Improving Service Availabiliy with Rule-Based Adaptation

Marc Adolf Reiner Jung Lars Blümke
{mad,rju,lbl}@informatik.uni-kiel.de
Kiel University, Germany

Abstract

Self-adaptive software systems change their deployment and configuration to address changing user behavior and workloads. Such systems follow a MAPE-K approach to observe and analyze the system, and subsequently plan and execute changes. They use operations, like the (de-)replication and migration of components, to reconfigure the system. During an adaptation, some services might become unavailable when services are replicated or migrated arbitrarily. This can cause interruptions to cross service transactions and temporary service malfunctions.

While some E-commerce platforms consider this acceptable, it is irritating to the consumer. In case of safety critical systems, like medication systems, the system must be serviceable during the transition from the old to the new configuration.

In this paper, we present a rule-based approach for adaptation actions. Our approach allows to address adaptation constraints on a abstract level and decouples the constrains from setup scripts – often used in container environments. Furthermore, we evaluated the feasibility of our approach and illustrate its ability to adapt a component based web system safely.

1 Introduction

The operation of cloud applications relies on adaptive mechanisms to react to varying workloads and other external effects. Modern cloud provider often provide automatic scaling facilities for software services. However, they do not use the software architecture when scaling services. Hence, they cannot anticipate the effect on the complete software system. Furthermore, their performance predictability is limited and they are, therefore, merely reacting to load changes.

MAPE-K approach [1] allows to address these limitations, as it uses design-time and runtime knowledge to steer adaptation. It divides the process of adaptation in four phases Monitoring, Analysis, Planning and Execution. Existing implementations, like SLAstic [3], use runtime observations to forecast utilization and subsequently adapt the system to meet service level agreements. Others address dynamic feature provisions using runtime model updates [4] and learn adaptations with an adaptive knowledge base [14].

Unfortunately, these approaches do not address service availability during adaptation. This may lead in complex software systems to temporarily unavailable features. Furthermore, approaches, like SLAstic, do not explicitly address state and activity of services, which can lead to information loss. Similarly, past planning and adaptation services in iObserve [8] did not address these requirements [11, 12].

In this paper, we present a solution to these challenges with a rule-based approach, ensuring data preservation and services availability during adaptation. We divide the planning phase into *Candidate Architecture Selection* and *Adaption Planning*. The former computes a new candidate architecture. The latter derives complex adaptation actions from the candidate architecture and transforms them into atomic actions. Subsequently, the execution phase maps these actions to cloud-API specific operations. Thus, we can keep adaptation rules, composed and atomic adaptation actions technology agnostic.

Section 2 introduces the iObserve service architecture. Section 3 describes the adaptation and execution phases. Section 4 presents preliminary evaluation results. And Section 5 summarizes our finding and introduces potential future work.

2 The iObserve Approach

The iObserve approach integrates design-time evolution and runtime adaption utilizing the same architectural models for both processes [7, 8]. The adaptation process is automated, but supports operator interventions, e.g., introduction of workload characteristics, planning rules and candidate architecture selection.

iObserve follows MAPE-K loop [1] to address adaptation (cf. Figure 1). We split the loop in different microservices [9] and each phase is represented by at least one service. This supports the deployment of iObserve alongside the monitored cloud application. We monitor the cloud application with Kieker [6] supplemented by additional probes for (un)deployment, (de)allocation and utilization. As the overarching model for design-time and runtime, we use the Palladio Component Model (PCM) [10]. At runtime the PCM is used in all services to ensure explainability through all phases supporting human intervention. The runtime model is updated based on monitoring events utilizing a correspondence model to map implementation level events to model level information [7]. Based on the updated model, iObserve analyzes

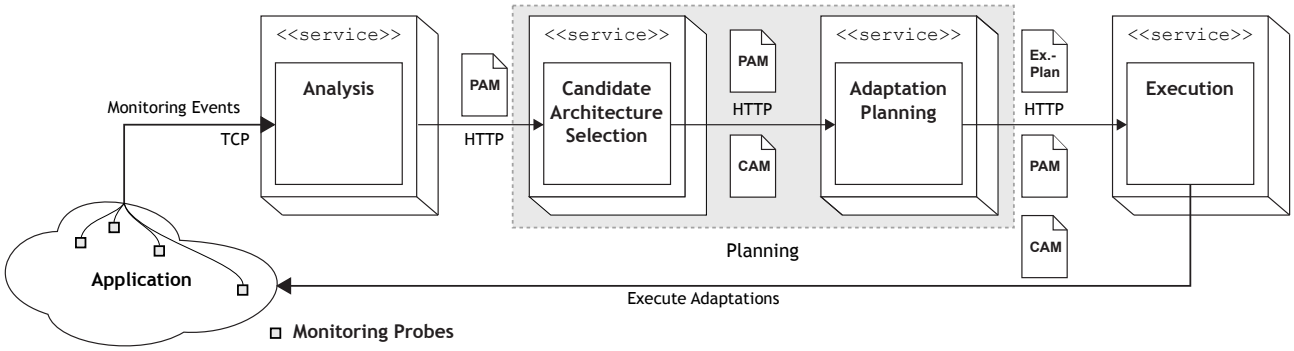

Figure 1: iObserve Architecture - Each MAPE-K phase is represented by an independent service

quality properties of the software system, like, performance and data privacy. Subsequently, in case quality requirements are not met, the *Analysis* service sends the Present Architecture Model (PAM) to the planning services. The planning is performed by two separate services (cf. Figure 1). The *Candidate Selection* service evaluates architecture alternative utilizing PerOpteryx [5] and automatically selects an optimal Candidate Architecture Model (CAM). This selection can also be deferred to an operator. The second service, *Adaptation Planning* creates the execution plan by deriving *composed adaptation actions* (migration, replication) and subsequently computing and ordering atomic adaptation actions of the execution plan to ensure service availability and data preservation during adaptation. Finally, the *Execution* service maps the atomic actions to implementation level and technology specific cloud API operations.

3 Planning and Execution

Our core contribution is rule-based adaptation planning adhering service availability and data preservation, alongside an execution service which transforms model level adaptations to specific cloud-APIs.

The Adaptation Planning services performs two tasks computing composed adaptation actions and transforming them into atomic adaptation actions for the execution plan. We use production rules for the Drools rule engine [2] to compute the composed adaptation actions from the inputs PAM and CAM. Using rules has the benefit that the adaptation process can be expressed in a concise and clear way without implementation details, like loops and recursion. Furthermore, it allows to modify the adaptation process to include additional constraints and new adaptation actions. The present rule set includes actions for migration, (de)replication, (de)allocation and the exchange of repository components, following the SLAstic approach [3] and preceding work in iObserve [11, 12].

These composed adaptation actions might need multiple atomic adaptation actions executed the right order to ensure availability and data preservation. They are generated by an extensible model transformation. We conceived eleven atomic adaptation action types based on existing cloud-APIs, like, Kubernetes, their limitations (e.g. cannot detect availability of a service), and previous work [11, 12, 13] They comprise service (un-)deployment, (dis-)connecting service, migration, request blocking, finish and resource (de-)allocation. While deployment and allocation are self-explanatory, (dis-)connecting service, request blocking and finish need more details. *Connecting service* is necessary in replication and migration cases where, e.g., load balancer reconfiguration must be performed explicitly. *Request blocking* ensures that a service does not receives new requests. And *finish* halts the execution plan until a service becomes available omitting unresponsive services is omitted.

The resulting execution plan is ordered to ensure service availability and data preservation, and then send to the *Execution* service. At present this service performs the actions in sequential order. Each action is mapped from model level to implementation level utilizing the correspondence model, i.e., the service identifies the correct cloud API and provider, and subsequently maps the action to specific operations. For example, in Kubernetes explicit connecting services is not necessary in replication scenarios, as this is automatically done by the Kubernetes, where plain Docker infrastructure and virtual machine infrastructure requires specific operations. The mapping is provided by multiple API specific action executors, which address the needs of each cloud API.

4 Evaluation

To prove the feasibility of our approach, we conduct several experiments on our local Kubernetes cluster. This cluster includes one master and four worker nodes. We test our approach with a distributed variation of the JPetStore [15]. It represents a simple web shop for animals. Thereby, we want to check the creation, application and result of the execution plan.

Here, we present a *migration and (de-)allocation* experiment that we conducted along others [13]. In this scenario, the component which controls the accounts is migrated to a new node and the corresponding CAM is send to the adaptation service. Thereby, the (de-)allocation of the new node, (de-)replication

Name	Desired	Available	Up-To-Date
account	1	1	57s
catalog	1	1	57s
account2	1	0	1s
catalog	1	1	58s
account2	1	1	2s
catalog	1	1	59s

Table 1: Excerpt of the following states in Kubernetes in the experiment.

and connection of the component have to be in the right order. We check all three described phases. The resulting the execution plan contains the expected actions. In Table 1, we see the actual application as following states of the Kubernetes cluster. Here, we only display the account components and the catalog component. The attributes describe the amount of the components and their state. As one can see, the first account service is shut down and a new one is created. There is a slight delay between the termination of the first one and the availability of the new one. Nevertheless, the replacement is already created and starting and there is no state in which there is no account component.

In our test case using Kubernetes, we could not observe failures in the modification of the deployments. All resulting software architectures were as we expected. For further details refer to [13].

5 Conclusion

We presented our rule-based adaptation approach to create a set of ordered composed adaptation actions that are translated to atomic adaptation actions for the execution phase. Through the microservice architecture and the rule-based system, we achieve a loosly coupled system that can be easily reconfigured and extended. This allows to include different strategies for adaptation planning and support different cloud provider APIs within the iObserve approach. Finally, we conducted a feasibility evaluation in our own Kubernetes cluster with an instance of the distributed JPetStore. The result was successful and as expected except in the migration. We encountered an issue where the first account component was already terminated while the second one was still starting and not available, while this should be avoided by the finish action, the initial implementation of the action only tested if the container is reachable.

In the future, we will address this issue and provide more specific finish actions. In addition, we want to evaluate the approach with a larger software system and changing demands to determine if the expectations hold in complex environments. External changes that occur during the adaptation also have to be regarded. For this, we need to check if the system is changing during other phases beside the monitoring phase. In the rule system, smarter rules and an comprehensible rule overview may improve its usability.

References

[1] A. Computing et al. “An architectural blueprint for autonomic computing”. In: *IBM White Paper* 31 (2006), pp. 1–6.

[2] M. Proctor et al. *Drools*. 2007.

[3] A. van Hoorn et al. “An Adaptation Framework Enabling Resource-efficient Operation of Software Systems”. In: *Proc. of WUP 2009 at ICSE 2010*. ACM, Apr. 2009, pp. 41–44.

[4] B. Morin et al. “Taming Dynamically Adaptive Systems Using Models and Aspects”. In: *Proc. of ICSE 2009*. ICSE ’09. IEEE Computer Society, 2009, pp. 122–132.

[5] A. Koziolek, H. Koziolek, and R. Reussner. “PerOpteryx: Automated Application of Tactics in Multi-Objective Software Architecture Optimization”. In: *Proc. of QoSA and SIGSOFT*. ACM. 2011, pp. 33–42.

[6] A. van Hoorn, J. Waller, and W. Hasselbring. “Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis”. In: *Proc. of ICPE 2012*. ACM, Apr. 2012, pp. 247–248.

[7] R. Heinrich et al. “Architectural Run-Time Models for Operator-in-the-Loop Adaptation of Cloud Applications”. In: *Proc. of MESOCA*. IEEE, Sept. 2015, pp. 36–40.

[8] R. Heinrich et al. *Run-time Architecture Models for Dynamic Adaptation and Evolution of Cloud Applications*. Tech. rep. Kiel, Germany, 2015.

[9] S. Newman. *Building Microservices*. O’Reilly Media, Inc., 2015.

[10] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, 2016. 408 pp.

[11] T. Pöppke. “Design Space Exploration for Adaption Planning in Cloud-Based Applications”. Master Thesis. KIT, 2017.

[12] P. Weimann. “Automated Cloud-to-Cloud Migration of Distributed Software Systems for Privacy Compliance”. Master Thesis. KIT, 2017.

[13] L. E. Blümke. “Planning and Execution of System Adaptations in Cloud-Based Environments”. Master Thesis. Kiel University, 2018.

[14] V. Klös, T. Göthel, and S. Glesner. “Comprehensible Decisions in Complex Self-Adaptive Systems”. In: *SE 2018*. Gesellschaft für Informatik, 2018, pp. 215–216.

[15] *MyBatis JPetStore application*. `http://www.mybatis.org/spring/sample.html`. 2017.