), iObserve utilizes the same architecture meta-model at design-time and run-time to provide rich models at run-time which are comprehensible for developers and operators, and which can be fed back into software evolution without the need of conversion and the risk of loss of knowledge. We do not target the extraction of a completely new run-time model from monitoring data but update specific parts of an existing design-time model. Therefore, interlinking design-time and run-time artifacts on model and implementation level is required. iObserve updates the design-time model by single parameters (e.g., usage properties or deployment contexts) and also reflects structural changes (e.g., replicated components or new execution containers) to address aforementioned run-time changes. A megamodel (Sec. 4.1) covers all involved artifacts and their relationships. The three key elements of the megamodel are (i) a meta-model for architectural run-time models (Sec. 4.2), (ii) at the heart a model defining the correspondence between observation data and the architectural run-time model (Sec. 4.3), and (iii) model-driven monitoring (Sec. 4.4).

### 4.1 iObserve Megamodel

Megamodels provide a notation for the relationships of

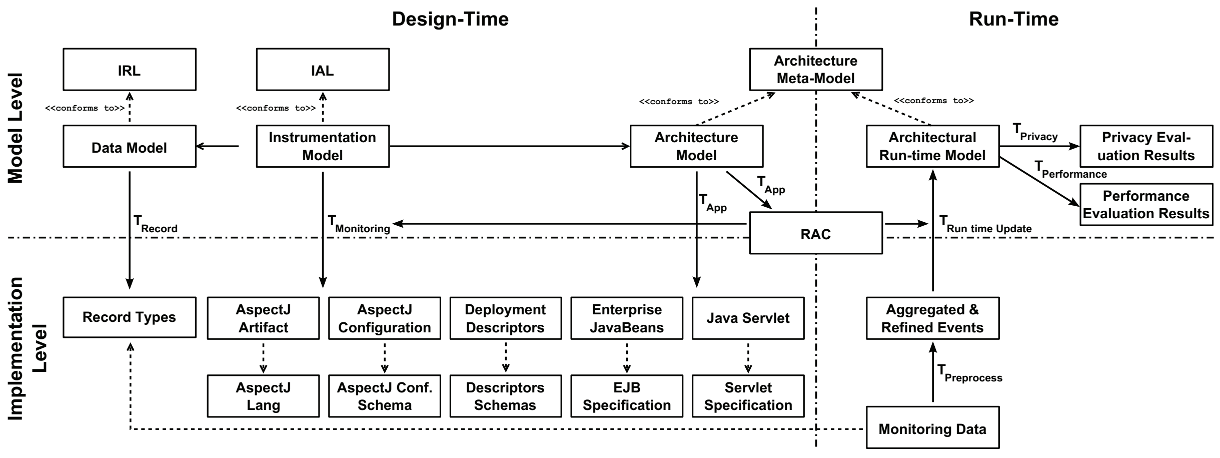

Figure 2: Overview of the iObserve megamodel in context of Java Enterprise Technology.

models, meta-models and transformations [15]. The iObserve megamodel depicted in Fig. 2 is divided into four sections defined by two dimensions: one for design-time vs. run-time, and one for model vs. implementation level. At design-time the megamodel shows how an architecture model is combined with our model-driven monitoring approach [32] and used for code generation. On the run-time side, monitoring data which relates to implementation artifacts is associated with architectural run-time model elements based on the run-time architecture correspondence model (RAC) [27].

The RAC is depicted in Fig. 2 below the architecture meta-model. It is the central element of the megamodel and crucial for the use of an architectural model at design-time and run-time. The RAC relates model nodes to implementation artifacts, like classes, methods, and XML files. It is initialized during code generation by the transformation $T_{App}$. The RAC is also used in the $T_{Monitoring}$ transformation for generation and configuration of probes by our model-driven monitoring approach [32]. Monitoring data can be described as an infinite model [11] and resembles a continuous data stream. As monitoring may produce large amounts of data (i.e. millions of events per second in large enterprise applications [16]), iObserve first filters and aggregates the monitoring data regarding the six types of run-time changes (cf. Sec. 3) using the $T_{Preprocess}$ transformation. Second, iObserve relates the implementation level monitoring data to architecture model elements using the semantic relationships in the RAC. Finally, iObserve applies the aggregated information to drive architectural run-time model update using the $T_{Run-time\ Update}$ transformation. The single parts of this approach are further detailed hereafter.

## 4.2 Architecture Meta-Models

Prospective architecture meta-models to be part of our megamodel must satisfy requirements *(R)* that result from the aforementioned run-time changes. It is important to recall at this point that, in the context of our research, we consider an architecture model already exists at design-time for doing predictions by probably making assumptions for information not available at design-time. It then becomes an architectural run-time model by updating certain parts of the model by observation data. Hence, combining design-time and run-time properties is straightforward since they rely on the same meta-model. *(R1)* For identifying C1, the architecture meta-model must reflect the application's usage profiles in terms of workload intensity and user behavior (e.g., services invoked by the users, paths the users traverse). *(R2)* The architecture meta-model must reflect the structure of the application and its environment in a component-based fashion to analyze the effect of reconfigurations (i.e. C2 to C6) and to ensure comprehensibility by humans during software evolution. *(R3)* Prospective architecture meta-models must allow for analyzing performance and privacy. This means quality-relevant properties of the software components and their context (e.g., branch propabilities in the usage profile, geo-location and processing rate of resources in the execution environment) must be represented in the architecture meta-model. Furthermore, associated analyzers for performance and privacy must be available.

Next we discuss how various candidates of meta-models satisfy the requirements. Layered Queueing Networks (LQNs) [51] and Queueing Petri Nets (QPNs) [3] are established prediction formalisms for software systems (e.g., [34]). They allow for conducting performance predictions based on system usage profiles (partly satisfy R1) and performance-relevant properties of the environment (partly satisfy R3). Yet, since they are general-purpose formalisms, they do not provide the specific modeling constructs for representing component-based software architectures. Thus, they do not satisfy R2 and are inadequate for analyzing the geo-location of components for privacy audits.

Meta-models for modeling software components, e.g. in the UML or Enterprise JavaBeans context, typically lack modeling of usage profiles (do not satisfy R1) and quality-relevant properties (do not satisfy R3). Some of these meta-models allow for quality annotations, e.g. UML SPT profiles [47]. However, they do not come along with analyzers for performance or privacy.

The Palladio approach [50] is tailored to component-based software architecture analysis. It relies on a comprehensive domain-specific meta-model – the Palladio Component Model (PCM). The PCM consists of several partial meta-models tailored to represent different aspects of a software system such as usage profile (satisfies R1), component structure (satisfies R2), deployment context, and execution environment as well as corresponding quality properties (satisfies

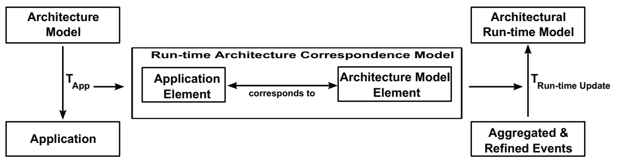
Architecture Model
Application
T_App
Run-time Architecture Correspondence Model
Application Element
corresponds to
Architecture Model Element
Architectural Run-time Model
T_Run-time Update
Aggregated & Refined Events

Figure 3: Overview of the Run-time Architecture Correspondence Model within the iObserve megamodel.

R3). Furthermore, the PCM is comprehensible to humans as the study in [39] indicates. There are several meta-models related to the PCM, such as the Descartes Meta-Model [7], and those surveyed by Koziolek [35]. These models have in common that they represent a detailed architecture specification in a component-oriented fashion. They are parameterized to explicitly capture the influences of the components' execution context, such as usage profile and hardware configuration [7]. In iObserve, we choose the PCM as a representative of these component-based meta-models, as it is established in the community and offers the most matured tool support. The PCM is applied as an architecture meta-model (see Fig. 2). The PCM provides all the modeling constructs to fulfill the aforementioned requirements [50] except for geo-location. However, it is straightforward to support geo-location by adding an attribute to execution environment model elements.

## 4.3 Run-time Architecture Correspondence

The record types model within the iObserve megamodel (lower left side of Fig. 2) exhibits a flat (i.e. non-hierarchical) structure where all records are contained in a large collection [11] and distinguished only by their type and their attributes. Monitoring events adhering to this collection reflect code artifacts which correspond to elements of the architecture model. Knowledge about this correspondence is lodged with the RAC depicted in Fig. 3.

The RAC bridges the divergent levels of abstraction between application monitoring outcomes on implementation level and component-based architecture models. It keeps the mapping between one or more source code artifacts (i.e. application elements) and elements of the architecture model. It provides these correspondence information for updating the run-time architecture model based on observations. An example is given in Sec. 5.

During the initial application development code is generated from the architecture model ($T_{App}$). While generating the code, the semantic relationships between the generated artifacts (e.g., Java classes) and the architecture model elements (e.g., logical components) are automatically recorded and stored in the RAC.

Once deployed, the application and the entire cloud system face various changes at run-time (cf. Sec. 3). These changes require the initial architecture model to be updated to continuously reflect the current system state at run-time. The model is updated by single parameters (e.g., usage properties or deployment contexts of migrated components) and structural elements (e.g., replicated components or new execution containers). While updating the model using the transformation $T_{Run-time\ Update}$ the RAC is applied to identify model elements to be modified, replicated or removed. Therefore, the RAC establishes the correspondence between monitored events at implementation level and the architecture model (e.g., between objects arose from Java classes and logical components in the model). Larger changes to the system architecture trigger evolution activities which require a human developer to change the architecture models. Subsequently, code is regenerated and recorded relationships in the RAC are updated.

Updating the architecture model by implementation level observations must not deviate its component-based fashion and, thus, its usefulness for humans during long-term evolution. In iObserve, the level of abstraction of the initial architecture model and the updated model (i.e. the architectural run-time model) is maintained, due to (a) both, the initial architecture model and the architectural run-time model, rely on the same meta-model, and (b) the decomposition of a design model element in one or more source code artifacts is recorded in the RAC while code generation and (c) restored while transforming monitoring events related to the source code artifacts to the component-based architectural run-time model. Thereby, identity is ensured by unique identifiers of the elements recorded in the RAC. The level of abstraction of the initial model does not affect the mapping in the RAC. Therefore, in analogy to existing component models, we do not predetermine the abstraction level used in the architecture model. Consequently, owing to the correspondence between model and code specified in the RAC, the abstraction level of the model cannot deviate from one update to another.

## 4.4 Model-driven Monitoring

For gathering information required to update architectural run-time models with respect to usage and deployment, the monitoring approach of iObserve integrates several instrumentation and monitoring technologies. As this requires different types and procedures to realize monitoring, we provide a model-based abstraction of instrumentation probes and data collection [26].

Monitoring is a run-time activity used to observe various system properties. Whereas the specification of what, how, and where to monitor is conducted at design-time. In iObserve, instrumentation probes are injected into the code while generation. Monitoring is perceived as a cross-cutting concern and realized as an aspect by means of aspect-oriented modeling [14]. Fig. 4 shows the point cuts used to express the locations of probes. Required information about the relationship of architecture model elements and their implementation is expressed in the RAC. Furthermore, the technology used to the probe realization is determined by a look up in the RAC to ensure the correct selection of probe generator or implementation. For example, the transformation requires information whether the advice must realize an interface of a JEE interceptor or generate an AspectJ artifact.

The definition of the monitoring aspect is realized through three models for the point cuts, the advices, and the data model for the observed information. The data model conforms to the Instrumentation Record Language (IRL) [32] to provide an implementation independent record specification which can be used across different technologies and programming languages. This record specification is realized by record types. The specification of point cuts and advices form an Instrumentation Aspect Model that conforms to the Instrumentation Aspect Language (IAL) [32]. Based on the

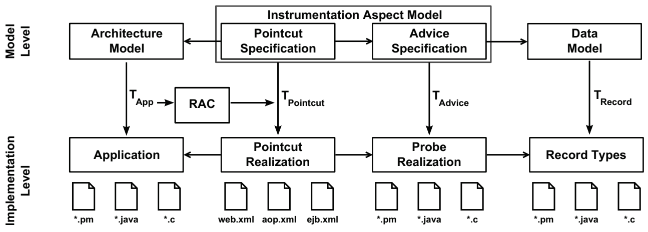

Figure 4: Overview of the models, transformations and code artifacts in the model-driven monitoring approach.

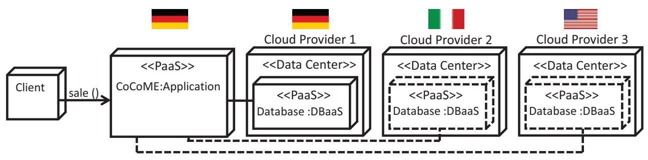

Figure 5: Actual (solid line) and conceivable (dashed line) component deployments within a global reach of prospective cloud providers in the experiment scenario.

models, transformations are used to generate the relevant implementation artifacts. As the IAL also supports native advices, which have been implemented by hand, not all of the advices must be generated supporting the possibility to use existing advices or include advices utilizing technologies not supported by the transformations.

## 5. FEASIBILITY EXAMPLE

This section gives a concrete application example of iObserve to demonstrate its feasibility for addressing challenges of modern cloud-based systems by handling run-time changes.

### 5.1 Application Scenario

The application example builds upon the established commuity case study CoCoME [53] and an associated evolution scenario [23]. CoCoME resembles a trading system as it may be applied in a supermarket chain. It implements processes at a single cash desk as well as enterprise-wide administrative tasks. CoCoME uses a database service hosted on data centers that are distributed around the globe, as shown in Fig. 5. The figure illustrates the CoCoME core application and the global reach of prospective cloud providers that offer Database-as-a-Service (DBaaS). Conceivable component deployments are illustrated by dashed lines. The supermarket chain is located within the European Union (EU). Thus, common privacy standards of the EU must be followed. According to these privacy standards sensitive data must not leave the EU during component migration or replication due to high risk of data leakage in other countries, e.g. the USA.

An advertisement campaign of the supermarket chain leads to an increased amount of sales and thus to variations in the application's usage profile (C1). Increased usage intensity causes an upcoming performance bottleneck due to limited capacities in the given service offering of the cloud provider currently hosting the database. Migrating (C2) or replicating (C3) the database from one data center to another may solve the scalability issues, however, may cause privacy issues [23]. Thus, the scenario addresses aforementioned challenges of decentralized data, limited visibility, and fluctuation in quality not foreseen during development.

The application example is realized with Java enterprise technologies (JEE), i.e. Enterprise JavaBeans and Java Servlets. It is deployed on the application server Glassfish which is compliant to JEE and widely used in productive settings such as in OpenStack cloud infrastructures. According to the challenge of limited visibility we assume in this scenario the platform service is rented. Thus, we cannot instrument Glassfish or use its internal monitoring system. We can only use probes inserted in our EAR bundle containing CoCoME which we can deploy over a service interface. The system utilizes a remote database node realized as a PostgreSQL database which provides appropriate features for migration and replication. Both nodes use a VirtualBox cloud image executed on OpenStack cloud infrastructures. As technical basis for monitoring, we choose the fast and reliable Kieker framework [28].

### 5.2 Applying iObserve

In the following we exemplify the filtering and processing of a sales operation (i.e. sale() depicted in Fig. 5) as well as the migration and replication of a database service. We assume code has been generated initially and the correspondences between model elements and source code artifacts has been stored in the RAC. Further, the application is deployed and running.

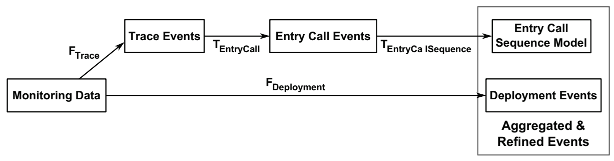

Figure 6: Excerpt of the transformation chain for monitoring data in $T_{Preprocess}$.

Fig. 6 illustrates the inner structure of the $T_{Preprocess}$ transformation shown in Fig. 2. While observing the running system Kieker produces a stream of heterogeneous monitoring events. Increased usage intensity in the scenario provokes changes in the model's workload specification (C1). iObserve filters out single entry and exit events to operations of the application (e.g., the sales operation) and aggregates them to sequences of events. Based on the sequences the new usage intensity is calculated and then transformed to the PCM workload specification.

The selection of entry and exit events from the stream of heterogeneous monitoring events is based on constraints kept in the filter $F_{Trace}$. Applying Kieker, monitoring data of the type BeforeOperationEvent are considered as entry level operations only if their related TraceMetadata has the value null in its parentTraceId attribute. Since this is the case for the sales operation it is considered as an entry level operation for further processing.

The $T_{EntryCall}$ transformation listens to the event stream of entry level operations and creates an entry call event for the sales operation. Based on user session information contained in the monitoring events $T_{EntryCallSequence}$ aggregates the sales operation together with other entry calls (e.g., for login). All observed user sessions are combined in a graph based entry call sequence model which is input to the $T_{Run-time\ Update}$ transformation shown in Fig. 2. Usage-related properties like path probabilities or usage intensity are calculated based on the sequence model.

$T_{Run-time\ Update}$ applies transformations according to the architecture meta-model (here the PCM) to update the architectural run-time model. As in the scenario we observe increased invocations to the sales operation, $T_{Run-time\ Update}$ modifies the model by an increased workload specification. As there might be various usage profiles in the model, the RAC is required for the transformation to identify which workload specification to be updated for the sales operation.

The updated model is then applied for performance simulation in Palladio (cf. [50]) which reveals an upcoming performance bottleneck. According to the scenario a planning routine automatically adapts the system by migrating (C2) or replicating (C3) the database service to another OpenStack instance located in the USA to address the performance issue. This evokes deployment and undeployment events observed by Kieker. These events among others contain information about associated classes. $T_{Preprocess}$ filters out deployment and undeployment events ($F_{Deployment}$ in Fig. 6). Identifiers (cf. Sec. 3) are used to detect migration or replication in the stream of events. $T_{Run-time\ Update}$ is applied to update deployment contexts (migration) of the database service components or copy the components (replication) in the PCM instance. Therefore, the RAC refers to components in the model corresponding to the observed events. Afterwards privacy checks (e.g., [55]) are applied which recognize privacy violations as the database has been moved outside the EU.

As exemplified, iObserve is able to cope with limited visibility of third party services, performance fluctuation, and supports the identification of privacy issues caused by decentralized data. More intelligent routines discussed in Sec. 6 may avoid the privacy issue by privacy-driven planning before migration or replication.

## 6. ROADMAP TO PLANNING AND EXECUTION PHASES

iObserve so far is focused on monitoring and analysis of cloud-based software applications. Addressing planning and execution phases of the MAPE control loop requires facing various challenges due to the complex and heterogeneous nature of cloud services. While different directions of further development are possible, we consider the following three pillars taken from [24] as next steps for evolving iObserve.

First, the global reach of cloud services requires putting special emphasis on privacy-driven planning. Second, the complexity of cloud services and the complexity of effects of external events lead to major challenges in finding an appropriate system design. Therefore, design space exploration may target performance and privacy based on architectural run-time models to be explored for solutions. Third, restrictions in fully automatic adaptation need to allow human operators to engage in the adaptation process. Operator-in-the-loop adaptation may overcome limitations of automatic adaptation. For all three pillars architectural run-time models (either used descriptively or prescriptively) are key artifacts as depicted in Fig. 7 [24].

According to the figure, if any performance or privacy issues are identified in the analysis phase, adaptation candidates are generated and specified as candidate prescriptive run-time models during the planning phase. Due to the different nature of performance and privacy, planning is performed by distinct techniques. The generated candidates

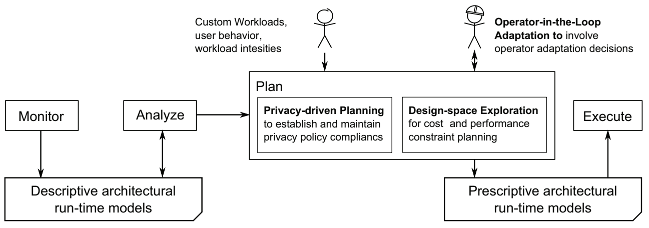
Custom Workloads,
user behavior,
workload intesities
Operator-in-the-Loop Adaptation to involve operator adaptation decisions
Plan
Privacy-driven Planning to establish and maintain privacy policy compliancs
Design-space Exploration for cost and performance constraint planning
Monitor
Analyze
Execute
Descriptive architectural run-time models
Prescriptive architectural run-time models

Figure 7: Illustration of the planning and execution phases in context of the MAPE control loop.

aim for solving the root cause that led to the raised issues. As part of the design space exploration, candidate models are evaluated and ranked (e.g., with respect to performance) building upon existing techniques [1]. Subsequently, these models are operationalized by deriving concrete tasks of an adaptation plan for the execution phase [52]. When selecting one concrete model among the candidates or deriving an adaptation plan cannot be done fully automatically, the human operator (cf. Fig 7) choses among the presented adaptation alternatives. In cases where no candidate could be created, e.g., due to lack of information or criticality of decision, the operator will also be involved. Realizing this approach requires tackling the following areas.

*Privacy-driven planning* requires techniques for determining degrees of freedom for adaptation that allow to resolve or mitigate privacy violations. They are defined around three key facets: (i) The identification and specification of appropriate cloud adaptation mechanisms which need to be included in the descriptive architectural run-time models in order to empower the adaptation routines for carrying out privacy-aware adaptations. (ii) Constraint-based generation of adaptation alternatives under consideration of the available mechanisms. And (iii) impact analysis of privacy related adaptations on multiple application characteristics.

Many cloud-based applications serve thousands if not millions of users whose behavior cannot be fully understood at design-time, as it varies over time, e.g. in case of sales campaigns. Furthermore, service offers and cloud products change over time in performance and cost. In combination with privacy concerns, this yields a complex problem space where multiple solutions can be computed forming a design space. *Design space exploration* is based on multiple criteria representing different dimensions for architectural adaptations of cloud applications during design-time and exploit this specification for generating and evaluating design alternatives during run-time. First, to determine the different degrees of freedom in the design-space, several input models are required beside the descriptive architectural run-time model, including cloud profiles and workload characterization models. The behavior model generation in iObserve will be extended to support behavior mixes based on different detected behavior patterns. Second, adaptation plans are translated into detailed adaptation tasks realizing the transformation of the present architecture into the target architecture. For task identification we build upon architecture-based assessment techniques [52].

*Operator-in-the-loop adaptation* can be split into two main activities. First, the introduction of information on external events, and second, the selection of adaptation plans from the design-space exploration. External parties come up with external events, such as sales campaigns, which cannot be foreseen at design-time. Therefore, the operator must be able to introduce new cloud profiles, data handling policies, and workload models at run-time. As workload models are complex, operators want to base them on observed behavior. Hence, iObserve will provide the ability to extract and modify intensities and behavior pattern. The design-space exploration might provide multiple adaptation candidates all representing a different element of a Pareto optimum. Human operators must assess and select adaptation plans which is supported by views and analyses. Therefore, we investigate the creation of views on the system that provide operators with up to date information about the actual cloud application derived from the architectural run-time models of iObserve, the different adaptation candidates, and the associated adaptation tasks.

## 7. RELATED WORK

While software adaptation and evolution is an architectural challenge [36] existing approaches dealing with the interplay between adaptation and evolution (e.g., [45, 48]) lack continuous modeling and updating of software architectures in component-based fashion and the engagement of human operators. Work related to iObserve can be distinguished into four major categories. First, work related to our architectural run-time models which can be separated in approaches for reusing design-time models at run-time and approaches for model extraction. Second, privacy-driven planning which is tackled from two different angles in related work – privacy-by-design and privacy audits. Third, approaches to design space exploration. Fourth, work on operator-in-the-loop adaptation.

Work on *reusing design-time models during run-time* (e.g., [43, 30, 9]) employs design-time models as foundation for reflecting software systems during run-time. [4] gives an overview of run-time modeling and analysis approaches. The work in [43] reuses sequence diagrams created during run-time to verify running applications against their specifications. However, the approach does not include any updating mechanisms that changes the model whenever the reflected systems is being alternated. Consequently, run-time changes are not supported. Other than this, the run-time models in [30, 9] are modified during run-time. These approaches employ workflow specifications created during design-time in order to carry out performance and reliability analyses during run-time. The approaches update the workflow models with respect to quality properties (e.g., response times) of the services bound to the workflow. However, these approaches do not reflect component-based software architectures. Further, this work updates the model with respect to single parameters and does not change the model's structure, which is required to reflect, e.g., C3 and C4.

Work on *model extraction* creates and updates model content during run-time. Approaches such as [56, 54, 58] establish the semantic relationships between executed applications and run-time models based on monitoring events (for a comprehensive list of approaches see [57]). Starting with a "blank" model, these approaches create model content during run-time from scratch, e.g. by observing and interpreting operation traces. Therefore, they disregard information that cannot be gathered from monitoring data, such as design perspectives on component structures and component boundaries. For instance, the work in [58] exploits process mining techniques for extracting state machine models from event logs. Without knowledge about the component structure developed during design-time, the extracted states cannot be mapped to the initial application architecture. In consequence, the model hierarchy is flat and unstructured, which hinders software developers and operators in understanding the current situation of the application at hand. Further, the work reflects processes but neither components nor their relationships (cf. C2-4). Other than this, the work in [54] extracts components and their relationships from observations for architecture comparison. With this approach we share the application of transformation rules to update a run-time model based on monitoring events. The resulting model in [54] is coarse-grained, which is sufficient for their purposes. However, when conducting performance and privacy analyses the observation and reflection of resource consumptions is crucial. Reflecting the consumption by the means of usage profiles requires processing event sets rather than single events, which outruns the capacity of this approach (cf. C1). Further, the observation and analysis of usage and component changes causes complex relationships between the investigated applications, probe types, and run-time models, which is not discussed in [54].

To summarize, design-time models reused at run-time provide good comprehensibility to humans, but are not updated with respect to structural changes yet. However, structural updates are required to reflect the run-time changes listed in Sec. 3. Work on model extraction automatically creates run-time models from scratch. As design-time decisions on application architectures cannot be fully derived from monitoring events the resulting models lack understandability.

Related work on *privacy-by-design* employs two main concepts – access control and deployment resp. elasticity rules. Research on access control mechanisms (e.g., [13]) investigates how to equip components with mechanisms that permit or grant data access. However, access control mechanisms face difficulties whenever the controlled component is migrated (C2) or replicated (C3) across data centers. Rules defined during design-time may not accurately capture geo-location constraints at run-time as they cannot consider the actual data storage and transfer.

Approaches for *privacy audits* of cloud services (e.g., [20]) use host geo-location and data possession to evaluate the compliance of privacy regulations. Host geo-location can be checked based on ping round trip times for a service interface utilizing different ping origins. However, the components behind the interfaces might be migrated or replicated. [55] proposes fundamentals for facilitating privacy-driven planning. Yet, existing work focuses on the identification of privacy violations and does not cover planning or adaptation aspects. Quality aware planning techniques exploit the proportional relation between allocated hardware and performance. For instance, calibrated run-time models [29] are used to steer planning activities and adjust hardware nodes to reach the desired performance. However, resolving privacy violations is inherently different from resolving performance issues.

To summarize, related work on privacy-driven-planning neglects two essential characteristics of cloud applications, (1) the dynamic migration and replication of cloud application components (cloud elasticity) across data centers; (2) the limited control and visibility of cloud elasticity by cloud customers (shared ownership).

*Design space exploration* relies on various aspects of the design space in model form, e.g. workload characterization and cloud profiles, to be explored for solutions. Workload characterization approaches used for adaptive systems and workload generation [64] focus on the construction of workload intensity patterns. However, the intensity alone is not sufficient. Therefore, a wide variety of methods are used to model user behavior and workloads (e.g., [40, 60]). Current work addresses complex user behavior [63] and the business impact on system usage [25]. However, the discrimination of different user behaviors and the aggregation of similar behaviors is complicated. The BEAR approach [19] addresses the construction and analysis of such behavior models. However, BEAR focuses on analysis at design-time and its user distinction based on IP addresses is not unique. Cloud profiles describe properties of cloud services including resources and cost, like in the CloudMIG approach [18]. Based on these models, design space exploration can be performed. An overview of design space exploration methods is given in [1]. CDOXplorer [17] assesses the fitness of cloud development options based on cloud profiles. To summarize, related work on design space exploration neglects the differentiation of user behavior and trade-off decisions between different quality attributes.

To the best of our knowledge, the combination of automatic and *operator-in-the-loop adaptation* is not covered in related work on self-adaptive systems as the main goal was to automate adaptation and eliminating human involvement. Other research areas however came up with related approaches. Control systems [5] involve humans in the operation of large scale distributed industrial plants and infrastructure networks. Recommendation systems [44] derive knowledge from source code or issue trackers to support developers in architecture comprehension [37], software migration [18], and performance optimizations [10]. CloudAdvisor [31] and CloudMIG [18] support multi-objective optimization. ExplorViz [16] provides 3D architecture visualization based on monitoring data. To summarize, while we could not find combinations of automatic and operator-in-the-loop adaptation, control and recommendation systems provide foundations to involve humans in the control loop.

## 8. CONCLUSION AND FUTURE WORK

This paper presented the iObserve approach to support adaptation and evolution of cloud-based software applications. The approach takes a model-based stance by proposing architectural run-time models for combining automated adaptation and human inspection. As an umbrella to integrate design-time models, code generation, monitoring, analysis, and run-time model update, we proposed a concise megamodel. An exemplary application of iObserve indicates its feasibility to handle run-time changes. Currently, iObserve supports the monitoring and analysis phases of the MAPE control loop model. We come up with a roadmap to consider planning and execution activities in the future.

Besides following the roadmap we plan to extend the evaluation of iObserve. As a next step we will conduct further experiments for evaluating the architectural run-time models of iObserve with respect to fidelity [4] and usefulness for human inspection and extend first scalability experiments.

## 9. REFERENCES

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *Software Engineering, IEEE Transactions on*, 39(5):658–683, 2013.

[2] B. H. C. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, pages 1–26. Springer, 2009.

[3] F. Bause. Queueing petri nets - a formalism for the combined qualitative and quantitative analysis of systems. In *5th Int'l Workshop on Petri nets and Performance Models*, pages 14–23. IEEE, 1993.

[4] N. Bencomo, R. France, B. H. C. Cheng, and U. Amann. *Models@run.time*. Springer, 2014.

[5] S. A. Boyer. *Scada: Supervisory Control And Data Acquisition*. International Society of Automation, 2009.

[6] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In P. Alexander, C. S. Pasareanu, and J. G. Hosking, editors, *ASE*, pages 183–192. IEEE, 2011.

[7] F. Brosig, N. Huber, and S. Kounev. Modeling parameter and context dependencies in online architecture-level performance models. In *15th Symposium on Component Based Software Engineering*, CBSE '12, pages 3–12. ACM, 2012.

[8] A. Bucchiarone, C. Cappiello, E. Di Nitto, S. Gorlatch, D. Mailänder, and A. Metzger. Design for self-adaptation in service-oriented systems in the cloud. In D. Petcu and J. Vásquez-Poletti, editors, *European Research Activities in Cloud Computing*. Cambridge Scholars Publishing, 2012.

[9] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. A framework for QoS-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, 2008.

[10] C. Chambers and C. Scaffidi. Impact and utility of smell-driven performance tuning for end-user programmers. *Journal of Visual Languages & Computing*, 28(0):176 – 194, 2015.

[11] B. Combemale, X. Thirioux, and B. Baudry. Formally defining and iterating infinite models. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS*, pages 119–133. Springer, 2012.

[12] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 2008.

[13] U. e Ghazia, R. Masood, and M. Shibli. Comparative Analysis of Access Control Systems on Cloud. In *13th Int'l Conference on Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing*, pages 41–46, 2012.

[14] T. Elrad, O. Aldawud, and A. Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *LNCS*, pages 189–201. Springer, 2002.

[15] J.-M. Favre. Foundations of model (driven) (reverse) engineering – episode i: Story of the fidus papyrus and the solarus. In *Dagstuhl post-proccedings*, 2004.

[16] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: Visual runtime behavior analysis of enterprise application landscapes. In *23rd Europ. Conference on Information Systems*. AIS, 2015.

[17] S. Frey, F. Fittkau, and W. Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *35th Int'l Conference on Software Engineering*, pages 512–521. IEEE Press, 2013.

[18] S. Frey and W. Hasselbring. The CloudMIG approach: Model-based migration of software systems to cloud-optimized applications. *Int'l Journal on Advances in Software*, 4(3 and 4):342–353, 2011.

[19] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *36th Int'l Conference on Software Engineering*, 2014.

[20] M. Gondree and Z. N. Peterson. Geolocation of data in the cloud. In *3rd conference on Data and application security and privacy*, pages 25–36. ACM, 2013.

[21] W. Hasselbring. Reverse engineering of dependency graphs via dynamic analysis. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*, pages 5:1–5:2. ACM, 2011.

[22] W. Hasselbring, R. Heinrich, R. Jung, A. Metzger, K. Pohl, R. Reussner, and E. Schmieders. iObserve: integrated observation and modeling techniques to support adaptation and evolution of software systems. Technical Report 1309, Kiel University, Kiel, Germany, 2013.

[23] R. Heinrich, S. Gärtner, T.-M. Hesse, T. Ruhroth, R. Reussner,

K. Schneider, B. Paech, and J. Jürjens. A platform for empirical research on information system evolution. In *27th Int'l Conference on Software Engineering and Knowledge Engineering*, pages 415–420. KSI Research Inc., 2015.

[24] R. Heinrich, R. Jung, E. Schmieders, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl. Architectural run-time models for operator-in-the-loop adaptation of cloud applications. In *IEEE 9th Symposium on the Maintenance and Evolution of Service-Oriented Systems and Cloud-Based Environments*. IEEE, 2015.

[25] R. Heinrich, P. Merkle, J. Henss, and B. Paech. Integrating Business Process Simulation and Information System Simulation for Performance Prediction. *Intl. Journal on Software & Systems Modeling*, 2015.

[26] R. Heinrich, E. Schmieders, R. Jung, W. Hasselbring, A. Metzger, K. Pohl, and R. Reussner. Run-time architecture models for dynamic adaptation and evolution of cloud applications. Technical Report 1503, Kiel University, Kiel, Germany, 2015.

[27] R. Heinrich, E. Schmieders, R. Jung, K. Rostami, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl. Integrating run-time observations and design component models for cloud system analysis. In *9th Int'l Workshop on Models@run.time*, pages 41–46. CEUR Vol-1270, 2014.

[28] A. v. Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *3rd Int'l Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, 2012.

[29] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev. S/T/A: Meta-modeling Run-time Adaptation in Component-Based System Architectures. In *9th Int'l Conference on e-Business Engineering*, pages 70–77. IEEE, 2012.

[30] D. Ivanovic, M. Carro, and M. Hermenegildo. Constraint-based runtime prediction of sla violations in service orchestrations. In *Service-Oriented Computing*, pages 62–76. Springer, 2011.

[31] G. Jung, T. Mukherjee, S. Kunde, H. Kim, N. Sharma, and F. Goetz. Cloudadvisor: A recommendation-as-a-service platform for cloud configuration and pricing. In *IEEE 9th World Congress on Services*, pages 456–463, 2013.

[32] R. Jung, R. Heinrich, and E. Schmieders. Model-driven instrumentation with Kieker and Palladio to forecast dynamic applications. In *Symposium on Software Performance*, pages 99–108. CEUR Vol-1083, 2013.

[33] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium*, pages 1287–1294. IEEE, 2012.

[34] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *Transactions on Software Engineering*, 32(7):486–502, 2006.

[35] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010.

[36] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering*, pages 259–268, 2007.

[37] S. Lee, S. Kang, S. Kim, and M. Staats. The impact of view histories on edit recommendations. *Software Engineering, IEEE Transactions on*, 41(3):314–330, 2015.

[38] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., 1985.

[39] A. Martens, H. Koziolek, L. Prechelt, and R. Reussner. From monolithic to component-based performance evaluation of software architectures. *Empirical Software Engineering*, 16(5):587–622, 2011.

[40] D. A. Menasce and V. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, 2001.

[41] A. Metzger and E. Di Nitto. Addressing highly dynamic changes in service-oriented systems: Towards agile evolution and adaptation. In *Agile and Lean Service-Oriented Development: Foundations, Theory and Practice*. IGI Global, 2012.

[42] A. Metzger (Ed.). Software engineering: Key enabler for innovation. NESSI White Paper, 2014.

[43] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@run.time to support dynamic adaptation. *IEEE Computer*, 42(10):44–51, 2009.

[44] M.P. Robillard. Recommendation systems for software engineering. *Software, IEEE*, 27(4):80–86, 2010.

[45] H. Müller and N. Villegas. Runtime evolution of highly dynamic software. In *Evolving Software Systems*, pages 229–264. Springer, 2014.

[46] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[47] Object Management Group. UML profile for schedulability, performance, and time specification. Technical report, 2005.

[48] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: Framework, approaches, and styles. In *Companion of the 30th Int'l Conference on Software Engineering*, pages 899–910. ACM, 2008.

[49] M. Papazoglou, K. Pohl, M. Parkin, and A. Metzger, editors. *Service Research Challenges and Solutions for the Future Internet: S-Cube – Towards Mechanisms and Methods for Engineering, Managing, and Adapting Service-Based Systems*, volume 6500 of *LNCS*. Springer, 2010.

[50] Reussner, Ralf H. et al., editor. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. ISBN: 978-0-262-03476-0.

[51] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, 1995.

[52] K. Rostami, J. Stammel, R. Heinrich, and R. Reussner. Architecture-based assessment and planning of change requests. In *11th Int'l Conference on Quality of Software Architectures*, pages 21–30. ACM, 2015.

[53] S. Herold et al. CoCoME – the common component modeling example. In *The Common Component Modeling Example*, pages 16–53. Springer, 2008.

[54] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.

[55] E. Schmieders, A. Metzger, and K. Pohl. Runtime model-based privacy checks of big data cloud services. In *13th Int'l Conference on Service Oriented Computing*. Springer, 2015.

[56] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723, 2011.

[57] M. Szvetits and U. Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *SoSyM*, 2013.

[58] W. van der Aalst, M. Schonenberg, and M. Song. Time prediction based on process mining. *Information Systems*, 36(2):450–475, 2011.

[59] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In *SPEC Int'l Performance Evaluation Workshop*, LNCS, pages 124–143. Springer, 2008.

[60] A. van Hoorn, C. Vögele, E. Schulz, W. Hasselbring, and H. Krcmar. Automatic extraction of probabilistic workload specifications for load testing session-based application systems. In *8th Int'l Conference on Performance Evaluation Methodologies and Tools*, pages 139–146. ICST, 2014.

[61] T. Vogel and H. Giese. Adaptation and abstract runtime models. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 39–48. ACM, 2010.

[62] T. Vogel and H. Giese. On unifying development models and runtime models (position paper). In *9th Int'l Workshop on Models at run.time*. CEUR, 2014.

[63] C. Vögele, R. Heinrich, R. Heilein, H. Krcmar, and A. van Hoorn. Modeling complex user behavior with the palladio component model. In *Symposium on Software Performance*, 2015. accepted, to appear.

[64] J. von Kistowski, N. R. Herbst, D. Zoller, S. Kounev, and A. Hotho. Modeling and Extracting Load Intensity Profiles. In *10th Int'l Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015.

[65] R. von Massow, A. van Hoorn, and W. Hasselbring. Performance simulation of runtime reconfigurable component-based software architectures. In *ECSA*, volume 6903 of *LNCS*, pages 43–58. Springer, 2011.

[66] Y. Brun et al. Software engineering for self-adaptive systems. chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer, 2009.