Architectural Run-time Models to Facilitate Quality-aware DevOps *

Robert Heinrich
Karlsruhe Institute of Technology (KIT), heinrich@kit.edu

1 Architectural Models Differ in Development and Operations

Building software applications by composing cloud services promises many benefits such as flexibility and scalability. However, it leads to major challenges like increased complexity, fragility and changes during operations that are unknown in development phase. These rapidly changing applications require communication and collaboration between software developers and operators as well as strong integration of building, evolving and adaptation activities.

DevOps is a set of practices to enable software developers and operators to work more closely and thus reduce the time between changing a system and putting the change into production, while ensuring high quality [6]. The software application architecture is a central artifact at the interface between development and operations. The phase-spanning consideration of software architecture is essential in DevOps practices. New architectural styles such as microservices are proclaimed to satisfy requirements like scalability, deployability and continuous delivery.

By simply introducing new architectural styles, however, current problems in the collaboration and communication among stakeholders of the development and operation phases are not solved. Existing architectural models used in the development phase differ from those used in the operation phase. Three differences are highlighted and described hereafter. These differences result in limited reuse of development models during operations and limited phase-spanning consideration of the software architecture.

Abstraction levels differ in architectural models from development and operations. Architectural models especially in early phases of development often adhere to the component-based paradigm to keep track of the architecture and structure the application. Instances of the Palladio Component Model (PCM) [9] are examples of architectural models used in development. During operations architectural models are applied to reflect the executed application, e.g. its current deployment or usage intensity. Architectural models used in operations are closer to an implementation level of abstraction (e.g., reflect method calls and class signatures) as they result from monitoring data related to source code artifacts. It is hard to reproduce component models from monitoring data as knowledge about the initial component structure and boundaries is missing.

For conducting DevOps practices it is useful to combine **prescriptive and descriptive architectural models**. Currently phase-spanning consideration of architecture is not well supported. Prescriptive architectural models are used during development to document the application to be designed and implemented. In operations descriptive architectural models are applied to reflect the executed application.

There is **different content (static vs. dynamic)** in architectural models used in development and operations. Models in development specify the static software design and structure. During operations architectural models are applied to reflect dynamic content such as in-memory objects, communications between objects and utilization of servers.

This paper proposes a model-driven approach, called iObserve [4], to handle the three differences in architectural modeling by shared models and transformations in between. Additionally, in this paper we discuss the integration of dynamic content into the iObserve approach.

2 iObserve Addresses Architectural Differences to Facilitate DevOps

The iObserve approach [4] considers development-level evolution and operation-level adaptation as two mutual, interwoven processes that influence each other. Fig. 1 gives an overview of iObserve. The evolution activities are conducted by human developers, while the adaption activities are performed largely automatically by predefined routines.

iObserve addresses architectural challenges in DevOps by following the MAPE (Monitor, Analyze, Plan, Execute) control loop model. MAPE is a feedback cycle for managing system adaptation [10]. iObserve extends the MAPE loop with shared models to ease the transition between operation-level adaptation and development-level evolution [7]. iObserve applies the PCM as an architecture meta-model for modeling usage profiles, the software architecture and deployment as well as quality properties. The software application is observed to update the architectural run-time model. The architectural run-time model reflects the executed application. It is constructed by enriching a PCM instance created during development with operational observations. Based on this up-to-date model,

*This work was supported by the DFG under SPP1593.

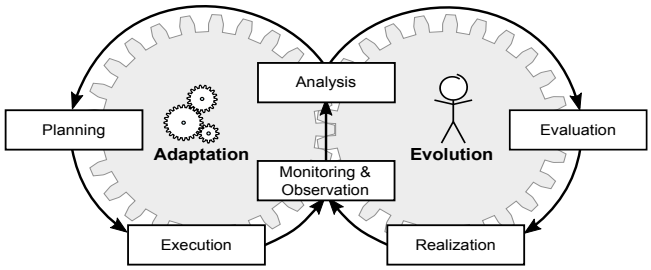

Figure 1: Overview of iObserve; Considering adaptation and evolution as two interwoven processes.

the application is analyzed to detect anomalies, e.g. increased usage intensity, and predict quality impact, e.g. an upcoming performance bottleneck. The architectural run-time model is then used as input for an adaptation plan to handle the anomalies, e.g. migrating a component from one cloud provider to another [8]. The plan is executed to update the application's configuration and deployment. Related work on model reuse and model extraction during operations is discussed in [4].

The iObserve approach provides a megamodel to **bridge the divergent levels of abstraction** in architectural models used in development and operations. Megamodels provide a notation for the relationships of models, meta-models and transformations [2]. The iObserve megamodel serves as an umbrella to integrate development models, code generation, monitoring, and model updates during operations. The iObserve megamodel is divided into four sections defined by two dimensions: one for development vs. operations, and one for model vs. implementation level. In the development section, the megamodel combines an architectural model with monitoring probes and uses it for source code generation. In the operations section, monitoring data related to source code artifacts is associated with architectural run-time model elements. Thus, the iObserve megamodel allows for reusing development models during operation phase.

iObserve employs **descriptive and prescriptive architectural run-time models** in the context of the MAPE control loop [4]. The Monitor phase exploits information from probes to keep the descriptive architectural run-time models causally connected with the underlying software application. These descriptive models are used during the Analyze phase to identify any anomalies and thus trigger adaptation. Adaptation candidates are determined by design space exploration and specified as candidate prescriptive run-time models during the Plan phase. Subsequently, these models are operationalized by deriving concrete tasks of an adaptation plan for the Execute phase. The tasks are derived by comparing candidate models to the original model and applying the KAMP approach to architecture-based change impact analysis [5]. KAMP builds upon PCM instances and provides for each change to an architecture element a set of tasks to implement the change. KAMP is applied to support operators in semi-automatically deriving adaptation plans, e.g. for solving performance and scalability issues [3].

PCM instances reflect static system design and structure and due to the extensions of iObserve represent the current properties of the executed application (e.g., its usage intensity and actual deployment) gathered by monitoring. **Live visualization of dynamic content**, like in-memory objects and their communications, will help operators adapting the application when anomalies exceed automated planning routines (cf. operator-in-the-loop adaptation [4]). Therefore, we are currently working on a combination of iObserve and the live visualization approach ExplorViz [1]. Similar to iObserve, ExplorViz monitors and aggregates events during operations to update architectural models. ExplorViz models reflect aforementioned dynamic content and relates it to execution nodes (e.g., virtual machines) and running components. However, ExplorViz lacks knowledge about initial design decisions from development phase. Integrating ExplorViz models into the iObserve megamodel combines the advantages of both approaches.

In consequence, the iObserve approach bridges differences in architectural models in development and operations regarding abstraction (component-based vs. close to implementation level), purpose (finding appropriate design vs. reflecting current application configuration) and content (static vs. dynamic).

References

[1] F. Fittkau et al. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *VISSOFT*. IEEE, 2013.

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

[3] C. Heger and R. Heinrich. Deriving work plans for solving performance and scalability problems. In *EPEW*, pages 104–118. Springer, 2014.

[4] R. Heinrich. Architectural run-time models for performance and privacy analysis in dynamic cloud applications. *SIGMETRICS Performance Evaluation Review*, 43(4):13–22, 2016.

[5] K. Rostami et al. Architecture-based assessment and planning of change requests. In *QoSA*, pages 21–30. ACM, 2015.

[6] L. Bass et al. *DevOps: A Software Architect's Perspective*. Pearson, 2015.

[7] R. Heinrich et al. Integrating run-time observations and design component models for cloud system analysis. In *MRT*, pages 41–46. CEUR Vol-1270, 2014.

[8] R. Heinrich et al. A platform for empirical research on information system evolution. In *SEKE*, pages 415–420. KSI, 2015.

[9] R. Reussner et al. *Modeling and Simulating Software Architectures - The Palladio Approach*. MIT P, 2016.

[10] Y. Brun et al. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer, 2009.