Planning and Execution of System Adaptations in Cloud-Based Environments

Master’s Thesis

Lars Erik Blümke

June 13, 2018

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by:  Prof. Dr. Wilhelm Hasselbring
Dr.-Ing. Reiner Jung
M.Sc. Marc Adolf
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 13. Juni 2018
Abstract

The operation of software systems which are subject to varying loads is a challenge. On one hand Quality of Service (QoS) requirements such as response times and availability have to be met, while on the other hand the operating costs have to remain minimal. Adaptive systems in cloud-based environments have become a popular approach in this regard: Cloud resources can be allocated fast if the system needs to adapt to higher loads and deallocated to save costs once they are no longer needed.

The MAPE-K loop is a common approach for automated computation of system adaptations. It is based upon the four phases Monitoring, Analysis, Planning, and Execution which all share a common Knowledge base. The model-based iObserve approach features a control loop based on the MAPE-K phases to address the challenges to the adaptation of cloud based systems mentioned above.

In this master’s thesis, we present concepts and implementations for the planning and execution phases within iObserve. We try to add an optimization algorithm to iObserve which creates candidate architecture models adapted to certain scenarios. A rule-based approach to derive concrete adaptation actions from these candidate models by comparing them to models of the present architecture is presented. The derived adaptation actions are refined to also take into account component dependencies and ensure the real system’s availability during the adaptation. We provide an implementation to actually apply the system adaptations to a real system. All our contributions to the iObserve approach are implemented in a service based architecture separating the different adaptation phases into dedicated services. Finally, we present a feasibility study which shows that besides technical issues with the optimization algorithm our approach is capable of adapting a component-based software system deployed on a Kubernetes cluster.
Contents

4.2.1 System Adaptations in the Existing Approach ......................... 41
4.2.2 Outline for the Evolution of Execution Plan Computation ............ 42
4.2.3 Implementation of the Execution Plan Computation Evolution ........ 42
4.2.4 Results of the Execution Plan Computation’s Evolution Process ........ 58
4.3 Execution of Execution Plans on Concrete Cloud Environments ........... 58
4.3.1 Execution of System Adaptations in the Existing Approach ............ 58
4.3.2 Outline for the Evolution of the Execution ................................. 59
4.3.3 Implementation of the Execution Evolution ................................. 59
4.3.4 Results of the Execution’s Evolution Process .............................. 64

5 Evaluation ................................. 65
5.1 Evaluation Goals ............................................. 65
5.2 Methodology ................................................. 65
5.3 Experimental Design .......................................... 66
  5.3.1 Execution Environment .................................. 66
  5.3.2 Adaptation Scenarios .................................. 68
5.4 Operation ......................................................... 70
5.5 Data Collection .................................................. 71
5.6 Results .......................................................... 71
  5.6.1 Replication Scenario ...................................... 72
  5.6.2 Dereplication Scenario .................................... 74
  5.6.3 Migration and (De-) Allocation Scenario ......................... 77
5.7 Discussion ......................................................... 81
5.8 Threads to Validity ............................................. 82
5.9 Summary .......................................................... 82

6 Related Work ................. 85
6.1 SLAastic ....................................................... 85
6.2 DiVA ............................................................ 85
6.3 Adaptive Knowledge Bases for Self-Adaptive Systems ..................... 86
6.4 CDOXplorer ...................................................... 87

7 Conclusions and Future Work ........................................... 89
7.1 Conclusions ...................................................... 89
7.2 Future Work ..................................................... 90

Bibliography ................................................. 93

Appendix ................. 97
A.1 LQN Solver Installation Instructions ................................. 98
A.2 Required PerOpteryx and Palladio Libraries ............................. 99
Chapter 1

Introduction

1.1 Motivation

The operation of software systems on a given hardware environment with varying load on the system is a challenge. There are high expectations to enterprise systems in terms of Quality of Service (QoS) metrics such as response times, throughput, and availability. The workload, e.g. the number of incoming clients on a website, may vary over time. Factors like the time of the day, the day of the week, seasonal factors like Christmas sales or sudden events like a commercial on television may influence the workload. At all time, enough hardware resources have to be available to meet the system’s QoS goals. At the same time, the operation costs shall remain as low as possible. This leads to the problem of either having to invest into additional hardware resources to keep the system responsive during a load peak but not utilizing them during the rest of the time. Or saving costs by only providing enough hardware resources for average load but losing the system’s responsiveness at high load peaks [Roy et al. 2011]. Available hardware resources also take a certain amount of time to start and reacting to load peaks just in time is another challenge for the operator. Additionally, hardware may break down, has to be maintained, and also compensated.

Cloud computing became more popular in this context. It is offering almost unlimited virtual resources on demand. They can be allocated and deallocated in a fast and dynamic way only when needed, allowing companies to allocate and pay for exactly the number of resources they need at the moment. With cloud computing, they do not have to maintain their resources. Instead, by using large-scale computer data centers, cloud computing enables a decrease in cost of electricity, network bandwidth, operations, software, and hardware [Armbrust et al. 2010]. Khajeh-Hosseini et al. [2010] present a case study on the migration of an IT system in the oil and gas industry from an in-house data center to the Amazon Web Services infrastructure. Over a period of five years, they noticed a decrease of operating costs by 37% and an increase of the system’s scalability and effectiveness, at the same time. However, reacting to load peaks or generally deciding when to adapt the system remains a challenge.
1. Introduction

1.2 Goals

In this thesis, we want to focus on the automated planning and execution of system adaptations in cloud-based systems. During its life cycle, such a system can face issues regarding its performance or privacy terms, for example. We present and extend the iObserve approach [Hasselbring et al. 2013], which is meant to support and optimize the operation of software systems in cloud-based environments in a semi-automated way. iObserve is already able to monitor a running software system and analyzes the data for the need for improvements. Previous works already enabled iObserve to plan certain system adaptations to address concrete performance [Pöppke 2017] or privacy goals [Weimann 2017]. These approaches were also capable of executing the adaptations automatically. With these preconditions iObserve is an relevant approach for further research concerning the planning and execution of system adaptations. We want to focus on a) shortcomings of the existing approaches concerning the availability of the cloud-based system during adaptation and b) improving the integration of the existing approaches into iObserve’s system architecture. We define the goals for this thesis in the rest of this section.

We use the Goal Question Metric (GQM) approach [Caldiera and Rombach 1994] to develop our goals, research questions, and evaluation metrics. The GQM is based on a top-down approach: At first, it specifies on an abstract level which goals shall be accomplished. A set of research questions then characterizes and refines each goal. Finally, data-based metrics for answering each question are defined.

G1: Architectural Integration of an Existing Architecture Optimization Approach  The existing approach for adaptation planning [Pöppke 2017; Weimann 2017] is already able to compute a model-based candidate architecture improving upon the system’s performance or the violation of privacy terms. The approach uses an external optimization tool to achieve this. We want to integrate this tool into iObserve. However, we want to provide a flexible architecture where the optimization algorithm remains exchangeable. Additionally, we want to enable an operator to step in when multiple (or no) adaptation alternatives have been found. We want to decouple the planning process and provide it as a separate service. For Goal 1 we define the following research questions:

- RQ 1.1: Can we integrate an existing optimization tool into iObserve to compute a candidate architecture?
- RQ 1.2: Can we provide a service-based architecture supporting exchangeable optimization algorithms?

G2: Improvement of an Existing Computation Method for Execution Plans to Address Availability of the Observed System During Execution  The existing approach for the execution of adaptations [Pöppke 2017; Weimann 2017] is already able to compute adaptation actions from the difference between models of the present and the candidate architecture. It
1.2. Goals

is capable of placing them in an executable order and applying them to the real system. We want to focus on the system’s availability during the execution and develop an approach which focuses on component dependencies and minimizes downtimes of the system. We want our architecture to support the exchange of computation algorithms as well as the exchange of mechanisms to access the cloud-providers APIs. We want to decouple the execution process and provide it as a separate service. For Goal 2 we define the following research questions:

- **RQ 2.1:** Can an execution plan which does not take into account all dependencies between system components lead to a not properly working system?

- **RQ 2.2:** Can we identify specific rules which help with the computation of execution plans?

- **RQ 2.3:** Can we provide a service-based architecture supporting exchangeable mechanisms for the computation and execution of execution plans?

G3: Evaluation of the Approach  We examine different aspects to answer our research questions. We have to consider whether our specific approach fulfills the requirements for stability, availability, and recovery for such an approach.

- For **RQ 1.1** we want to evaluate if the optimization tool used in the existing approach can be integrated into iObserve.

- For **RQ 1.2** we want to evaluate if the planning phase can be decoupled into a separate service. Additionally, we want to know if our architecture potentially enables the use of different optimization tools in the future.

- Regarding **RQ 2.1**, simple execution plans which do not take into account all dependencies between system components might compromise the system’s availability. For example, they could cause unavailable components during migration, forgotten data transfers and so on. We want to evaluate the effects a too simple execution plan can cause.

- For **RQ 2.2** we want to evaluate whether we are able to find certain rules for the computation of execution plans which ensure that a functioning adaptation plan can be found.

- For **RQ 2.3** we want to evaluate if the execution phase can be decoupled into a separate service and if our architecture potentially supports the use of different mechanisms in the future.
1. Introduction

1.3 Document Structure

This thesis is structured as follows: In Chapter 2 we present foundations related to the approach of iObserve in general and technologies we will use in particular. In Chapter 3 we give an overview on iObserve’s architecture and present what its different phases are supposed to do. In Chapter 4 we reference to the existing approaches of planning and execution within iObserve. We describe in which way we built upon them and explain our design decisions. We evaluate our approach in Chapter 5 and discuss the results along with possible threats to validity. In Chapter 6 we present similar approaches to iObserve in general or parts of its control loop in particular. With Chapter 7 we conclude our work, reflect upon our results, and present future work.
Foundations and Technologies

We present a number of concepts and technologies which are relevant to the thesis. Generally, the implementation part of the thesis will take place in the context of iObserve. We introduce this approach, together with its underlying concept MAPE-K, and the internally used Palladio Component Model. The Pipe-and-Filter Framework TeeTime is also presented in this context. We introduce the PerOpteryx Eclipse plug-in for software architecture optimization. An existing approach to the planning and execution of system adaptations which uses this plug-in is also presented. We introduce the Goomph plug-in for Gradle which addresses the integration of plug-in based Eclipse RCP applications. We also introduce the Drools rule management system and the Kubernetes container orchestration system which will both be utilized in this thesis. Finally, we present the Goal Question Metric approach as a measurement mechanism for feedback and evaluation.

2.1 The MAPE-K Approach to Autonomic Computing

Autonomic computing is intended to address the complexity of IT-systems by using technology to manage technology [IBM 2006]. By using autonomic computing, systems can be designed to be self-configuring (dynamically adapt to changes in the environment), self-healing (detect and initiate corrective actions for system malfunctions), self-optimizing (reallocate resources in response to dynamically changing workloads) or self-protecting (detect, identify, and protect against threats). Autonomic computing is often realized with an internal control loop structure, where autonomic adaptations are applied to the system with every iteration [Kephart and Chess 2003; IBM 2006]. MAPE-K is a common approach for addressing the challenges of autonomic computing. The letters M, A, P, E represent different phases within the control loop: Monitoring, Analysis, Planning, and Execution. All four phases share Knowledge.

During the monitoring phase details from the managed system resources are collected. “The details can include topology information, metrics, configuration property settings and so on. This data includes information about managed resource configuration, status, offered capacity, and throughput” [IBM 2006]. These information are the basis for the following phases of the loop. Additionally, the data may also be further aggregated and filtered in this phase.
2. Foundations and Technologies

- The analysis phase contains methods to observe the collected information, find anomalies, and decide whether adaptation actions need to be applied to the system. Model-based approaches may be used in this context to represent the real system. A certain policy defines whether the results of the analysis result in the adaptation of the system.

- Triggered by the analysis phase, the planning phase is concerned with addressing the issues detected by the analysis phase. The goal of this phase is to develop and plan an adaptation of the system, which improves on the given issues. Several approaches are possible to achieve this, “ranging from a single command to a complex workflow” [IBM 2006].

- The execution phase finally applies the planned adaptation actions to the system. For this reason “some actions may need to be taken to modify the state of one or more managed resources” [IBM 2006]. Some sort of scheduling of the different adaptation actions may also be necessary.

![Figure 2.1. Functional details of the autonomic manager [IBM 2006]](image)

In the original blueprint of the approach by IBM [2006] these phases form a so-called autonomic manager as shown in Figure 2.1. An autonomic manager gets his inputs through a sensor interface and modifies his environment through an effector interface (at the bottom of Figure 2.1). On the other hand, an autonomic manager provides sensor and effector interfaces to other autonomic managers (at the top of Figure 2.1). Through these interfaces,
other autonomic managers can monitor the current manager’s actions and affect them by determining its policy. This adds the concept of modularization to the approach. Multiple autonomic managers can be combined to reduce every single manager’s complexity and be able to even realize large autonomic systems. In the context of this thesis, autonomic computing is used to adapt a cloud-based system’s infrastructure to changing requirements to the system. In particular, iObserve is using the MAPE-K approach.

### 2.2 The Palladio Component Model

The Palladio Component Model (PCM) is a meta-model for modeling component-based software systems [Becker et al. 2009]. It is composed of five different submodels with each of them modeling another domain of a software system. These domains are inspired by different developer roles in the real world. The submodels of the PCM are:

- **Repository model**, which represents the domain of a component developer. It includes different components of a software system, their roles, interfaces between them, and the signatures of interface methods. As implied by its name, this model acts as a repository for any kind of software components. However, it does not contain any information about a composed system’s architecture, i.e. how the different components are arranged in a concrete system architecture.

- **System model**, which represents the domain of a system architect. The system model defines a software system’s architecture on a conceptional level. Its main components are assembly contexts and assembly connectors. An assembly context encapsulates a component from the repository model and exposes this component’s provided and required roles. The exposed roles of an assembly context can be connected with assembly connectors. Such a connector always connects a required and a provided role. The system’s architecture is modeled by arranging components in assembly contexts and connecting them. Note that a component may be encapsulated in different assembly contexts if it is used at different positions within a system’s architecture. The system model does not include any information about the system’s deployment, i.e. how many deployed instances there are of each assembly context.

- **Resource environment model**, which represents the domain of the system deployer. It represents hardware environment available to the software system and contains information about servers and network links. The servers in the resource environment model are called resource containers. A network link between two resource containers can be modeled with a linking resource. The resource environment model does not include any information about the system’s deployment, i.e. which assembly context instances are running on which resource container.

- **Allocation model**, which also represents the domain of the system deployer. It models the actual deployment of a software system. Its core component are so-called allocation
2. Foundations and Technologies

contexts. An allocation contexts maps an assembly context to a resource container. Thus, it represents the deployment of a concrete instance of the component encapsulated in the allocation context on a concrete resource container.

Usage model, which represents the domain of a domain expert and models use cases of the entire software system to extract certain user behaviors. It is not relevant for this thesis as we only focus on system architectures and deployments.

The PCM is implemented as an Ecore model using the Eclipse Modeling Framework\(^1\) (EMF). Therefore, the submodels can contain links to other submodels. An allocation context from the allocation model links an assembly context from the system model and a resource container from the resource environment model, for example. Additionally, the EMF provides mechanisms to serialize the Ecore submodels in an XML file format. In this thesis the PCM is used to represent cloud-based architectures.

2.3 TeeTime

TeeTime\(^2\) is a Pipe-and-Filter framework available for Java and C++ [Wulf et al. 2017]. We use TeeTime in this thesis to create a Pipe-and-Filter architecture for the planning and execution phases. Both phases process data in subsequent steps and the TeeTime framework provides a flexible architecture for this scenario. The Pipe-and-Filter architecture style is suited for the processing of data streams. The filters, as components in this architecture style are called, provide input and output interfaces often referred to as ports. Data is received on the input port, processed within the filter and the results are provided at the filter’s output port. The pipes represent the connectors in the Pipe-and-Filter architecture style. A pipe describes a binary relationship between two filters by connecting an output port to an input port [Monroe et al. 1997].

In the TeeTime framework, filters are called stages. Each stage has typed input and output ports. Stages are connected with pipes between output and input ports. The framework offers different abstract stages which follow different patterns. Producer stages only have an output port while consumer stages only have an input port. Filter stages have an input and output port of the same type while the input and output port have different types in transformation stages. There is also a basic abstract stage without any predefined ports. All abstract stages can be extended to implement individual stages and additional ports can be added. Composite stages can be assembled of multiple sub-stages. TeeTime also provides a number of predefined ready-to-use stages for common tasks such as general


2.4 iObserve

The iObserve approach supports the adaptation and evolution of cloud-based software systems through run-time observation and continuous quality analysis [Heinrich et al. 2015]. Hence, it enables software engineers to constantly keep track of their system’s properties. Amongst others, it observes the system’s performance and privacy goals at runtime, which are important quality metrics in cloud-based environments.

![Figure 2.2. iObserve adaptation and evolution life-cycle [Hasselbring et al. 2013]](image)

iObserve uses a model-driven approach to represent, analyze, and adapt a software...
system’s architecture as well as its deployment at runtime. Figure 2.2 shows an overview of the iObserve approach with its two interwoven processes adaptation and evolution.

The adaptation cycle uses the PCM meta-model internally and features a control loop which resembles the MAPE-K approach from 2.1. At first, the cloud-based software system is instrumented to enable monitoring and a model of the system is created at design time. Afterwards, the system is continuously monitored and observed to detect component migration, (de-)replication, (de-)allocation, resizing of virtual machines or changes in the system’s usage intensity or user behavior. Observed low-level events are preprocessed and used to update the runtime architecture model which is passed to the following phase. The analysis checks if the performance and privacy goals are met in the system’s current state. If this is not the case anymore or if a violation is predictable in the near future, iObserve invokes planning routines to adapt the software system to fulfill its requirements again. If possible, these routines are performed automatically. However, this may not always be possible. Therefore, an operator may step in to choose from multiple possible adaptation alternatives. If the chosen adaptations pass the evaluation, they are executed automatically and the loop restarts.

If the system still fails in achieving the performance and privacy goals, an operator may inspect and evaluate the system’s architecture manually within the evolution cycle. Such an operator may be a software engineer with additional domain-specific knowledge. iObserve supports them during this task by providing the current runtime architecture model. If the software architect finds and realizes an appropriate solution, the architecture runtime-model has to be updated and the loop restarts. The evolution cycle does not have to be a result of missed performance or privacy goals. It can also represent a software system’s evolution over time due to changing requirements and operation conditions. So far, iObserve only incorporates the monitoring and analysis phase. Planning and execution have been addressed in Pöppke [2017]’s and Weimann [2017]’s master’s theses but only as extensions to the analysis. During this thesis, we plan on extending their additions to iObserve as standalone phases.

2.5 PerOpteryx

PerOpteryx [Koziolek et al. 2011] aims at automated architecture reasoning and analysis and also optimizing a software system’s architecture. It takes PCM architecture models as input and computes several candidates which are optimal trade-offs between the considered quality attributes and enables the software developer to choose from them. It is capable of taking into account several degrees of freedom into its decision processes such as the component deployment, hardware sizing, and processing power of hardware components, component selection and configuration options of components, servers, and middleware. To do this, it incorporates architectural performance tactics into a metaheuristic optimization process which encodes the architecture improvement as an optimization problem and
2.5. PerOpteryx

applies an evolutionary algorithm as a general-purpose, problem-independent optimization strategy. For performance analysis, it uses expressive layered queuing networks.

For a concrete PCM instance, PerOpteryx is trying to improve the performance and the costs. Because in many cases multiple, possibly conflicting quality attributes are considered, the tool is searching for so-called Pareto-optimal candidates. A candidate is Pareto-optimal iff there exists no other candidate that is better in all quality criteria. PerOpteryx outputs multiple candidates which are Pareto-optimal with respect to all candidates evaluated and approximates the set of Pareto-optimal candidates as well. Figure 2.3 shows the general process of the architecture optimization process with PerOpteryx. It includes the following steps:

1. In the first step, the degree of freedom instances are automatically extracted from the PCM instance. This, for example, results in a set of all different, possible allocations of different system components on different performing hardware components.

2. In the second step, the evolutionary algorithm is applied and creates new candidates. In detail, it executes these sub-steps:

![PerOpteryx process model](image_url)

*Figure 2.3. PerOpteryx process model [Koziolek et al. 2011]*
2. Foundations and Technologies

- Step 2a evaluates the generated genomes.
- Step 2b selects the best genomes.
- Step 2c Reproduces new genomes from the selected one using crossover (merging the genomes of two candidates) and mutation (varying design options like the deployment).

3. In the third step, the results are presented to the software architect.

PerOpteryx comes as an Eclipse plugin per default. In the context of this thesis, we plan on using the tool during the planning phase of iObserve. The fact that PerOpteryx, as iObserve, uses the PCM as the internal model of the represented system’s architecture is a major advantage in this context.

2.6 An Existing Planning and Execution Approach for iObserve

There is already an approach called “Design Space Exploration for Adaptation Planning in Cloud-based Systems” [Pöppke 2017] for iObserve. It uses PerOpteryx for the planning phase and also implements an execution phase. Another approach called “Automated Cloud-to-Cloud Migration of Distributed Software Systems for Privacy Compliance” [Weimann 2017] focuses on addressing privacy issues with iObserve and was developed in cooperation with the first approach. They extend iObserve’s pipe-and-filter architecture as shown in Figure 2.4. In the Analysis package, the SnapshotBuilder filter creates a copy of the architecture model at the end of the analysis phase. It sends the EMF URI to the CandidateGenerationFilter in the Planning package. This filter contains the ModelProcessing filter, the ModelOptimization filter, and the CandidateProcessing filter. The first uses the EMF URI to receive the architecture model from the file system. It initializes a PlanningData record containing the model URI. It preprocesses the model and also adds the preprocessed model’s URI to the PlanningData before sending it to the ModelOptimization filter. This filter calls PerOpteryx to generate an optimized architecture model. This model’s URI is also added to the PlanningData record which is then sent to the CandidateProcessing filter. This filter selects a created candidate and creates all required information for further processing. These information are now stored in an AdaptationData record which is used by all following stages. The CandidateProcessing filter is connected to the SystemAdaptation filter in the Adaptation package. This filter contains the AdaptationCalculation filter which compares the model of the current system’s architecture to the optimized architecture model and derives the adaptation actions. They are also stored in the AdaptationData record which is forwarded to the next filter, the AdaptationPlanning filter. This filter places the actions in an executable order and the AdaptationExecution filter executes the ordered action sequence on the real cloud-based system. As shown in Figure 2.4, the architecture of this approach is structured in different packages. All these packages are part of the analysis project of the iObserve approach. Note
2.6. An Existing Planning and Execution Approach for iObserve

that Pöppke and Weimann use a different terminology for the architecture models used in their approach. They use the term runtime-model instead of present-architecture-model and redeployment-model instead of candidate-architecture-model.

Candidate Generation with PerOpteryx  The approach uses PerOpteryx for the generation of planning alternatives. As mentioned in 2.5, the tool’s evolutionary algorithm is able to compute planning alternatives with respect to cost and performance constraints and is using the PCM. The approach contains an extension to the standard PCM meta-model.
2. Foundations and Technologies

for cloud-based systems but remains compatible with PerOpteryx. The design space for optimization of a cloud-based application is spanned by the available cloud resources and the components that can be allocated to them. The number of components is limited. With cloud resources, this becomes more complex. Cloud providers usually define a set of VM types. However, the number of replications of VMs of these types is practically unlimited. PerOpteryx was not designed with cloud-applications in mind. Especially possibly unlimited VM replications are a problem because the possibilities of allocating components are practically unlimited as well. For this reason, the allocation of components is constrained to groups of resource containers of the same type, which limits the design space for PerOpteryx. These groups are called allocation groups. Elasticity is still guaranteed by replicating the allocation groups but the issue of having unlimited possibilities of allocating components is solved. A disadvantage of this approach is that it is no longer possible to allocate components arbitrarily to any kind of resource container. PerOpteryx is configured to use two degrees of freedom. The allocation degree, which enables the (de-) allocation of components onto any available allocation groups and the replication degree, which describes the possibility of a resource container to be (de-) replicated.

The main goals for the optimization are to decrease the response time of the application while minimizing the deployment costs at the same time. Pöppke [2017] states that adding additional constraints, like privacy goals is still possible. For the evaluation of the candidates during its evolutionary algorithm, as described in Section 2.5, PerOpteryx can use the analysis tools the PCM provides. In this approach, an LQN solver is used because it supports server replications and its numerical computing approach is inherently faster than the simulation approach of other alternatives. Instructions on how to install the LQN solver can be found in Appendix A.1. Finally, the candidates computed by PerOpteryx are received. Because there can be multiple Pareto-optimal candidates, the tool was adapted to always output the one with the lowest costs. Pöppke [2017] states that this “tends to be the one with the least amount of resource containers in use”. Generally, this might also be the input candidate because PerOpteryx always regards it as a possible output candidate.

Computations of Adaptation Actions The approach already realizes the comparison of the current architecture’s runtime-model and the planned architecture’s redeployment-model to derive the necessary adaptation actions. For this reason, the so-called system-adaptation-model and a comparison algorithm were introduced. The algorithm is able to compute the actions which are necessary to transfer the runtime-model into the redeployment-model. The system-adaptation-model visualized in Figure 2.5, is another Ecore model representing the available adaptation actions and the information needed to execute them [Pöppke 2017; Weimann 2017]. The model contains the following actions:

3Layered Queuing V5 LQN Solver. URL: https://github.com/layeredqueueing/V5 (visited on 16/03/2018).
2.6. An Existing Planning and Execution Approach for iObserve

- **Assembly Context Actions**
  - Allocate Action
  - Deallocate Action
  - Migrate Action
  - Change Repository Action

- **Resource Container Actions**
  - Acquire Action
  - Replicate Action
  - Terminate Action

The Assembly Context Actions model changes of software components. The Allocate Action models the new or first deployment of a component, while the Deallocate Action represents the undeployment of a component. With the Migrate Action, a component is moved from one resource container to another. The Change Repository Action is used when a component is replaced with an equivalent component due to better performance characteristics, for example. The Resource Container Actions model changes of resource containers. The Acquire Action represents the start of a server or virtual machine while the Terminate Action represents its shutdown. The Replicate Action clones a resource container instance with all deployed components [Weimann 2017]. These actions roughly match the five reconfiguration options for the adaptation of component-based software systems Van Hoorn et al. [2009] proposed. They also named allocation, deallocation, and migration and described the principles of replication and dereplication under the terms load balancing and load unbalancing.
Figure 2.5. The system-adaptation-model containing all available adaptation actions [Pöppke 2017]
2.6. An Existing Planning and Execution Approach for iObserve

An additional component graph shown in Figure 2.6 is created. This graph contains the relevant information for the computation of execution plans. Such a graph is used because it is difficult to access the particular data in the different PCM models. The graph contains the DeploymentNode which is a host representation and the ComponentNode which is a component representation. The ComponentEdge represents data streams and interfaces [Weimann 2017]. This graph is used for the implementation of the comparison algorithm for assembly context actions which will be introduced now. All components referenced in the algorithm are of type ComponentsNode.

The comparison algorithm is shown in Algorithm 1. It uses a dictionary of all components from the runtime-model (l. 1) to create a list of transformation actions (l. 2). The INIT procedure (l. 4-8) maps each runtime-model component to the id of its assembly context to create the dictionary. For a list of all components of the redeployment-model, the CALCULATEACTIONS procedure (l. 11-30) tries to get the correspondent runtime-model component (l. 13). This is realized by looking up the id of the redeployment-model component’s assembly context in the dictionary. If no such component is found it needs to be allocated and an AllocateAction is created (l. 14-15). If a component is found whose allocation content id matches two more cases are possible. If the runtime-model component’s id differs from the redeployment-model component’s id the component must have been swapped with an equivalent one. In this case, a ChangeRepositoryAction is created (l. 17-19). If the ids of runtime- and redeployment-model component’s resource containers differ, the component must have been migrated. A MigrateAction is created in this case (l. 20-22). All runtime-model components whose assembly context still exists in the redeployment-model are then removed from the dictionary (l. 24). All components that remain in the dictionary are not needed anymore. They can be deallocated and DeallocateActions are created for each of them (l. 27-29). A similar algorithm is used to compute the resource container actions. In this second algorithm, a dictionary which maps resource container ids to deployment nodes is created.

**Ordering of Adaptation Actions** Due to dependencies in cloud-based environments, the adaptation actions computed so far cannot be executed in an arbitrary order. For example, a component can’t be allocated to a resource container which has not been
2. Foundations and Technologies

Algorithm 1 Action calculation algorithm [Weimann 2017]

1: Dictionary components
2: List of Action actions
3:
4: procedure Init(List<Components> runtimeComponents)
5:   for all runComponent ← runtimeComponents do
6:     components.put(runtimeComponent.AssemblyContextID, runComponent)
7:   end for
8: end procedure
9:
10:
11: procedure CalculateActions(List<Components> reDeplComponents)
12:   for all reDeplComp ← reDeplComponents do
13:     runComp ← get(reDeplComp.AssemblyContextID)
14:     if runComp == Null then
15:       actions.add(new AllocateAction(...))
16:     else
17:       if runComp.ComponentID != reDeplComp.ComponentID then
18:         actions.add(new ChangeRepoAction(...))
19:       end if
20:       if runComp.ResContainerID != reDeplComp.ResContainerID then
21:         actions.add(new MigrateAction(...))
22:       end if
23:     end if
24:     components.remove(reDeplComp.AssemblyContextID)
25:   end for
26:
27:   for all runComp ← components do
28:     actions.add(new DeallocateAction(...))
29:   end for
30: end procedure
2.6. An Existing Planning and Execution Approach for iObserve

acquired yet [Pöppke 2017]. Pöppke and Weimann use a dedicated algorithm for ordering
the adaptation actions which orders the actions basically based on their type. For this
reason, three assumptions are needed [Weimann 2017]:

\( \Rightarrow \) Each component is affected by an action only once.

\( \Rightarrow \) The Change Repository Component Action does not affect a component.

\( \Rightarrow \) A server never gets acquired and terminated in one sequence.

Weimann [2017] states that these assumptions are correct since the comparison algorithm
described in the previous section calculates a direct transition into a redeployment state.
Additionally, the order of actions of the same type does not matter. However, the following
conditions are needed to derive an order of different action types:

<table>
<thead>
<tr>
<th>Action</th>
<th>Pre-Execution-Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>execute after Acquire</td>
</tr>
<tr>
<td>Deallocate</td>
<td>execute before Terminate</td>
</tr>
<tr>
<td>Migrate</td>
<td>execute after Acquire and before Terminate</td>
</tr>
<tr>
<td>Change Repository Component</td>
<td>execute before Migration</td>
</tr>
<tr>
<td>Acquire</td>
<td>-</td>
</tr>
<tr>
<td>Terminate</td>
<td>-</td>
</tr>
<tr>
<td>Replicate</td>
<td>-</td>
</tr>
</tbody>
</table>

Based on these conditions the following order of action types is derived:

1. acquire actions
2. change repository component actions
3. deallocate, allocate, and migrate actions
4. terminate and replicate actions

However, Pöppke [2017] admits that this approach assumes that there are no further
dependencies between the components. Additionally, possible changes of other artifacts
such as configuration files, build-scripts or test cases are not taken into account.

Application of Execution Plans  So far, the adaptation actions have been computed on
a model basis but to complete iObserve’s MAPE-K loop, these actions finally have to be
applied to the real system in the real cloud-based environment. Pöppke and Weimann
use so-called independent wrapper scripts to call technology specific scripts on the cloud
provider’s resource containers. They also use Apache JClouds\(^4\) as a flexible middleware for

2. Foundations and Technologies

compatibility to different cloud provider APIs in between.

The wrapper scripts are Java classes which set up certain parameters and commands and then execute a technology dependent script on a resource container via the JClouds API. Such a script is running on one resource container and can only affect system components on this particular resource container. It can not apply actions to other containers. Considering the performance aspects, this means that the script consumes processing resources its container itself and, for example, it is not possible to run the scripts on their own resource container to separate their and the system’s resource consumption. In the context of his ordering algorithm, Pöppke [2017] admitted that his approach "does, however, rely on the action scripts to consider all dependencies between modules and execute the necessary actions to update them". As an alternative to the wrapper and technology dependent scrips, he discusses the use of a separate tool which might run on its own resource container and could affect several other resource containers. However, he claims that such a tool would have to be developed for each specific technology.

2.7 Building Eclipse RCP-Applications with Goomph

The Eclipse platform is designed to serve as an open tools platform and allows that its components can be used in individual client platforms as well. Therefore, the term Rich Client Platform\(^5\) (RCP) is used for the minimal set of Eclipse plug-ins needed to build a standalone RCP-application. In this thesis, we want to automatically build a RCP-application of PerOpteryx. Goomph is relevant in this context, because it allows to automatically build RCP-applications during a regular Gradle build.

Building an RCP-application usually starts with an Eclipse plug-in project. Such a plug-in can then be exported into something called a product. Besides the individual plug-in itself, such a product contains all required core plug-ins of the RCP. During the export of a product the required dependencies are resolved and stored in the product to ensure it can be executed as a standalone application. The export of such a product happens manually out of the Eclipse IDE. Build tools such as Maven or Gradle are commonly used in software production. Goomph\(^6\) is a plug-in for Gradle which allows to automate the export of a RCP-application. Goomph is also capable of other features such as generating a completely set up Eclipse IDE on build but for this thesis we only focused on its ability of building RCP-applications.

The Goomph plug-in itself is imported and set up in the build.gradle file of a project’s root


\(^6\)DiffPlug LLC. Goomph: IDE as build artifact. URL: https://github.com/diffplug/goomph (visited on 08/05/2018).
directory. The RCP application’s specific parameters uid, applicationName, pluginPrefix, pluginProjectDirectory, projectId, and productName are configured in the gradle.properties file in the project’s root directory. The compile time dependencies of the RCP-application are defined in the project specific build.gradle files as with any other Java project. This file also needs to include the RCP-application’s runtime dependencies to enable Goomph to generate the executables automatically. Goomph also allows the inclusion of different versions of a certain dependency. This is useful in more complex projects where different parts of an RCP-application require different versions of a dependency.

2.8 Drools

In this thesis, we want to use Drools\(^7\) to define rules for the computation of adaptation actions. Drools is a Business Rules Management System for the Java platform. It provides a Business Rule Engine together with a Java API to use the engine from a Java application. Drools is a product of the KIE (Knowledge is Everything) family by JBoss and the KIE keyword is present in many API components.

In this section, we only cover the parts relevant to our thesis but a detailed user guide\(^8\) can be referred for more information. The Drools libraries can be integrated to a Java project with modern build tools such as Maven or Gradle. A Drools project has the structure of a normal Maven or Gradle project with src/main/java and src/main/resources folders. In preparation of using the engine an additional kmodule.xml file has to be created in the src/main/resources/META-INF folder to define the KieBases and KieSessions that can be created. The rules can then be defined in files with the .drl extension in the src/main/resources folder or any subfolder. There are two relevant parts for using the Drools rule engine from a Java application. Firstly, the rules are defined in the .drl files mentioned before. Secondly, a KieSession has to be initialized in the Java code to use the rule engine. A session has its own working memory and facts can be inserted to it or removed from it. The facts are represented by Java objects. The rules are written in their own syntax and follow a when-then scheme. The when-part defines constraints which decide whether a rule can be applied. In the then-part actions can be defined to be executed if the when-constraints are fulfilled. These actions in the then-part are defined in Java-syntax. They can include any Java command such as print outs or modifications to objects in the working memory. Adding or removing facts is possible as well. Note that all Java classes referenced in a rule file need to be imported like in a normal Java class.

There are stateless and stateful KieSessions. The first fire all rules as soon as the facts are added and thereby modifying the working memory just once. The latter are fired explicitly

\(^7\)JBoss Inc. Drools - Business Rules Management System. URL: https://www.drools.org/ (visited on 07/05/2018).

\(^8\)JBoss Inc. JBoss Drools User Guide. URL: https://docs.jboss.org/drools/release/7.6.0.Final/drools-docs/html_single/index.html#_user_guide (visited on 06/05/2018).
2. Foundations and Technologies

from the Java code when the developer wishes to. Thus, it is possible to add some facts to the session’s working memory, fire the rules, and let them modify the working memory. Later, more facts can be added from outside and the rules can be fired again. Rules within the drools rule engine can be given weights which affect the order in which the rules are fired if multiple rules can fire due to the facts available in the working memory. Priorities can be given to rules by adding a salience value to a rule. Higher values imply a higher priority. The default salience value is 0 and positive as well as negative salience values are allowed.

2.9 Kubernetes

Kubernetes\(^9\) is an open-source system for the deployment of containerized applications. It also features mechanisms for scaling and managing such applications. Its primary purpose is container orchestration. The idea behind containerization platforms like Docker\(^10\) is to include an application in a container with all its dependencies such as binaries, libraries, JAR files, configuration files, scripts and so on. With all required dependencies available in the container, it can be run in any execution environment which is able to run the Docker engine. Technically, all relevant information is stored in a container image and the container instances are then started from that image. In applications which consist of multiple, independent microservices the different services can be packed into containers [Krochmalski 2017]. Kubernetes ensures that all containers of an containerized application run on physical or virtual machines in the managed execution environment. Additionally, it takes care of all running containers, balancing load between them, adding or removing containers to scale the application, and replacing crashed containers [Sayfan 2017].

Now we take a closer look at the concepts Kubernetes builds upon. Specifications for the different components are defined and provided in YAML files [Krochmalski 2017]. We only provide a brief overview of the following core concepts which are relevant for this thesis:

- **Pods**: Pods are the basic unit within Kubernetes. They consist of one or multiple Docker containers. Containers within a pod share the same network and disk resources and can communicate over localhost. A pod has its own IP address and each container can communicate with any other pod or service in the cluster if it knows its IP address.

- **Replica Sets**: With replica sets a certain number of running pods can be defined for a pod type. Kubernetes then starts the required number of instances automatically. If too many pods are running they will be terminated. Kubernetes also replaces pods which died to assure that the specified number of replicas is available at any time.

\(^9\)Kubernetes – Production-Grade Container Orchestration. URL: https://kubernetes.io/ (visited on 05/06/2018).
\(^{10}\)Docker. URL: https://www.docker.com/ (visited on 05/06/2018).
2.9. Kubernetes

- **Deployments**: Deployments are a higher level of abstraction than pods and replica sets. Only when a deployment has been created, Kubernetes actually schedules a pod’s defined number of replicas onto individual nodes in the cluster. Apart from that, a deployment allows to update the number of replicas or even roll back to a previous deployment.

- **Services**: Services group one or more pods together and are given their own IP addresses as well. However, while a pod can die and be replaced by another pod with a different IP address a service’s exposed IP address remains constant during the service’s lifetime. Thus, services do not lose track of each other as may happen with pods. A service also offers a load balancer which distributes workloads among the service’s pods.

- **Namespaces**: Namespaces are a grouping mechanism Kubernetes provides. They allow the isolation of different environments in the same cluster. Components of a certain namespace are not affected by adaptations to components in another namespace. Namespaces are optional in Kubernetes. The default namespace is used if no other is specified.

The hardware environment to run the described components is called a cluster. It consists of the following components [Krochmalski 2017; Sayfan 2017]:

- **Cluster**: A cluster contains all hosts, storage, and networking resources available to Kubernetes to distribute workloads among them. An execution environment may even consist of multiple clusters.

- **Nodes**: Nodes represent single hosts within a cluster no matter if it is a physical or virtual machine. They acts as worker and executes the pods which are assigned to them. A node can run one or multiple pods. Due to their worker nature, nodes were also called minions in the past.

- **Master**: The master is another node of a Kubernetes cluster. Unless the other nodes, it does not run any containers. The master is responsible for the scheduling of the pods across the cluster’s worker nodes taking into account the available resources on each node.

A common approach to the container orchestration with Kubernetes is to create specifications for the envisioned components in dedicated YAML files as mentioned before. With kubectl Kubernetes provides a command line tool itself to administrate and monitor deployments on a cluster. To access a cluster programmatically, Kubernetes provides client libraries\(^1\) for several programming languages. For Java, there is an official library, as well as two community maintained libraries by OSGi and Fabric8 at this point of time. Generally, Kubernetes is relevant for this thesis as our available cloud-based execution environment is a Kubernetes cluster. With iObserve being written in Java, a Java client library is relevant as well.

2. Foundations and Technologies

2.10 The Goal Question Metric Approach

The Goal Question Metric (GQM) approach [Caldiera and Rombach 1994] is a measurement mechanism for feedback and evaluation in the context of software development. It helps to structure a development process by defining the key elements goals, questions and metrics. Figure 2.7 shows the tree structure of the GQM with goals being refined into several questions and metrics used to answer the questions. The different key elements cover different levels of the development process:

![GQM model structure](image)

Figure 2.7. GQM model structure [Caldiera and Rombach 1994]

1. The goals are defined on a conceptual level. A goal specifies the purpose of measurement, the object to be measured, an issue to be measured, and a viewpoint from which it is taken. An example for a goal is: “Improve (purpose) the timeliness of (issue) change request processing (object) from the project manager’s viewpoint (viewpoint)” [Caldiera and Rombach 1994].

2. The questions are defined on an operational level. A set of questions characterizes the assessment of a specific goal and tries to break down the issue into its major components. Examples for questions are: “What is the current change request processing speed?” or “Is the performance of the process improving?” [Caldiera and Rombach 1994].

3. The metrics are defined on a quantitative level. They aim at answering questions in a quantitative way based on a certain data set. Such data can be objective if they only depend on the object and not on the viewpoint from which they are taken. If they also depend on the viewpoint, then the data is considered subjective. Examples for objective metrics are the average cycle speed or the standard derivation for the first question. For the second question a rating by the project manager represents a subjective metric. The same metric can be used to answer different questions.

In the context of this thesis, we use the GQM approach to structure our research.
Chapter 3

An Approach to System Adaptations in Cloud-Based Systems

The main focus of this thesis are the adaptation planning and executions phases of the MAPE-K approach [Kephart and Chess 2003; IBM 2006] introduced in Section 2.1. The iObserve approach, introduced in Section 2.4, suits this intention as its control loop for automatic adaptation resembles the MAPE-K loop. Its existing implementation for the planning and execution phases summarized in Section 2.6 has the following shortcomings:

a) A central part of planning phase is based on the call to a standalone RCP-application of PerOpteryx which has to be set up manually beforehand.
b) The planning phase does not take into account potential dependencies between components of the observed system such as inter-component communication routines.
c) The planning and execution phases are part of iObserve’s analysis service and are not provided as separate services.

We address these shortcomings in this thesis and present approaches and solutions to improve upon them. The rest of this chapter is structured as follows: First, we will present a short overview of iObserve’s model-based approach and the service-based architecture we address in this thesis. We shortly introduce some details concerning the data flow between the services in the following section. The last three sections in this chapter attend to the computation of architecture candidates, the computation of execution plans, and their execution. They introduce the concepts behind these phases and explain what is going to happen in each phase.

3.1 Overview of the Adaptation Approach

In this section we present an overview of our approach. Even though iObserve generally follows the MAPE-K approach some details need to be clarified. Within iObserve, the K (knowledge) is represented by architecture models. During iObserve’s analysis phase a model representing the observed system is created. We call this first model the present-architecture-model (PAM). Given a PAM, we want to compute a candidate model representing an architecture which improves upon the given issues. We call this second model the candidate-architecture-model (CAM). We want to redeploy the real architecture’s components in a way which is conform to the CAM. The process of computing a CAM from
An Approach to System Adaptations in Cloud-Based Systems

The PAM is called planning. The actual redeployment process is called execution. For this process, we have to compute concrete transformation steps, so-called adaptation actions, to reconfigure the software system to conform to the CAM. These actions are derived on a model basis meaning that they represent the transformation steps needed to transform the PAM into the CAM. However, executing these actions on the real system will also transform the real system’s present architecture into its planned architecture. Ideally, we are able to perform the actions in an order which allows the system’s adaptation without compromising its availability. Such a sequence of ordered adaptation actions is called an execution plan.

We subdivide the planning phase into the generation of candidate-architecture-models and the derivation of execution plans. Subdividing the planning phase enables us to separate the both parts into independent services. The first is implemented in the planning service while the latter is implemented in the adaptation service. The execution stays a single phase represented in the execution service. Providing a microservice architecture with separate services for the different phases increases iObserve’s modularity. It also helps to reduce the complexity of architecture adaptation problems by splitting them up and facilitates the implementation of alternative strategies in the future. Figure 3.1 shows an overview of the different services and the data passed between them via TCP connections. We will refer to it in detail in the following paragraphs. The rest of this section includes a brief overview of the three planning and execution phases, its functions, inputs, and outputs. Further details are discussed in the corresponding paragraphs while implementation details are explained in Chapter 4.

![Figure 3.1. Phases of iObserve’s control loop split up into services](image-url)
3.2 Data Flow Between the Services

Generating a Candidate Architecture The first part of the planning phase shown in Figure 3.1 receives the present-architecture-model as input via a TCP connection. In this phase an optimization algorithm optimizes the PAM with respect to certain optimization goals. This results in a candidate-architecture-model. In this thesis, we use PerOpteryx for the optimization because it is compatible to the PCM models used by iObserve. PAM as well as CAM are then sent to the following phase via TCP connections.

Deriving an Execution Plan The second part of the planning phase shown in Figure 3.1 receives PAM and CAM as input via TCP connections. The adaptation actions are computed from the difference between both models. We use a rule based approach based on Apache JBoss Drools to compute the actions in this thesis. In Section 3.4 we point out how we use adaptation actions of different levels to finally create an execution plan. In general, we compute certain adaptation actions and place them in a certain order. This order is highly important to improve the observed system’s availability during its adaptation. The ordered actions are then refined to address possible dependencies between system components. The refined actions form the execution plan which is passed to the next phase via a TCP connection. PAM and CAM are also passed to the next phase because they are referenced by the actions in the execution plan.

Executing an Execution Plan The execution phase shown in Figure 3.1 receives the execution plan, as well as the PAM and CAM as input via a TCP connection. We want to apply the actions included in the plan to the observed system. The actions themselves are independent from concrete cloud environments. Therefore, we need a compatible interface to a certain cloud environment. In this thesis, we provide an implementation for cloud-based systems realized with Kubernetes. Executing the actions from the execution plan finally transforms the observed system’s architecture into its intended state.

3.2 Data Flow Between the Services

The different services have to propagate their results to the following services within iObserve’s control loop in some way. We take a closer look at what kind of data is propagated between the different services: The analysis service passes the updated PAM to the planning service. The planning service passes the received PAM as well as the optimized CAM to the adaptation service. Once received, the adaptation service uses the models to compute an execution plan which is based on an Ecore model as well. This execution plan model is then passed to the execution service together with PAM and CAM. We can see that all services sent Ecore models to their respective successor service. Ecore models can be serialized into XML-based files which means that the model propagation can be seen as a number of file transfers.
3. An Approach to System Adaptations in Cloud-Based Systems

3.2.1 TCP Writer and Reader Stages

As all cases of model propagation mentioned above require a way to pass files from one service to the next one, we are able to implement reusable TCP file writer and reader components for this task. We used iObserve’s already existent MultipleInputTcpReaderStage as a blueprint for their implementation. This stage receives monitoring records in iObserve’s analysis service from multiple connections using java.nio’s socket channels. We only need a single point to point communication between our writer and reader. In conformance with the existing reader we also use the java.nio socket channels for communication. Both components are implemented as filters using the TeeTime framework because we aim for a Pipe-and-Filter architecture within the different services with the reader as its starting point and the writer as its ending point.

The SingleConnectionTCPWriterStage can connect to a correspondent reader with a given hostname and port. The correspondent reader can be part of the next service in iObserve’s control loop. The writer extends TeeTime’s consumer stage and receives an arbitrary file on its input port. This file can be the serialized architecture model. Once a channel to the reader is created, this filter sends the following parts, as depicted in Figure 3.2: 1) Four bytes (an integer) containing the number of bytes required for the file name, 2) The file name in bytes, and 3) the file itself as bytes before closing the channel. Once another file is received on the filters’ input port, a new channel is created.

<table>
<thead>
<tr>
<th>Filename length n as int</th>
<th>Filename as bytes</th>
<th>File content as bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>n+4</td>
</tr>
</tbody>
</table>

Figure 3.2. Scheme of bytes sent with the SingleConnectionTCPWriterStage

Sending the filename is important to ensure that the different PCM submodel types can be distinguished in the receiving service. The number of bytes for the filename has to be sent to tell the correspondent reader which bytes belong to the filename and which belong to the file’s content. Thus, we create a simple protocol enabling the reader to decode the incoming bytes into the filename and the file itself.

The SingleConnectionTCPReaderStage provides a server socket channel available to the writers via its hostname and port. The filter extends TeeTime’s producer stage which means it is declared active by default and constantly listens to incoming files. Once a writer’s request to the server socket channel is accepted, the filter creates a socket channel to receive the incoming data. As specified by the writer, the reader reads 1) the four bytes with the length n of the file name, 2) the n bytes containing the file name, and 3) the bytes containing the file. The received file is stored in a directory specified in the filter’s constructor before the channel is closed.
3.3. Generating a Candidate Architecture

3.2.2 Model Collector Stages

With PCM models consisting of different submodels stored in different files we also need a way to first collect all model files before continuing within the service. On the other hand, having a directory of different submodel files we need a way to send them one after another to the following stages. We implemented TeeTime filters for these functionalities as well:

- The **ModelDir2ModelFilesStage** receives an directory on its input port and sends the included PCM submodel files to its output port. It is implemented as a composition of TeeTime’s predefined `Directory2FilesFilter` and `FileExtensionSwitch`. This internal architecture enables us to define that only files with certain extensions are passed to the output port. In our case, we ensure that only PCM submodel types are read from the directory. We generally use the `ModelDir2ModelFilesStage` before a `SingleConnectionTCPWriterStage`.

- The **ModelFiles2ModelDirCollectorStage** receives files on its input port and checks their file extensions. When files from all required PCM submodel types have been received, this filter passes the parent directory to its output port. To keep this filter as simple as possible we assume that all received files are in the same parent directory. Once models from all submodel types have been received, the filter will send the parent directory to its output port when another submodel file is received on the input port. This behavior is intended as it enables us to receive updates to only one certain submodel without having to always receive all submodels again. We generally use the `ModelFiles2ModelDirCollectorStage` after a `SingleConnectionTCPReaderStage`.

3.3 Generating a Candidate Architecture

In the first part of the planning phase a model of our observed system’s present architecture, the PAM, is available. The goal of this phase is to improve the given architecture with respect to certain optimization goals such as performance [Pöppke 2017] or privacy [Weimann 2017]. This optimization then results in a new architecture candidate which is represented by the CAM in iObserve. In this thesis, the optimization is performed by PerOpteryx by applying its evolutionary algorithm iteratively to the PAM and thereby transforming it to the CAM.

Optimizing a software system’s architecture is an extremely complex task as there are several, often conflicting, parameters influencing this task. For example, one way to improve a system’s responsibility is to increase the amount of available hardware resources. However, this would come at the cost of increased operating expenses which are also limited in an enterprise environment. A common term in this context is the Pareto-optimum which defines a state in which it is possible to improve on a given property while not degrading
3. An Approach to System Adaptations in Cloud-Based Systems

any other property of a given set of properties [Pöppke 2017]. The PerOpteryx Eclipse plug-in presented in 2.5 focuses on the generation of Pareto-optimal candidates of PCM models and was used by Pöppke [2017] and Weimann [2017] for their approach.

3.4 Deriving an Execution Plan

An execution plan consists of a sequence of adaptation actions which transform the observed system’s present architecture into its intended state represented by the CAM. We mentioned that a certain order is required to perform such a transformation without compromising the systems availability. But how can a different order actually affect the systems availability? We take a closer look at the interaction of different system components and particularly their dependencies to answer this question. We present reasons for runtime reconfigurations and take a look at influences to the reconfiguration such as component states and dependencies between components. We also take a look at different approaches to the computation of execution plans.

3.4.1 Reasons for Runtime Reconfiguration of Cloud-Based Systems

The aim for reconfiguration at runtime results from the fact that in business applications availability is an important factor. During any downtimes of a system customers are potentially lost to competitors. We already discussed the need for dynamic adaptation to reduce operating costs, as well as ensuring the systems responsibility. Ideally, the system stays available at all times while the adaptations take place. For this reason, it is necessary to adapt the system at runtime. Systems with microservice architectures fit the given scenario well. An architecture consisting of different microservices is usually more robust and can be scaled by deploying multiple instances of a component [Hasselbring and Steinacker 2017]. The software infrastructure also evolves in this regard and more frameworks such as Kubernetes or Docker Swarms provide advanced infrastructure controls such as automatic load balancing, for example.

3.4.2 Influences to the Reconfiguration

There are two things we have to consider while rearranging the system components: a) Components may have an internal state and b) components may communicate and therefore depend on one another.

Component State A component’s state may be defined by several aspects. The common sense of a state implies that certain data is retained and not discarded after a single computation cycle. The kind of data however is highly specific to the concrete system component. This may reach from a simple counter variable counting the overall number of computation cycles to complex data records stored on a storage medium. Even a database may represent
a component’s state. However, depending on a system’s architecture databases may also be considered as individual components. Components which are added to a running system can not always assume that the system is in its initial state. They might need to discover the system’s state and synchronize its internal state with the system’s state [Oreizy et al. 1998].

Examining a component at a certain point in time can lead to another aspect of component states. Is the component performing a computation at the time of adaptation execution or does it wait for incoming requests, for example? We do not want to abort running transactions by reconfiguring the system as this would result in incomplete or incorrect system behavior. We have to either ensure that aborted transactions are finished or repeated after the reconfiguration. Alternatively, we have to assume that at certain points in time reconfigurations may have to be wait for running transactions to finish. Oreizy et al. [1998] state in this context that a system’s runtime environment may prohibit adaptation actions while any of a component’s function are on the execution stack. They generally propose system architectures which are designed to tolerate sudden loss of functionality or state.

Component configuration parameters such as certain ports or addresses to listen to or receive data from are also part of a component’s state. However, such parameters possibly referring to other components may change after reconfiguration as we will discuss in the next paragraph.

Dependencies between Components Component-based systems include different components communicating with each other. A component may receive requests from other components as well as computing and providing data to other components itself. The PCM models these dependencies by using roles. A component’s interfaces are separated into provided and required interfaces. The former define those interfaces which are provided to other components and the latter are required from other components. Thus, the requiring component may be able to post requests via the providing component’s interface.

If one of the involved components is affected by the system adaptation, chances are that the other involved component(s) will also be affected in some way. For example, component A provides data to component B which can send requests to A via a TCP connection. If the system is reconfigured in a way which moves component A onto another resource container, the TCP connection B uses for its requests also needs to be reconfigured. Figure 3.3 depicts such a situation: Even though only one component (A) was directly affected by the system adaptation, the other component (B) had to be reconfigured as well. Instead of posting requests to A via localhost, B now needs the IP address of A’s new resource container after the adaptation.

A connection between two components may even have to be maintained during the reconfiguration. This depends on the cloud-based system’s architecture and especially on
3. An Approach to System Adaptations in Cloud-Based Systems

![Diagram showing reconfiguration efforts between dependent components](image)

**Figure 3.3.** Example for reconfiguration efforts between dependent components

how tight or loose the depending components are coupled. Loosely coupled systems such as microservice architectures are more likely able to compensate for a connection issue due to the system’s reconfiguration. In tightly coupled systems even a short connection issue due to reconfiguration may result in errors and misfunctions of the system. In this case, more emphasis has to be put on the smooth reconfiguration of the connections. Thus, an implementation has to be very specific to the cloud-based system which is why we focus on loosely coupled systems in the implementation part of this thesis.

### 3.4.3 Methods to Compute an Execution Plan

There are several ways to compute an execution plan. One approach is to start developing the execution plan during the planning phase. It seems intuitive to log the planning algorithm’s decisions on potential system adaptations and use these model-based data to derive the actual execution plan. Pöppke [2017] discusses this approach in his thesis and states that in the case of PerOpteryx the relevant information could be derived from the generated design decision file. However, he also states that such an approach was too specific to PerOpteryx as an optimization algorithm.

Instead, Pöppke and Weimann implemented the computation of an execution plan based on the comparison of PAM and CAM as presented in Section 2.6. This approach leads to additional computation effort as a result of the comparison. It leaves the available data from the PerOpteryx design decision file unused. However, an arbitrary model-based planning algorithm as PerOpteryx does not necessarily consider all dependencies of components in a real system discussed in the previous section. Simply transferring the planning algorithm’s adaptation steps to the execution plan and therefore to the real system might even compromise the system’s functionality and availability. This is because on a model basis, the availability of a cloud-based application is not relevant. The planning algorithm may just propose to migrate a component from one resource container to another, for example. Yet it would not consider if the component is available all the time during the migration. On a model level a migration is simple. No connections need to be established, no components need to be started up or shut down only the model has to be modified. This is not the case in a real system.
3.5 Executing an Execution Plan in a Concrete Cloud Environment

Once an execution plan has been computed, it has to be applied to the observed system thereby transforming it to conform with the CAM computed earlier. While all parts of iObserve’s control loop described so far are independent from the cloud infrastructure the observed system runs on, this is not the case with the execution phase. As we do not work on a model level but on the real cloud-based system in this phase the methods to access and configure the cloud infrastructure are key. An implementation of the execution of system adaptation actions is highly dependent on the interfaces a cloud vendor provides to configure its cloud infrastructure. An execution plan may even contain actions which are not supported by a certain cloud provider’s adaptation interface at all because they are handled internally, for example. Different providers may also use different levels of abstraction. On one platform a user might get access to every virtual machine and is able to deploy component individually, for example. On other platforms such as Kubernetes, this might not be the case because they use a higher level of abstraction. They take away the task of deploying individual component instances on individual machines from the user and provide other configuration mechanisms such as the definition of requirements for component types [Sayfan 2017]. The execution phase has to be capable of dealing with scenarios like this.

Another aspect risen by the execution is the question whether executed adaptations transform the real system in the expected way. In complex systems where we have to consider component states and dependencies between components, unexpected system configurations can result from the execution. The observed system may also have already changed before the execution plan has been executed resulting in an obsolete plan. Robust cloud-based systems have an edge here but additional strategies for dealing with unexpected adaptation results can also help in this regard. Such strategies may incorporate the supervision of the execution of system adaptations, the creation of recovery points, or the step-wise adaptation on a redundant copy of the system [Rohr et al. 2006].
Title: Architecture Evolution

In this chapter, we present the evolution iObserve’s planning and execution architecture has undergone during the course of this thesis. We started with the existing planning and execution phase’s implementation and evolved it to improve upon the existing basis. This chapter presents the evolution of the generation process of candidate architectures (Section 4.1), the development process of execution plans (Section 4.2), and the execution process of system adaptations (Section 4.3). Each section is structured as follows: First, we describe the situation in the existing approach and point out areas for improvement. We then explain which part we want to improve in particular and the reasons for the improvement. We outline the approach we want to use as well. Afterwards, we present our implementation. Finally, we reflect upon our implementation and whether it was able to achieve the intended improvements.

4.1 Generation of Candidate Architectures

In Section 3.3 we presented the approach of generating candidate architectures improving upon the present architecture on a model level. We already mentioned that the existing approach uses the PerOpteryx Eclipse plug-in for this task. In this section, we focus on the integration of PerOpteryx into iObserve and present our evolution approach.

4.1.1 Candidate Generation in the Existing Approach

Being an Eclipse plug-in, PerOpteryx is intended to be used from within the Eclipse IDE and configured via a customized run configuration dialog. To use PerOpteryx in an automated scenario like iObserve’s control loop Pöppke and Weimann had to create a standalone RCP-application to set up and start PerOpteryx. This application named peropteryx.plugin is available as an Eclipse plug-in project within the iObserve project. The required executables have to be generated manually, though. This is done by exporting the plug-in project as a product from its .product file within the Eclipse IDE. Once generated, the executables can be called from iObserve’s source code. iObserve has to be given the location of the executables beforehand, to locate them.

Problems with the described existing approach are a risk of misconfiguration by human
4. Architecture Evolution

error as well as a potential platform dependency: A user of iObserve might not notice that he has to manually export the RCP-application as a product and then tell iObserve the location of the generated executables. The PerOpteryx RCP-application depends on more than 400 other plug-ins. These dependencies have to be resolvable to ensure that the RCP-application can be exported. We determined that resolving all dependencies is another problem depending on the operating system as well as the Eclipse version. While Pöppke and Weimann seem to have used Eclipse Neon on Windows we encountered problems using Eclipse Oxygen on MacOS. One problem could only be solved by manually replacing certain dependencies specific to the Windows file system by their correspondent MacOS specific equivalents. Another problem seemed to result from the different Eclipse versions used and could only be solved by manually changing the versions of certain dependencies.

Given the situation above, the setup process of the existing approach is complicated and error-prone. Being based on the manual export of an Eclipse RCP product, it does not support techniques like continuous integration. In the next subsection, we describe how we approach these problems with an evolution of the existing approach.

4.1.2 Outline for the Evolution of Candidate Generation

Integration as a Dependency  Our first approach to improving the integration of PerOpteryx is based on the fact that every Eclipse plug-in basically consists of a number of .jar archives. All of the plug-in’s logic is contained in these archives and PerOpteryx is no exception in this regard. The optimization plug-in’s core logic is contained in 20 jar-archives with the prefix `de.uka.ipd.sdq.dsexplore`. On top of that there are more than 400 dependencies overall to the Eclipse platform and other plug-ins. We assume that this does not effect our approach because every dependency is another jar-archive which can be added to iObserve’s build path. Thus, instead of running PerOpteryx as an Eclipse plug-in or standalone RCP-application we try to run it programmatically from within iObserve. Note that this approach still sees PerOpteryx as a black box and tries to integrate the complete tool into iObserve. Therefore, we assume that all runtime components required by PerOpteryx can be setup programmatically from iObserve only by referencing the provided archives. The Eclipse workspace is an example for such a runtime component which has to be set up beforehand as it is used to store the PCM models during the optimization process. Being able to start PerOpteryx programmatically, would enable us to replace the need for manually exporting an RCP-application with all its risks of errors. On top of that, all required dependencies could be imported during the build and we would be able to provide a fixed execution environment for PerOpteryx.

Automated Integration  Another approach to improve the integration of PerOpteryx is to automate the RCP-application’s export process. As described in the previous section, the manual export is error-prone and platform dependent. By automating this process we
4.1. Generation of Candidate Architectures

could ensure that the correct executables for each operating system were created. We could also specify a fixed location where these executables had to be located within the iObserve project. Integrating these actions into iObserve’s build process ensures that everything is set up when the build finishes successfully. Possible problems during the export would cause the build to fail. This prevents a user of iObserve from executing the adaptation loop when PerOpteryx is not set up properly.

4.1.3 Implementation of Candidate Generation Evolution

An Attempt to the Integration of PerOpteryx as a Dependency  We tried to run the PerOpteryx plug-in from iObserve directly, by integrating the existing PerOpteryx headless plug-in’s code into iObserve’s code. We tried to provide the required eclipse core libraries as well as the required PerOpteryx and Palladio libraries. A full list of used libraries can be found in Appendix A.2. We tried to start and configure the Eclipse platform programmatically. This is not a common approach but this way we would have been able to configure and use PerOpteryx directly from iObserve without the need of configuring an additional Eclipse RCP-application. Listing 4.1 shows our approach. In line 2, we used the startup method of the EclipseStarter class. This class is intended to start the Eclipse platform. In line 11 we tried to manually start the ResourcesPlugin which usually initializes the workspace. The existence of a workspace was mandatory because it was used to create a working directory for PerOpteryx. In line 5 and 6 transitive dependencies for the ResourcePlugin were started. However, an internally needed RegistryProvider still remained uninitialized and we were not able to initialize it sufficiently with an own implementation in line 8.

```
// Start Eclipse programmatically
final BundleContext context = EclipseStarter.startup(new String[0], null);

// Required for initialization of ResourcePlugin
new Activator().start(context);
InternalPlatform.getDefault().start(context);
RegistryFactory.setDefaultRegistryProvider(new MyRegistryProvider());

// Initializes workspace
new ResourcesPlugin().start(context);
```

Listing 4.1. Attempt to start Eclipse programmatically

In an alternative attempt, we tried to install and start the required OSGi bundles programmatically. Therefore we tried to use the install bundle method on the context object from Listing 4.1 for the jar including the ResourcesPlugin in line 4-7 of Listing 4.2.
4. Architecture Evolution

The bundle was then started with the `start` method in line 9. This attempt did not solve our problem with the not initialized `RegistryProvider` either.

```
// Transitive bundles for org.eclipse.core.resources were installed here

// The Bundle including the ResourcesPlugin we're actually interested in
Bundle bundle = context
    .installBundle(
        new File(".../org.eclipse.core.resources_3.12.0.v20170417-1558.jar")
            .toURI().toURL().toString());
bundle.start();
```

Listing 4.2. Manual installation of osgi bundles

As we were not able to initialize a functional platform for running PerOpteryx this way, we decided to implement an Eclipse RCP-application for the execution of PerOpteryx. Such an implementation is close to the headless PerOpteryx plug-in Pöppke [2017] and Weimann [2017] developed.

The Integration of PerOpteryx as a RCP-Application using Gradle  

After all approaches of integrating PerOpteryx into iObserve directly failed, we decided to stick to the implementation of a standalone version of PerOpteryx as introduced by Pöppke and Weimann [Pöppke 2017; Weimann 2017]. They already implemented a Eclipse RCP-application which sets up and executes the actual PerOpteryx Eclipse plug-in. They provided their approach as an additional plug-in project called `peropteryx.plugin`. It was up to the developer to import this project into their Eclipse workspace and generate the platform specific executables from the product file. For simplifying the set up process and improving the portability and platform compatibility, we created a copy of the `peropteryx plugin` project and called it `peropteryx.rcp`. We now use Gradle to build the `peropteryx.rcp` project together with all other sub-projects of the iObserve approach. Thus, the developer does not need to manually import an additional plug-in project into their workspace.

The PerOpteryx RCP-application itself depends on 252 additional plug-ins. With these plug-ins introducing additional dependencies we end up with 414 necessary dependencies for the execution of the PerOpteryx RCP-application. All dependencies are defined in the application’s `.product` file or, for debugging purposes, in a correspondent Eclipse run-configuration created by the developer. Creating the platform specific RCP-application’s executables still has to be performed manually from the `.product` file out of the Eclipse IDE. This is due to the fact that Gradle does not support the building of Eclipse RCP-applications per default. For this reason, we tried to use the Goomph Gradle plug-in described in Section 2.7. With the Goomph plug-in, using the Gradle target `assemble.all` while building the iObserve project is supposed to automatically generate PerOpteryx executables for
4.1. Generation of Candidate Architectures

Windows, Mac OS, and Linux in the deploy subproject.

The Goomph plug-in itself is imported and set up in the build.gradle file in the root directory of the iObserve project. Additionally, the RCP-application’s specific parameters uid, applicationName, pluginPrefix, pluginProjectDirectory, projectId, and productName are configured in the gradle.properties file in the root directory of the iObserve project. The compile time dependencies of the RCP-application are defined in the project specific build.gradle file. Additionally, the build.gradle file needs to include the RCP-application’s runtime dependencies to enable Goomph to generate the executables which had to be performed manually from Eclipse before. For some dependencies multiple versions are required. All versions have to be listed in the equinoxLaunch section of the build.gradle file in iObserve’s root directory. All dependency versions except for the newest also have to be specified in the dependencies section in the build.gradle file of the target.maven project. The reason for this is that Gradle otherwise ignores the older dependency versions.

Although we were able to use Gradle together with Goomph to generate executables for a minimalistic “Hello World” application, the generation failed for our PerOpteryx RCP-application. Therefore, we were not able to get rid of the manual generation of executables at this point of time. As we were not able to integrate PerOpteryx into iObserve directly, we still have to use calls to these executables from the OptimizeModel filter. By using Gradle and Goomph to build the project and generate the executables, we would have ensured that they are actually present. But this way, the risk of the error prone, manual generation of executables remains necessary. However, even a correctly exported PerOpteryx which can be invoked from iObserve terminates with a runtime exception. This means that even though PerOpteryx was integrated, we are not able to actually use it to generate a CAM.

The Planning Service in iObserve  The planning phase is implemented as its own service within iObserve’s service oriented architecture. Its starting point is the PlanningMain class in the planning package. The Pipe-and-Filter architecture of the planning service is shown in Figure 4.1. The PAM model files are received via a TCP connection in the SingleConnectionTCPReaderStage and collected in the ModelFiles2ModelDirCollectorStage. The ModelProcessing filter computes the allocation groups introduced in Section 2.6. It was refactored during this thesis. Thus, the filter itself now includes the computation of allocation groups and does not refer to an additional class outside the Pipe-and-Filter architecture for the transformation. Once the models include the allocation groups, the model files are passed to the ModelOptimization filter. This filter copies the processed PAM model files and calls the PerOpteryx application with the directory containing the copied files. Thus, the copied files are transformed into the CAM. Once PerOpteryx has finished its optimization, the filter passes the directory containing the optimized CAM to its dedicated output port. The PAM directory is forwarded to a second output port. Both ports are followed by an ModelDir2ModelFilesStage. This filter sends all PCM submodel files to the
4. Architecture Evolution

SingleConnectionTCPWriterStage which sends them to the adaptation service.

![Diagram of the Pipe-and-Filter architecture of iObserve's planning service.](image)

Figure 4.1. Pipe-and-Filter architecture of iObserve's planning service

4.1.4 Results of the Candidate Generation’s Evolution Process

In this section, we presented two approaches to the integration of PerOpteryx into iObserve. The first approach was not successful because we were not able to programmatically set up all parts of the Eclipse platform required for the execution of PerOpteryx. The presented approach is no common practice and the complexity of the Eclipse platform itself made debugging more difficult. Given the small amount of information available on the approach we did not have more time for debugging during the course of this thesis.

In our second approach, we wanted to use PerOpteryx as a RCP application. We tried to use the Goomph plug-in for Gradle to automatically generate the required executables but we failed. We were able to invoke PerOpteryx with manually exported executables but it crashed shortly afterwards due to a runtime exception. We did not have the time for further investigations. However, except for the problems with PerOpteryx, we were able to separate the planning routines into the planning service. Additionally, the PerOpteryx RCP application can now be built and imported using Gradle and does not have to be imported to the iObserve project manually.

4.2 Computation of Execution Plans

In Section 3.4, we have taken a look at the runtime reconfiguration in cloud-based systems and seen potential adaptation scenarios. Now, we focus on providing an implementation of the execution plan creation taking into account these scenarios. We present our evolution of the existing approach in this section.
4.2. Computation of Execution Plans

4.2.1 System Adaptations in the Existing Approach

The seven different adaptation actions used in the existing approach were defined by the system-adaptation-model presented in 2.6. They were computed by the comparison algorithm shown in Algorithm 1 we presented in that section. Afterwards, they were ordered based on their type to address the cloud-based system’s availability. There was no dedicated execution plan in the existing approach. The computed adaptation actions were stored as a list of assembly context actions respectively a list of resource container actions in the AdaptationData record. Therefore, this record represented an execution plan although it contained much more additional information. Besides the adaptation actions, the record also contained the names of the execution scripts, as well as the EMF URIs to reference the runtime and redeployment model. It also contained an EMF URI to the folder containing the execution scripts and lists containing new and removed allocation contexts and resource containers. The record was passed from filter to filter and continuously filled with more information.

A problem with the described existing approach is that Pöppke [2017] and Weimann [2017] assumed that there are no dependencies between different components of the cloud-based system. The presented order based on the types of actions ensures for example that resource containers are allocated before components are deployed, undeployed, or migrated. This ensures that all resource containers from the CAM are available before one deploys, undeploys, or migrates components on them in the real system. However, on this level of abstraction it is not taken into account what happens if components depend on each other. The system’s availability can be affected in this case.

Another aspect which can be improved in the existing approach is the nested comparison algorithm. Using a dictionary which maps assembly context ids to the runtime components from the model’s graph representation assumes that assembly contexts represent deployed component instances. This is wrong. Allocation contexts should be used instead of assembly contexts because the latter only represent a system architecture’s assembly but not deployed instances. Besides this issue, the code become difficult to read with the nested if-then-else conditions for the detection of actions. Adding additional conditions to detect more advanced actions in the future even increases this problem.

The lack of a dedicated execution plan represents a third area of possible improvement. Even though the AdaptationData records act as an execution plan, their multi-purpose use is not ideal. Too many different information are mixed in the record and it is difficult to see in which filter of iObserve’s control loop which of the record’s attributes is set.
4. Architecture Evolution

4.2.2 Outline for the Evolution of Execution Plan Computation

We address the problem of dependencies between different components with an approach building upon two phases. In the first phase, we detect higher level adaptation actions just as in the existing approach. In a second phase however, we define a sequence of lower level actions to execute a higher level action. These lower level actions will also take into account possible dependencies to other components. We arrange them in an order which ensures the system’s availability.

Instead of using the comparison algorithm from the existing approach we use a rule-based comparison approach. A rule based system enables us to replace the nested if-then-else conditions from the algorithm and by equivalent rules. The dictionary can be replaced as well. The idea is that we simply insert the components from the PAM’s and CAM’s graph representation into the rule engine and receive a system-adaptation-model containing a list of ordered higher level adaptation actions. This approach is more extensible as additional rules can easily be added in the future, whereas this becomes more difficult with each additional statement in nested if-then-else statements. Additionally, we fix the issue that assembly contexts were used to represent deployed instances instead of allocation contexts.

We introduce a dedicated execution plan. Instead of passing the existing AdaptationData records from filter to filter and adding more information, we adapt the existing Pipe-and-Filter architecture in a way that only the relevant data is forwarded from one stage to the next. The execution plan is created at the very end of iObserve’s adaptation service and then sent to the execution service. We implement the execution plan as a PCM model to ensure that it is serializable. Given the fact that the plan is sent from one service to another, this is an important property.

4.2.3 Implementation of the Execution Plan Computation Evolution

Composed Adaptation Actions  Van Hoorn et al. [2009] and Von Massow et al. [2011] proposed five adaptation actions: Replication, dereplication, migration, allocation, and deallocation for the adaptation of component-based software systems. In their theses, Pöppke and Weimann introduced the seven adaptation actions allocation, deallocation, migration, change of repository component, acquisition, termination, and replication [Pöppke 2017; Weimann 2017]. In Section 2.6, we already determined that these actions roughly match the ones van Hoorn proposed but there are differences which we examine in this section.

One difference is the terminology. Pöppke’s and Weimann’s allocation and deallocation refer to the deployment and undeployment of software component instances. They are semantically represented by van Hoorn’s replication and dereplication of a software component instance. A replication however, goes beyond the simple deployment of a new instance which could be thought of as the instantiation of a plain new Java Object from a
4.2. Computation of Execution Plans

class. It also implicates that a certain component instance with all its connections to the rest of the system and potentially even its internal state is replicated. In the analogy of Java Objects this means that the newly created Object’s attributes and even its references to other objects are initialized in conformance with an existing object. The migration is defined similar in both approaches and represents the action of moving a component instance from one machine to another. The change of a repository component does not exist in van Hoorn’s terminology. It represents the action of exchanging an existing component instance by an instance of a different but equivalent component. Pöppke’s and Weimann’s acquisition and termination refer to the allocation and deallocation of machines. They are semantically represented by van Hoorn’s and von Massow’s allocation and deallocation actions. Finally, Pöppke’s and Weimann’s replication action refers to the action of cloning an entire machine with all its deployed component instances. It is not directly represented in van Hoorn’s and von Massow’s terminology but can be created as the allocation of a new machine followed by the replication of all deployed components of an existing machine onto the new one. Table 4.1 shows a comparison of the two different terminologies.

Table 4.1. Comparison of the different adaptation action terminologies

<table>
<thead>
<tr>
<th>Pöppke [2017]; Weimann [2017]</th>
<th>Van Hoorn et al. [2009]; Von Massow et al. [2011]</th>
</tr>
</thead>
<tbody>
<tr>
<td>De-/Allocation</td>
<td>De-/Replication</td>
</tr>
<tr>
<td>Migration</td>
<td>Migration</td>
</tr>
<tr>
<td>Change of Repository Component</td>
<td>-</td>
</tr>
<tr>
<td>Acquisition/Termination</td>
<td>De-/Allocation</td>
</tr>
<tr>
<td>Replication</td>
<td>-</td>
</tr>
</tbody>
</table>

We decided to use van Hoorn’s and von Massow’s terminology in our approach. One reason for this decision is their definition of a replication taking into account both the newly deployed component and the replicated component as discussed above. Another reason for us to choose their terminology is that the terms allocation and deallocation are commonly used with reference to hardware resources and we prefer staying with well-established terms in this context. We also add the change of repository component introduced by Pöppke and Weimann to our set of adaptation actions. There exist fitting use cases in real systems and as iObserve uses PCM models which natively support the exchange of equivalent repository components this is reasonable. The composed adaptation actions we choose for our thesis are listed below.

Repetition and Dereplication describe the process of deploying or undeploying another instance of an already existing component. In Figure 4.2, we can see that the component $C_2$ in the system state on the left side is deployed on both resource containers on the right side. From left to right, this depicts a replication and from right to left a dereplication. For a replication, connections to other components and potentially the component instance’s state are replicated as well. For a dereplication, we assume
4. Architecture Evolution

that the state of the dereplicated component instance can be discarded and incoming requests to this component instance are blocked [Von Massow et al. 2011].

Figure 4.2. Replication (→) and dereplication (←) of a component instance

Migration describes the process of moving a component instance from one resource container to another. Figure 4.3 depicts how the component C2 is migrated between the left and the right resource container. We can think of it as a replication followed by a dereplication meaning that a migration's impacts concerning connections and the migration of the component instance’s state are the same [Von Massow et al. 2011].

Figure 4.3. Migration of a component instance (↔)

Change of Repository Component describes the process of exchanging a component instance with an instance of an semantically equivalent component. In Figure 4.4 we can see that the C2 component from the system state on the left side is replaced by a component C2’ in the system state on the right side. The old component instance’s connections and state need to be reestablished for the new component’s instance [Pöppke 2017; Weimann 2017].

Figure 4.4. Change of a repository component instance (↔)
4.2. Computation of Execution Plans

- **Allocation** and **Deallocation** describe the process of allocating a new container for component deployment or deallocating an existing resource container [Von Massow et al. 2011]. Figure 4.5 shows an allocation of the resource container $N_{m+1}$ from left to right and a deallocation is depicted from right to left.

![Figure 4.5. Allocation (→) and deallocation (←) of a resource container](image)

We updated the system-adaptation-model according to this terminology. Figure Figure 4.6 shows the updated model. Besides the new terminology, we also updated the attributes of the actions. These updates are also highlighted in the figure below: Newly added attributes are marked with an asterisk symbol (*) while removed attributes are crossed out. The modifications ensure that the relevant information for dealing with dependencies between components is available in the model. We list our modifications and the reason why we applied them for each of the composed actions below. The indentation represents the model’s class hierarchy.

- **Composed Action**: We made this class abstract because we only allow the concrete actions introduced above.

  - **AssemblyContextAction**: We removed the `sourceAssemblyContext` attribute and replaced it by the `targetAllocationContext` attribute. By using an allocation context, we are able to receive the AssemblyContext as well as the ResourceContainer of the affected component instance. We use the ‘target’-prefix because the attribute represents the action’s target and not its source. We also added the attributes `targetProvidingAllocationContexts` and `targetRequiringAllocationContexts`. These are lists of allocation contexts providing an interface to the affected component instance respectively requiring one from it. They are needed to identify depending and dependent components of the affected component instance. We made this class abstract because we only allow concrete actions and the AssemblyContextAction is only a supertype containing common attributes.

  - **ReplicationAction**: We removed the `newAllocationContext` attribute because it is represented by the `targetAllocationContext` attribute of the action’s parent class. Instead, we added the `sourceAllocationContext` attribute. It represents the component which is actually replicated by this action. A replication’s source component needs to be referenced because the source components state might have to be replicated as well.
4. Architecture Evolution

- **DereplicationAction**: We removed the `oldAllocationContext` attribute because it is represented by the `targetAllocationContext` attribute of the action’s parent class.

- **MigrationAction**: We removed the `newAllocationContext` attribute because it is represented by the `targetAllocationContext` attribute of the action’s parent class. We kept the `sourceAllocationContext` attribute which represents the component before its migration. Additionally, we added `sourceProvidingAllocationContexts` and `sourceRequiringAllocationContexts`. They list the allocation contexts providing an interface to the affected component instance respectively requiring one from it prior to the migration.

- **ChangeRepositoryComponentAction**: We removed the `newRepositoryComponent` because it can be accessed via the `targetAllocationContext` of the action’s parent class. This is due to the fact that it references the target assembly context which references the encapsulated component. Instead, we added the `sourceAllocationContext` attribute which represents the allocation context of the component instance which is being replaced by the action. This is relevant because the old component’s state may have to be applied to the new component.

- **ResourceContainerAction**: We replaced the `sourceResourceContainer` attribute by the `targetResourceContainer` attribute because the attribute represents the action’s target and not its source. Additionally, we added the `targetLinkingResources` attribute. It lists the linking resources connected to the affected resource container. With the linking resources we can find out to which networks a resource container is connected which is relevant if we allocate or deallocate new resource containers. We made this class abstract because it is only a supertype containing common attributes.

- **AllocationAction and DeallocationAction**: We did not add or remove any attributes because all relevant information is stored in these action’s parent class.

We updated the existing factories for these actions to support our modifications. These were the `ActionFactory`, the `AssemblyContextActionFactory`, and the `ResourceContainerActionFactory`. 

46
Figure 4.6. The updated system-adaptation-model containing all composed adaptation actions
4. Architecture Evolution

Atomic Adaptation Actions  The composed adaptation actions we introduced in the previous section, are useful to describe the system's adaptation on an abstract level. When it comes to actually adapting the system in a cloud-based environment, they can not be executed directly. Several steps have to be executed to perform a composed action. We call these steps the atomic actions in this thesis. We chose the term "composed" action because such an action is composed of a sequence of atomic actions. We chose the term "atomic" actions because each atomic action is applicable to a cloud based environment. We assume that in a sequence of actions one atomic action has to be completely executed before the next action will be executed. We present an example to clarify the difference between composed and atomic actions before we introduce the atomic adaptation actions used in this thesis: Gomaa and Hussein [2004] say that to replace a given system component, this component first has to stop being active and become quiescent. Components communicating with the given component need to stop communication and the given component needs to be unlinked, removed and replaced by a new component. Finally, the communication needs to be relinked and restarted. The replacement of a component is the composed action here. Even though we did not define a replace action in the previous section, it is equivalent to the change of repository component and similar to the migration action we defined. The atomic actions to execute the composed action in the example stop the component from being active, stop communication with other components, unlink, remove, and replace the component. Relinking and restarting the communication are atomic actions as well. So, we saw that we split one single composed action into seven atomic actions.

We implemented the atomic actions in an EMF model as we did with the composed actions. The model's root contains an ordered list of all atomic actions and acts as our execution plan. Therefore, the entire model is called the execution plan model. The model is shown in Figure 4.7 and we list the different classes of the execution plan model below. The indentation depicts the model's class hierarchy.

▷ AtomicAction: This abstract class is the parent class of all atomic actions.

▷ AssemblyContextAction: This abstract class extends AtomicAction and is the parent class of all atomic actions related to assembly contexts. Actions extending this class therefore affect a system's software components. The class references the targetAllocationContext which is the allocation context of the component affected by a concrete child action.

▷ DeployComponentAction: This action represents the deployment of a new component instance on a certain resource container. The information which component and which resource container are included in the parent classes targetAllocationContext. This action must not only trigger a component's deployment and return as our evaluation showed in 5. It has to wait until the deployed component has actually become available.

▷ UndeployComponentAction: This action represents the undeployment of a com-
4.2. Computation of Execution Plans

ponent instance from a certain resource container. The information on which com-
ponent and which resource container are included in the targetAllocationContext
of the parent class. We assume that before executing this action the affected
component has already been unlinked and its state has been migrated if needed.

- **MigrateComponentStateAction**: This action represents the migration of a compo-
nent’s state to a component of the same or an equivalent type. The sourceAllocation
Context references the component’s assembly context and resource container. The
state is migrated to the component defined by the targetAllocationContext in this
action’s parents class.

- **ConnectComponentAction**: This action represents the connection of a newly
deployed component. Besides the targetAllocationContext of its parent class, it
lists the targetProvidingAllocationContexts and the targetRequiringAllocation
Contexts. The former contain those components which provide an interface for
requests to the target component. The latter contain those components which
require an interface for request from the target component. These components
need to be connected to the target component.

- **BlockRequestsToComponentAction**: With this action, we indicate that incom-
ing requests to the targetAllocationContext defined in the parent class need to
be blocked. The targetRequiringAllocationContexts reference the components
which require an interface for requests from the target component. Those are the
components whose potential requests we want to block with this action.

- **DisconnectComponentAction**: This action represent the disconnection of compo-
nent to be undeployed. Besides the targetAllocationContext defined in its par-
ent class, it lists the targetProvidingAllocationContexts and the targetRequiring
AllocationContexts. The further contain those components which provide an
interface for requests to the target component. The latter contain those compo-
nents which require an interface for request from the target component. These
components need to be disconnected from the target component.

- **FinishComponentAction**: This actions is intended to ensure that a component is
able to finish ongoing transactions. We decided to always finish running transac-
tions first because repeating aborted transactions after the system’s reconfiguration
may hold the risk of side effects. Note that we use the term transaction here only
for a components computation steps and do not assume that such a transaction is
atomic and totally rolled back once it is aborted during execution.

- **ResourceContainerAction**: This abstract class extends AtomicAction and is the
parent class of all atomic actions related to resource containers. Actions extend-
ing this class therefore affect a system’s hardware nodes. The class references the
targetResourceContainer which is the resource container affected by a concrete child
action.

- **AllocateNodeAction**: This action represents the allocation of a resource container
in a cloud-environment.
4. Architecture Evolution

- **DealocateNodeAction**: This action represents the deallocation of a resource container in a cloud-environment.

- **ConnectNodeAction**: This action represents the connection of a resource container in a cloud-environment. It contains the `targetConnectors` which is explained in the next action.

- **DisconnectNodeAction**: This action represents the disconnection of a resource container in a cloud-environment. The `targetConnectors` contain the linking resources to be established. Linked resource containers can also be accessed from the linking resource. Network connections between different resource containers are physical in the first place which means that we are not able to establish a connection which does not exist physically. However, these actions are created for network configuration operations.
Figure 4.7. The execution plan model containing all atomic adaptation actions
4. Architecture Evolution

Note that these actions only define what kind of actions have to be executed in order to adapt a cloud-based system. They provide the available information from the CAM but their concrete implementation is part of the execution phase. We also implemented a factory class called `AtomicActionFactory` containing methods for initializing the different actions with the required PCM components.

Assigning the Correspondent Atomic Actions to a Composed Action  Now that we have defined the composed actions as well as the atomic actions, we can define the required sequence of atomic actions in order to execute a composed action. We present the mapping we chose in the following paragraph:

- For a ReplicationAction, a new component needs to be deployed. Potentially, the component's state needs to be migrated to this new instance as well [Oreizy et al. 1998]. Finally, the communication connections need to be established for the new instance. Therefore, a DeployComponentAction, a MigrateComponentStateAction, and a ConnectComponentAction are created.

- For a DereplicationAction, incoming requests to the component instance need to be blocked meaning that no new calls are dispatched to this instance. Running transactions need to finish [Von Massow et al. 2011]. Afterwards, the communication bindings have to be removed [Morin et al. 2009] before the component can be undeployed. Therefore, a BlockRequestsToComponentAction, a FinishComponentAction, a DisconnectComponentAction, and an UndeployComponentAction are created.

- A MigrationAction can be implemented as a replication followed by a dereplication [Von Massow et al. 2011]. Therefore, a DeployComponentAction, a MigrateComponentStateAction, and a ConnectComponentAction followed by a BlockRequestsToComponentAction, a FinishComponentAction, a DisconnectComponentAction, and an UndeployComponentAction are created.

- A ChangeRepositoryComponentAction is similar to a migration with the difference that the new equivalent component instance is deployed on the same resource container as the old one. Gomaa and Hussein [2004] state that the component to be replaced “has to stop being active and become quiescent, the components that it communicates with need to stop communicating with it; the component then needs to be unlinked, removed and replaced by the new component, after which the configuration needs to be relinked and restarted.” We chose to deploy the new component instance before the old instance is stopped to improve the cloud-based system’s availability. Thereby, we ensure that either the old or the new component is available at any time during the adaptation. This order is also proposed by Oreizy et al. [1998] but they also point out that both components must not be simultaneously active during the change. We do not consider this constraint in our implementation as our focus in this thesis is on the cloud-based system’s availability. However, we address this
4.2. Computation of Execution Plans

issue in our discussion concerning future work in Section 7.2. So far, we create a DeployComponentAction, a MigrateComponentStateAction, and a ConnectComponentAction followed by a BlockRequestsToComponentAction, a FinishComponentAction, a Disconnect ComponentAction, and an UndeployComponentAction.

For an Allocation action, a resource container needs to be allocated and made available for network communication. Garlan et al. [2003] mention that a server needs to be found, activated, and connected. In this thesis, finding a server is already completed by PerOpteryx. Depending on the cloud environment, this server still has to be allocated or activated and connected. As mentioned before, connecting does not reference to the physical network connections but rather to their configuration in this context. Both operations highly depend on the cloud environment. We create a AllocateNodeAction and a ConnectNodeAction.

For a DeallocateAction, we use the inverse operations to the allocation of a resource container. It needs to be disconnected and deallocated and we create a DisconnectNodeAction and a DeallocateNodeAction.

A Rule-Based Approach for Deriving an Execution Plan from Model Comparison  
We considered two potential rule engines for our approach. The rule engine had to be freely available and embeddable in Java to integrate it into iObserve. At first, we considered the Jess\(^1\) rule engine. This rule engine can be embedded into Java code and is available at no cost for academic use. However, a research based academic license has to be requested first. Additionally, even though the Jess rule engine is around since 1995 there are no regular updates. The last official release of version 7 was in 2008. Even though a public alpha of version 8 is promoted on the website which was released in 2013, no stable version 8 has been released since then. Therefore, we continued our search for a rule engine. We choose to use the JBoss Drools rule engine presented in 2.8. It is also freely available and can be embedded in Java. The Drools rule engine receives updates on a regular basis which is why we chose it over Jess. In this thesis, we use the version 7.6.0 of the drools-compiler and drools-core libraries.

Rules for Identifying Composed Adaptation Actions  
Using a rule engine enables us to use a different approach to the generation of the composed adaptation actions. We still use the graph representations of PAM and CAM. However, we do not have to iterate through all components and nodes in the CAM graph, getting the PAM graph components from a dictionary, and applying nested if-then-else statements anymore. We only add all graph components – namely component nodes (representing software component instances) and deployment nodes (representing resource container instances) – to a stateless KieSession’s working memory no matter if they belong to the CAM or the PAM. Additionally, we

\(^1\)E. Friedman-Hill, Jess, the Rule Engine for the Java Platform URL: http://www.jessrules.com/jess/ (visited on 06/05/2018).
create a new system-adaptation-model instance and add it to the working memory as well. With the rules we describe later, the rule engine is able to detect and create the right composed adaptation actions and add them to the new system-adaptation-model instance. A stateless session is sufficient in this context because we insert all current components at once. A stateful session would only make sense when facts are added or removed over time which is not the case here. We added an enumerated type called ModelGraphRevision to the components of the simplified model graph to distinguish PAM and CAM components. This type contains the two values RUNTIME for PAM graph components and REDEPLOYMENT for CAM graph components. We also fixed the issue where assembly contexts were seen as deployed component instances in the original graph. An assembly context was mapped to a resource container in the graph factory. We changed this in a way that a graph component node basically represents an allocation context. It refers to the deployment node representing the resource container and also contains a reference to the assembly context of the represented component. However, it does not link the assembly context to a resource container directly anymore. We then designed our rules to replicate the comparison algorithm of the existing approach. We implemented the following rules for the creation of assembly context actions:

- **Create Replication Action**: This rule is fired when there is a component node from the CAM graph but no component node from the PAM graph representing the same allocation context. This means that there is a new allocation context representing a component instance in the CAM which was not present in the PAM. This new instance must have been created by replication of an existing instance. Therefore, a replication action is created. If there already have been multiple instances of this same component in the PAM, we are not able to identify which instance exactly has been replicated. In this case an arbitrary one is chosen as the replication’s source by the rule engine. This rule has a salience value of 0.

- **Create Dereplication Action**: This rule is fired when there is a component node from the PAM graph but no component node from the CAM graph representing the same allocation context. This means that there is an allocation context representing a component instance in the PAM which does not exist any more in the CAM. The old instance must have been dereplicated. Therefore a dereplication action is created for this component instance. This rule has a salience value of 0.

- **Create Migration Action**: This rule is fired when there is a component node from the PAM as well as a component node from the CAM representing the same allocation context whose resource containers differ. This means that the same component instance is deployed on different resource containers in PAM and CAM. The instance must have been migrated. Therefore, a migration action is created for this component instance. This rule has a salience value of 0.

- **Create Change Repository Component Action**: This rule is fired when there is a component node from the PAM as well as a component node from the CAM representing the
same allocation context whose encapsulated components differ. This means that a component instance from the PAM is represented by an instance of a different component in the CAM. The component must have been exchanged by an equivalent component from the repository. Therefore, a change repository component action is created for this component. This rule has a salience value of 0.

For the creation of resource container action actions we created the following rules:

- **Create Allocation Action**: This rule is fired when there is a deployment node from the CAM graph but no deployment node from the PAM graph representing the same resource container. This means that there is a new resource container in the CAM which was not present in the PAM. This new resource container must have been allocated. Therefore, an allocation action is created for this resource container. This rule has a salience value of 1.

- **Create Deallocation Action**: This rule is fired when there is a deployment node from the PAM graph but no deployment node from the CAM graph representing the same resource container. This means that there is a resource container in the PAM which does not exist any more in the CAM. The resource container must have been deallocated. Therefore, a deallocation action is created for this resource container. This rule has a salience value of -1.

We used the salience values to create a general order of the composed adaptation actions. Allocations of new resource containers have the highest priority and are always executed first. This way we ensure that there are no migrations to resource containers which are not allocated yet. Deallocations have the lowest priority and are always executed last. Thus, we ensure that all deallocations or migrations of components to different resource containers have finished before a resource container is deallocated. Note that in contrast to the existing approach all adaptation actions related to software components have the same priority. This is a design decision which results from the fact that the low-level atomic adaptation actions take care of dependencies between components and component states. We do not care when exactly a component is adapted because the atomic adaptation actions for the adaptation take care to reroute requests from depending components or migrate the state.

An order of the composed adaptation action can only be used to improve the cloud-based systems overall performance. For example, it would make sense migrate two components which both have to be migrated from resource container A to resource container B shortly after another to avoid longer communication distances for a longer time period. However, this is not the goal of this thesis as we only focus on availability in general.

**The Adaptation Service in iObserve** The processes described in this section were implemented as a separate service in the adaptation package. The service’s starting point is the AdaptationMain class. Figure 4.8 shows its pipe and filter architecture starting with two paths each containing a SingleConnectionTcpReaderStage followed by an ModelFiles2ModelDir.
4. Architecture Evolution

CollectorStage. As in the planning service, we use this sequence of filters to receive and collect the PCM models. One path receives the PAM while the CAM is received on the second path.

The AdaptationDataCreator initializes the AdaptationData records Pöppke [2017] and Weimann [2017] introduced and synchronizes the two incoming paths. It is a composite stage as shown in Figure 4.8 and includes a ModelCollector stage followed by a ModelGraphCreator stage. The former collects the model directories of PAM and CAM and stores their paths on the file system in the AdaptationData record. The latter receives the initialized record, builds the simplified model graphs presented in section 2.6 for both models, and adds them to the record. The ActionFactoryInitialization stage initializes the action factory Pöppke and Weimann created for their system-adaptation-model with the paths to the PAM and CAM directories. Thus, the factory is able to access the models and include references to the models into the produced actions. The stage passes the received AdaptationData records to the ComposedActionComputation. As described above, this stage adds the simplified model graphs together with an empty system-adaptation-model to the drools rule engine which initializes the model with a list of composed adaptation actions.

The model is then passed to the AtomicActionComputation stage which is another composite stage shown in Figure 4.9. The SystemAdaptationModel2ComposedActions sends the
4.2. Computation of Execution Plans

composed actions contained in the model to its output port keeping their order. For each incoming composed action, the ComposedActions2AtomicActions stage computes the atomic actions needed to execute the composed action. In our implementation, this was realized by using concrete transformation classes for the different composed action types. The transformation classes provide the uniform IComposed2AtomicAction interface. We also considered implementing these transformation classes as individual filters connected to TeeTime's MultipleInstanceOffFilter. While this would have been a nicer solution to distinguish the concrete types of composed adaptation actions, there was the risk of loosing the order of the actions. As long as the MultipleInstanceOffFilter and the different transformation filters run in the same thread, the order stays unchanged. As soon as some of the transformation filters are declared active and run in different threads, the order only depends on which transformation finishes first. Given the fact that TeeTime is designed for the parallelization of stages and does not provide a mechanism to force a stage to be passive, we did not want to rely only on our configuration. Therefore, we choose the implementation with the decoupled transformation classes over transformation filters. The atomic actions are collected and stored in an execution plan model in the AtomicActions2ExecutionPlan stage. We use a TeeTime signal initially sent from the SystemAdaptationModel2ComposedActions stage to indicate the end of the action stream to the AtomicActions2ExecutionPlan stage.

Since we use a PCM model for the execution plan, we are able to serialize the exe-
4. Architecture Evolution

cution plan in the ExecutionPlanSerialization stage. The serialized plan is passed to the AdaptationResultDistributor. It passes the execution plan, as well as the PAM and CAM to the following SingleConnectionTcpWriterStages which send them to the execution service.

4.2.4 Results of the Execution Plan Computation’s Evolution Process

In this section, we presented our approach to the computation of execution plans which extends upon the existing approach. We introduced a new more common terminology for adaptation and adapted the existing system-adaptation-model to this terminology. On top of that, we were able to extend the model to store more information which now allows us to identify referencing components or sources of replications. We exchanged the nested computation algorithm by a rule-based approach for the computation of the composed adaptation actions to improve readability and extendability. During this task we fixed an issue were the semantics of assembly contexts and allocation contexts were mixed up in the generated graph representation. By adding the low-level atomic adaptation actions, we were able to address component dependencies and internal component states. Using another EMF model for the atomic actions, we can provide a serializable execution plan which can be sent to the execution phase.

4.3 Execution of Execution Plans on Concrete Cloud Environments

In Section 3.5, we presented the approach and challenges iObserve’s execution service faces with respect to different cloud providers. In this section we present our implementation of a flexible execution phase within iObserve’s control loop.

4.3.1 Execution of System Adaptations in the Existing Approach

We already mentioned the problem of different cloud provider’s interfaces which have to be accessible from iObserve’s execution phase. Pöppke [2017] and Weimann [2017] used JClouds as a middleware component between iObserve and a system’s cloud infrastructure. Thus, they were able to use a platform independent part in iObserve which calls platform dependent execution scripts via JClouds as described in Section 2.6. They discussed the use of multiple platform dependent execution implementations in their theses and came to the conclusion that this results in too much effort for implementation and especially maintenance.

However, the available cloud infrastructure during this thesis is a Kubernetes cluster and JClouds does not support Kubernetes at this point of time. Therefore, we can not use the existing approach with its JClouds middleware and have to provide an alternative solution. Another problem is that our modifications to iObserve’s adaptation phase also
affect the execution phase: We introduced the new atomic adaptation actions. The existing execution is not able to process these new actions and the platform dependent execution scripts would have to be updated as well.

### 4.3.2 Outline for the Evolution of the Execution

To address the problems with the existing approach mentioned in the previous section we provide mechanisms which enable the execution service to receive execution plan instances and execute the contained atomic adaptation actions. We realize this with an architecture supporting different cloud provider specific implementations. In this thesis, we provide an implementation for Kubernetes. Implementations to interfaces of other cloud providers or middleware components can be added in the future. We have to consider certain aspects to enable our execution service to perform system adaptations on a cluster. Besides access to the adaptation interfaces provided by Kubernetes we also have to consider the level of abstraction which comes with Kubernetes. As described in Section 2.9, Kubernetes is a powerful platform which includes advanced mechanisms which go beyond the simple access to the cloud resources. Examples for these mechanisms are the included load balancing and auto repair features which let the programmer define a certain number of component instances in advance. The Kubernetes platform then automatically manages the load balancing and also starts new instances if running instances stop working on their own. A direct deployment of component instances to machines is not intended with Kubernetes. Therefore, we adapt our interpretation of resource containers from the PCM model.

### 4.3.3 Implementation of the Execution Evolution

**Representation of Resource Containers in Kubernetes**

So far, when we referred to the PCM model’s resource containers we always associated them with the actual hardware components, i.e. the available servers or virtual machines. A resource container instance would represent a concrete machine with all its specifications concerning processing rate or storage speeds. Component instances could be allocated on this particular machine. This association is intuitive, easy to understand, and fits many cloud environments such as the Amazon Web Services Pöppke [2017] and Weimann [2017] used for in the evaluation of their implementation. However, this does not apply to a Kubernetes cluster. Even though such a cluster consists of a number of nodes as described in Section 2.9 assigning specific component instances to specific resource containers is not intended in Kubernetes.

A naive approach might map PCM allocation contexts to Kubernetes pods and PCM resource containers to Kubernetes nodes. In this case, we had to make sure that a pod runs on the node representing the resource container specified in the allocation context. This could be achieved using a `nodeselector` field in the pod’s specification. Such a field defines certain labels and values for a pod which an executing node then needs to define as
4. Architecture Evolution

well to be considered suitable for the pod. Using the particular resource container id there would ensure that this pod is only deployed on this particular node. Figure 4.10 depicts such a situation: On the left side, we see a PCM model extract where the assembly context encapsulating a component called account is connected to the resource container with name node1 and a processing rate of 15. The available hardware environment is depicted on the right side. With the naive approach the account component is deployed on the machine node1. However, this is not the way the nodeselector field is intended to be used. Actually the idea is to only use it to narrow the set of possibly executing nodes for a pod. disktype: ssd is an example label’s key-value pair which is limiting the potential executing nodes for a pod to those containing an SSD storage. Using the fixed resource container id here would always leave exactly one possible execution node and all automatic load balancing features of Kubernetes would be lost.

![Figure 4.10. Interpretation of resource containers as nodes](image)

Therefore, we decided to use an alternative mapping which does not impede Kubernetes in its abilities of load balancing. Instead of assuming that a resource container represents a machine, we now assume that it represents a Kubernetes pod specification. As described in Section 2.9 Kubernetes relies on containerized application components. For our interpretation we assume that a pod encapsulates exactly one container with a component inside. The resource container’s hardware specification now represents this containerized component’s hardware requirements. An allocation context now only determines in which kind of pod a component is encapsulated. Allocating the same component to two resource containers with different hardware specifications now means that there are two kinds of pods created. Each of them having different hardware requirements. The decision on how to balance pods between nodes is left to Kubernetes completely. This interpretation raises the level of abstraction above the deployment of component instances to concrete machines. An example can be seen in Figure 4.11. On the left side, we see the same PCM extract as in the previous example. However, this time the resource container represents a pod specification which is the reason why we changed its name to accountPod1. The processingRate of 15 now specifies the pod’s requirements. It is no longer deployed to a specific resource container as we see on the right side of Figure 4.11. Kubernetes can deploy the pod on any node with a sufficient processing rate. In this example node1 and node2 are
valid alternatives whereas the processing rate of node3 is too low.

Figure 4.11. Interpretation of resource containers as pod specifications

Additionally, a container image is needed to start a pod. A reference to the image must be accessible from the PCM model. Thus, the correct image for a certain software component can be found. It makes sense to associate an image with an assembly context instead of a plain component. An assembly context is part of a systems architecture and containers are usually configured to integrate into a certain architecture in terms of their exposed ports, for example. However, the PCM was not designed for containerized applications and assembly contexts do not contain a dedicated image locator field or something similar. For this reason, a so called correspondence model was created in the past to solve this issue in iObserve. It contains entries of the type AssemblyEntry which contain an implementationId field where the location of an image can be specified. Such an entry also references the assembly context in the system model. We are able to find the image for a certain assembly context with this information.

The Executor Interface We provide an approach which is compatible with all atomic actions and can be used to access the infrastructures of different cloud providers with the executor interface shown in Listing 4.3. It is parameterized with a generic type T extending an atomic action and defines a single method - the execute method. This method receives an atomic action of type T as an argument. Thus, we force implementing executor classes to be parameterized with a concrete atomic adaptation action meaning that its execute-method only accepts instances of this particular action type as its argument.
4. Architecture Evolution

```java
public interface IExecutor<T extends AtomicAction> {
    void execute(T action);
}
```

**Listing 4.3. Interface for executors**

In the execution service’s Pipe-and-Filter architecture this allows us to define a stage forwarding the incoming atomic adaptation actions to the appropriate executor. The parameterized interface thereby allows us to specify the action’s type but still supports executor implementations for various cloud providers such as Kubernetes or JClouds. Note that the executors themselves are not TeeTime filters to ensure that the executions take place in a single thread and therefore happen sequentially.

**Implementation of Executors for Kubernetes**

Many of our atomic adaptation actions are automated by the Kubernetes system. We do not have to manually connect or disconnect components or nodes in Kubernetes. There is no need to block incoming requests to a component before its undeployment or wait for running transactions to finish in Kubernetes as well. This is due to the fact that the termination of pods is performed gracefully in Kubernetes. The migration of states may be necessary depending on the migrated components. We do not implement it in this thesis, because the components included in our evaluation application mentioned in Chapter 5 do not require a migration of state. We used the Fabric8 client for the communication between the executors and the Kubernetes cluster in our implementation. We choose this library because at this point of time, it was the most commonly used and well documented library of the three available Java client libraries mentioned in Section 2.9. In this thesis, we define the following executors for the remaining atomic adaptation actions:

- **AllocationExecutor**: The AllocationExecutor executes an AllocateNodeAction. Due to the fact that there is no allocation in Kubernetes, it only creates a Deployment specification. It inserts the resource container specific attributes such as its name and its hardware specifications. The resource container name and the prepared deployment specification are then stored in a map which is shared with the DeploymentExecutor.

- **DeploymentExecutor**: The DeploymentExecutor receives a DeployComponentAction. It extracts its target allocation context and receives the resource container name from the associated resource container. The executor then checks if there already exists a Kubernetes deployment with this specific name on the cluster. If this is the case, the

---


4.3. Execution of Execution Plans on Concrete Cloud Environments

deployment action is executed by simply incrementing the number of replicas of the received Kubernetes deployment. Otherwise, the deployment must take place to a newly allocated component. Therefore, the executor looks up the resource container name in the map shared with the AllocationExecutor and receives the prepared deployment specification. It adds the name of the required Docker image which is received from the correspondence model available to the execution service. The name of the assembly context to deploy is added as well before the deployment is created on the cluster. The executor then constantly checks whether the deployment has become ready. This is currently realized with a busy waiting loop because, at this point of time, the cluster does not automatically report when a component became ready after its deployment. This way, we ensure that the executor only return once the deployment actually became available.

- **UndeploymentExecutor**: The UndeploymentExecutor gets an UndeployComponentAction. It extracts the name of the resource container which is linked to the target allocation context. It then receives the deployment with this name from the cluster and decrements its number of replicas. Note that decrementing this value to zero does not remove the deployment. This is done by the executor for deallocations.

- **DeallocationExecutor**: The DeallocationExecutor receives a DeallocateNodeAction and extracts the affected resource container’s name from its target resource container. It then deletes the deployment with the given name from the Kubernetes cluster.

**The Execution Service in iObserve** The execution service’s architecture is shown in Figure 4.12. The entry point for our Kubernetes-specific implementation is the Kubernetes ExecutionMain class. It loads specific attributes such as the Kubernetes namespace and the location of the correspondence model from a configuration file. The architecture starts with three paths of SingleConnectionTcpReaders which receive the serialized execution plan, as well as the serialized PAM and CAM. The latter are not accessed directly from the execution service but the atomic actions within the execution plan reference to some of their components. Therefore, they have to be present as well. The readers for the PAM and the CAM are followed by ModelFile2ModelDirCollectorStages to collect the different model files as in the adaptation service. All three paths then lead to the ModelCollector stage, which ensures that execution plan as well as PAM and CAM are present. Therefore, this stage is declared active in the TeeTime configuration. Only the execution plan is then forwarded to the ExecutionPlanDeserialization stage, which deserializes the execution plan into a Java object. The execution plan object is then send to the ExecutionPlan2AtomicActions stage. It extracts the list of atomic adaptation actions from the execution plan and forwards the individual actions to its output port preserving their order. The AtomicActionExecution stage then executes the different actions sequentially using the dedicated executors. The executors are set in the filter’s constructor and implement the executor previously described executor interface. In Figure 4.12 we can see that the architecture of the AtomicActionExecu-
4. Architecture Evolution

The filter is similar to the ComposedActions2AtomicActions filter’s architecture we described in Section 4.2. We use this architecture to ensure that the execute methods of different executors are executed sequentially and do not interfere. Implementing the executors as independent TeeTime stages would mean that they could be declared active. With the resulting parallel execution we could not make any assumption regarding a sequential execution.

![Pipe-and-Filter architecture of iObserve’s execution service](image)

**Figure 4.12.** Pipe-and-Filter architecture of iObserve’s execution service

4.3.4 Results of the Execution’s Evolution Process

We presented our approach to an adaptable execution service in this section. We redesigned its architecture to enable the processing of our previously introduced execution plans. It now provides executor interfaces which can be used by implementations for specific cloud providers. We also provide a first implementation of executors for Kubernetes clusters. A new interpretation of the PCM guaranteed its that it still remains compatible with higher level cloud infrastructures such as Kubernetes.
In this chapter we evaluate our approach to the planning and execution of system adaptations. Therefore certain evaluation goals are specified in Section 5.1 and a suitable evaluation methodology is chosen in Section 5.2. In Section 5.3 we present our experimental design. The evaluation’s operation is described in Section 5.4 followed by details on how we collected the data on the executed experiments in Section 5.5. The results are presented in Section 5.6 and discussed in Section 5.7. We address threads to validity in Section 5.8 and summarize our results in 5.9.

5.1 Evaluation Goals

With iObserve’s service based architecture, we split the general system adaptation task into different subtasks. Table 5.1 lists the different tasks. We want to find out whether our approach is capable of creating execution plans for different runtime adaptation scenarios of a real world cloud based system (T1). Additionally, we want to find out if our approach is able to apply the created execution plans to the real system thereby adapting its architecture correctly (T2). In conjunction with the previous tasks, we want to find out whether the serviced based architecture of our approach works as intended (T3).

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Creation of an execution plan</td>
</tr>
<tr>
<td>T2</td>
<td>Application of an execution plan to the system</td>
</tr>
<tr>
<td>T3</td>
<td>Expected behavior of the service based architecture</td>
</tr>
</tbody>
</table>

5.2 Methodology

We evaluate our approach by the means of a feasibility study. Such a study can show whether a particular system, practice or design approach can be accomplished at all. Thus, a feasibility study is appropriate to evaluate scenarios where issues are addressed in a completely new way [Shaw 2003]. Even though our approach was based on an existing
5. Evaluation

approach, we added many completely new aspects such as the rule engine, the atomic adaptation actions and the Kubernetes execution interface. Therefore, a feasibility study is meaningful for our thesis to evaluate whether our newly introduced approaches could be implemented successfully.

5.3 Experimental Design

In this section we present our experimental design. We introduce the execution environment as well as the different adaptation scenarios we observe during the evaluation experiment.

5.3.1 Execution Environment

The execution environment for the evaluation of our additions to iObserve comprises a distributed web application resembling the observed system and a Kubernetes cluster as cloud infrastructure. We observe a forked version of the MyBatis JPetStore\(^1\) with iObserve.

**Distributed JPetStore** The JPetStore web application resembles an online shop for pets. It offers various different pets from the five categories fish, dogs, cats, reptiles, and birds. It features a catalog with detailed information for each pet such as its name, its race, the number of pets in stock, and its price. A user can choose the animals from the catalog and add them to his shopping cart. He has to create an account to finally proceed to the checkout to buy the chosen pets.

From an architecture point of view, the JPetStore web application consists of the four components frontend, account, catalog, and order, as well as additional account, catalog, and order database components. The architecture is shown in Figure 5.1. One can see that the frontend component is connected to all of the three other main components. Each of them is connected to its database component and the catalog and order components also have access to each others databases. All databases are part of a single database component. Docker containers already exist for the frontend, account, catalog, and order components which can be used for this evaluation. At this point of time, the databases are located in their respective component’s containers and not in an additional database component as depicted in Figure 5.1. This will be subject to changes in the future. However, it is the reason why no database components can be seen in our evaluation logs in Section 5.6.

**The Evaluation Cloud Environment** The Kubernetes cluster available for this evaluation consists of one master node and four worker nodes as depicted in Figure 5.2. The master machine is named cc01 while the worker machines are named nc05 to nc08. For our

---

evaluation run, we will start the iObserve adaptation and the execution services on the master node. The planning service cannot be tested due to the problems which occurred during the integration of PerOpteryx. Instead we run a version of the planning service which mocks the actual model optimization and outputs predefined versions of present-architecture-model (PAM) and candidate-architecture-model (CAM).
5. Evaluation

5.3.2 Adaptation Scenarios

We use different minimal adaptation scenarios to evaluate our approach. As a context for our evaluation, we can imagine that our software system faces a sharp increase in account registrations after an advert ran on TV. The load on the accounting service grows and may lead to Service Level Agreement (SLA) violations. Therefore, the iObserve planning computes a new target deployment of the JPetStore. After a short period of time, the load reduces again and the planning computes another configuration for the reduced load. The scenarios are designed to cover all relevant adaptation actions. We created predefined PAM and CAM models, as well as initial Kubernetes architecture configurations for each of the following scenarios. The scenario data is located in a dedicated repository [Blümke 2018] which can be used to reproduce the evaluation experiments we refer to.

Replication  In this scenario, we perform a replication of the account component to an already existing resource container. Such a replication can be used to increase the overall performance by distributing the load among the available hardware resources in our evaluation context. The initial architecture shown in Figure 5.3 (1) contains resource containers for all four JPetStore components with one component instance deployed on each resource container. A second resource container with different hardware specifications for account components already exists but no component instances are deployed on it yet. The replication operation deploys an instance of the account component on the empty resource container as depicted in Figure 5.3 (2). This results in exactly one component instance deployed on each resource container and two resource containers for the account component. Performing the replication repeatedly results in more component instances deployed on the target resource container. This scenario could also contain the allocation, but we will cover the allocation in the migration scenario. For this reason, we decided to leave it out here to keep the scenario as minimalistic as possible.

Dereplication  In this scenario, we perform a dereplication of the account component from a resource container. We can imagine that a dereplication is performed after the load peak has been overcome in our evaluation context. We assume that the planning outputs a configuration which implies that the account service can be scaled down in the observed system. In the initial architecture shown in Figure 5.4 (1), one instance of each JPetStore component is deployed on a separate resource container except for the account component. There are two instances of the account component and each of them is deployed on a different resource container. The dereplication actions undeploys one of them leaving its resource container empty as depicted in Figure 5.4 (2). This scenario may also start with more than one account component instance on one resource container to allow performing multiple dereplication in a row. The replication and dereplication scenarios may also be performed alternately if they are targeted at the same component instance.
5.3. Experimental Design

Migration and (De-) Allocation  In this scenario we perform a migration of the account component to a newly allocated resource container. Referring to our evaluation context, we can imagine that the new resource container offers more computing resources and therefore increases the performance. In the initial architecture shown in Figure 5.5 (1) one instance of each JPetStore component is available and deployed on a separate resource container.

In a first step, a new resource container is allocated resulting in two account resource containers overall as depicted in Figure 5.5 (2). The new resource container is still empty,
5. Evaluation

![Diagram showing steps of the JPetStore allocation scenario](image)

Figure 5.5. Steps of the JPetStore allocation scenario

though. In the next step, a new instance of the account component is deployed on this empty container as depicted in Figure 5.3. This results in exactly one component deployed on each resource container once again. So far, there are two resource containers with different specifications hosting an instance of the allocation component. The old account component instance is then undeployed from its resource container to finish the migration and leaving the old allocation resource container empty. This can be seen in Figure 5.4. Finally, this empty resource container is deallocated resulting in an architecture with four resource containers and one component instance deployed on each of them. However, the allocation, migration, and deallocation actions now left the account component on a resource container with different hardware specifications. Figure 5.6 depicts this final adaptation step.

5.4 Operation

To execute the evaluation experiments we got access to the Kubernetes master via a Secure Shell (SSH) connection. On the master, we cloned our prepared experiment repository. Besides the already described predefined PCM models and Kubernetes configurations, the repository also contains scripts to set up and execute the different iObserve services. Due to the fact that the Kubernetes master node only had a Java Runtime Environment installed, we were not able to build iObserve on the master node. Therefore, we had to build iObserve on our local machine. Afterwards, we copied the built tar archives containing the compiled versions of planning, adaptation, and execution services to the Kubernetes master node using secure copy (SCP). We unpacked the tar archives in a dedicated directory and created working directories for each of the three services. We used a script called
5.5 Data Collection

startup-kube-jpetstore.sh defined in the repository [Blümke 2018] to startup the JPetStore web application in a deployment configuration which fits our current adaptation scenario. Another script called iobserve-setup-all.sh was used to provide the predefined PAM and CAM to the mocked planning service as well as the correspondence model to the execution service in their working directories. Finally, we started the iObserve services with scripts called iobserve-run-<servicename>. The mocked planning service has to be started last because it immediately sends out the predefined PAM and CAM and the adaptation and execution services have to be available at this point of time.

5.5 Data Collection

For the evaluation of T1, we use the execution plan created by iObserve’s adaptation service. It can be found in the service’s working directory. We monitor services, deployments, and pods of the used Kubernetes cluster to evaluate T2. The cluster’s particular states can be displayed with the kubectl get <target> command for the targets services, deployments, and pods. We use a script which prints the command’s output to a file once a second. Thus, we are able to retrace all changes happening on the cluster. We observe the services while running and search their outputs for errors or exceptions for the evaluation of T3.

5.6 Results

In this section, we present the results of our evaluation experiments. Note that in the execution plans the terms runtime model for the PAM and redeployment model for CAM are used. This is due to the fact that our implementation evolved the existing approach of
5. Evaluation

Pöppke [2017] and Weimann [2017] who used this terms. For staying consistent throughout the entire implementation we also used these terms in our experiments. For reasons of simplicity, we shortened the paths of the references an execution action has to the PCM models in the execution plan. The original paths can be seen in the experiment’s repository [Blümke 2018]. Also note that we only include those states of the logged Kubernetes services, deployments and pods in this section which include relevant changes to the cluster’s state. The complete logs are also available in the repository on the lbl-results branch.

5.6.1 Replication Scenario

T1: Creation of an Execution Plan  Listing 5.1 shows the execution plan created during the replication scenario. It contains the following actions:

- DeployComponentAction in lines 2-4 which references the target allocation context from the CAM.
- MigrateComponentStateAction in lines 5-8 which references the target allocation context from the CAM and the source allocation context from the PAM.
- ConnectComponentAction in lines 9-13 which references the target allocation context and its providing and requiring allocation contexts from the CAM.

Listing 5.1. Serialized execution plan in replication scenario

T2: Application of an Execution Plan to the System  The JPetStore service shown in line 2 of Listing 5.2 remains active during the entire experiment.
5.6. Results

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpetstore</td>
<td>ClusterIP</td>
<td>10.110.84.90</td>
<td>&lt;none&gt;</td>
<td>8080/TCP</td>
<td>38s</td>
</tr>
</tbody>
</table>

**Listing 5.2.** Output of `kubectl get services` during replication scenario

Listing 5.3 depicts how the deployments change throughout the experiment. In the initial state in lines 1-6 we see that there are not any instances of the account2 deployment in line 3 yet. All other deployments each have one running instance. The second state in lines 8-13 shows the output after the replication was executed. The desired, current and up-to-date instances of account2 were incremented to 1. However, there are still 0 instances available in this state. In the final state in lines 15-20 we see that one instance of each deployment is finally available.

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>38s</td>
</tr>
<tr>
<td>account2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>38s</td>
</tr>
<tr>
<td>catalog</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>38s</td>
</tr>
<tr>
<td>frontend</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>38s</td>
</tr>
<tr>
<td>order</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>38s</td>
</tr>
</tbody>
</table>

**Listing 5.3.** Outputs of `kubectl get deployments` during replication scenario

In Listing 5.4, we see the different states of the pods throughout the experiment. The first state in lines 1-5 shows the cluster’s initial configuration before the execution plan was executed. One pod of each component is running. There is no account2 pod yet. In the second state shown in lines 7-12 we see the first output after the replication was executed. A new pod with the name account2 was created in line 9. This new pod is not ready yet and its status indicates that a container is currently being created for it. In the final state in lines 14-19 all five pods are running.

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>41s</td>
</tr>
<tr>
<td>account2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>41s</td>
</tr>
<tr>
<td>catalog</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>41s</td>
</tr>
<tr>
<td>frontend</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>41s</td>
</tr>
<tr>
<td>order</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>41s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>44s</td>
</tr>
<tr>
<td>account2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>44s</td>
</tr>
<tr>
<td>catalog</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>44s</td>
</tr>
<tr>
<td>frontend</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>44s</td>
</tr>
<tr>
<td>order</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>44s</td>
</tr>
</tbody>
</table>
5. Evaluation

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account-546cddf8f-9m4qt</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>39s</td>
</tr>
<tr>
<td>catalog-67c66d6c6f-6flfl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>39s</td>
</tr>
<tr>
<td>frontend-69b8d9cf76-rmwjq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>39s</td>
</tr>
<tr>
<td>order-59d6d9bdf4-m2zhq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>39s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account-546cddf8f-9m4qt</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>41s</td>
</tr>
<tr>
<td>account2-546cddf8f-nbfs4</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>2s</td>
</tr>
<tr>
<td>catalog-67c66d6c6f-6flfl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>41s</td>
</tr>
<tr>
<td>frontend-69b8d9cf76-rmwjq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>41s</td>
</tr>
<tr>
<td>order-59d6d9bdf4-m2zhq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>41s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account-546cddf8f-9m4qt</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>44s</td>
</tr>
<tr>
<td>account2-546cddf8f-nbfs4</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5s</td>
</tr>
<tr>
<td>catalog-67c66d6c6f-6flfl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>44s</td>
</tr>
<tr>
<td>frontend-69b8d9cf76-rmwjq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>44s</td>
</tr>
<tr>
<td>order-59d6d9bdf4-m2zhq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>44s</td>
</tr>
</tbody>
</table>

Listing 5.4. Outputs of `kubectl get pods` during replication scenario

**T3: Expected Behavior of the Service Based Architecture**  No problems occurred after we started the planning, adaptation, and execution services. The mocked planning service terminated after it had sent the predefined PAM and CAM to the adaptation service. The models were stored in the service's working directory. The adaptation service sent the execution plan, as well as PAM and CAM to the execution service in whose working directory they were stored. The adaptation and execution service continued running. We were able to send more sets of models to them by restarting the mocked planning service repeatedly.

**5.6.2 Dereplication Scenario**

**T1: Creation of an Execution Plan**  The execution plan shown in Listing 5.5 was created during the migration scenario. It contains the following actions:

- BlockRequestsToComponentAction in lines 2-5 which references the target allocation context and one target requiring allocation context from the PAM.

- FinishComponentAction in lines 6-8 which references the target allocation context from the PAM.
5.6. Results

- DisconnectComponentAction in lines 9-13 which references the target allocation context and its target providing and requiring allocation contexts from the PAM.

- UndeployComponentAction in lines 14-16 which references the target allocation context from the PAM.

```xml
<org.iobserve:ExecutionPlan>
  <actions xsi:type="org.iobserve:BlockRequestsToComponentAction"/>
  <targetAllocationContext href="runtimemodel#_MODdwFw3Eei_Pr=AEFAvMw"/>
  <targetRequiringAllocationContexts href="runtimemodel#
    _1plbIA6EeiMaMdqfMnouQ"/>
</actions>

<actions xsi:type="org.iobserve:FinishComponentAction"/>
  <targetAllocationContext href="runtimemodel#_MODdwFw3Eei_Pr=AEFAvMw"/>
</actions>

<actions xsi:type="org.iobserve:DisconnectComponentAction">
  <targetAllocationContext href="runtimemodel#_MODdwFw3Eei_Pr=AEFAvMw"/>
  <targetProvidingAllocationContexts href="runtimemodel#
    _xLwtUAA6EeiMaMdqfMnouQ"/>
  <targetRequiringAllocationContexts href="runtimemodel#
    _1plbIA6EeiMaMdqfMnouQ"/>
</actions>

<actions xsi:type="org.iobserve:UndeployComponentAction">
  <targetAllocationContext href="runtimemodel#_MODdwFw3Eei_Pr=AEFAvMw"/>
</actions>
</org.iobserve:ExecutionPlan>
```

Listing 5.5. Serialized execution plan in dereplication scenario

T2: Application of an Execution Plan to the System  Listing 5.6 shows the JPetStore service in line 2 which remained active throughout the entire experiment.

```sh
NAME  TYPE  CLUSTER-IP      EXTERNAL-IP  PORT(S)     AGE
jpetstore  ClusterIP  10.110.84.90  <none>     8080/TCP  3m
```

Listing 5.6. Output of `kubectl get services` during dereplication scenario

In Listing 5.7, we see the changes to the deployments throughout the experiment. In the first state in lines 1-6 we see the scenario's initial state with one instance of each deployment. There are two deployments for the account component namely the account deployment in line 2 and the account2 deployment in line 3. In the second state in lines 8-13 the dereplication was performed and the number of instances of the account2 deployment in line 9 was decremented to 0.
5. Evaluation

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3m</td>
</tr>
<tr>
<td>account2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3m</td>
</tr>
<tr>
<td>catalog</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3m</td>
</tr>
<tr>
<td>frontend</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3m</td>
</tr>
<tr>
<td>order</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3m</td>
</tr>
</tbody>
</table>

Listing 5.7. Outputs of `kubectl get deployments` during dereplication scenario

Listing 5.8 shows the changes to the pods. In the first state in lines 1-6 we see five pods running with one instance each. There are two pods, `account` and `account2`, for the account component. The second state in lines 8-13 shows the output after the dereplication’s execution. The `account2` pod in line 10 changed its status to terminating. In the final state in lines 15-19 the `account2` pod has disappeared. The other four pods are still running.

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account-546cddf8f-9m4qt</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m</td>
</tr>
<tr>
<td>account2-546cddf8f-nbfs4</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>2m</td>
</tr>
<tr>
<td>catalog-67c66d6c6f-6flfl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m</td>
</tr>
<tr>
<td>frontend-69bd9cf76-rmwwj</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m</td>
</tr>
<tr>
<td>order-59d6d9bdf4-m2zhq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account-546cddf8f-9m4qt</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m</td>
</tr>
<tr>
<td>account2-546cddf8f-nbfs4</td>
<td>1/1</td>
<td>Terminating</td>
<td>0</td>
<td>3m</td>
</tr>
<tr>
<td>catalog-67c66d6c6f-6flfl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m</td>
</tr>
<tr>
<td>frontend-69bd9cf76-rmwwj</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m</td>
</tr>
<tr>
<td>order-59d6d9bdf4-m2zhq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account-546cddf8f-9m4qt</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>4m</td>
</tr>
<tr>
<td>catalog-67c66d6c6f-6flfl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>4m</td>
</tr>
<tr>
<td>frontend-69bd9cf76-rmwwj</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>4m</td>
</tr>
<tr>
<td>order-59d6d9bdf4-m2zhq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>4m</td>
</tr>
</tbody>
</table>

Listing 5.8. Outputs of `kubectl get pods` during dereplication scenario
5.6. Results

T3: Expected Behavior of the Service Based Architecture There were no problems after we started the planning, adaptation, and execution services. The mocked planning service terminated after it had sent the predefined PAM and CAM to the adaptation service. The models were stored in the service’s working directory. The adaptation service sent the execution plan, as well as PAM and CAM to the execution service. They were stored in the working directory of the execution service. The adaptation and execution services continued running. More predefined PCM models could be sent to them by restarting the mocked planning service repeatedly.

5.6.3 Migration and (De-) Allocation Scenario

T1: Creation of an Execution Plan The execution plan shown in Listing 5.9 was created during the migration scenario. It contains the following actions:

- AllocateNodeAction in lines 2-4 referencing the target resource container from the CAM.
- ConnectNodeAction in lines 5-9 which references the target resource container and its two attached connectors from the CAM.
- DeployComponentAction in lines 10-12 which references the target allocation context from the CAM.
- MigrateComponentStateAction in lines 13-16 which references the target allocation context from the CAM and the source allocation context from the PAM.
- ConnectComponentAction in lines 17-21 which references the target allocation context and its providing and requiring allocation contexts from the CAM.
- BlockRequestsToComponentAction in lines 22-25 which references the target allocation context and one target requiring allocation context from the PAM.
- FinishComponentAction in lines 26-28 which references the target allocation context from the PAM.
- DisconnectComponentAction in lines 29-33 which references the target allocation context and one target providing allocation context, as well as one target requiring allocation context from the PAM.
- UndeployComponentAction in lines 34-36 which references the target allocation context from the PAM.
- DisconnectNodeAction in lines 37-41 which references the target resource container and its two attached connectors from the PAM.
- DeallocateNodeAction in lines 42-44 which references the target resource container from the PAM.
5. Evaluation

Listing 5.9. Serialized execution plan in migration scenario

```
<org.iobserve:ExecutionPlan>
  <actions xsi:type="org.iobserve:AllocateNodeAction">
    <targetResourceContainer href="redeploymentmodel#_K6XZkFnMxEEiXof1znnLDQw"/>
  </actions>
  <actions xsi:type="org.iobserve:ConnectNodeAction">
    <targetResourceContainer href="redeploymentmodel#_K6XZkFnMxEEiXof1znnLDQw"/>
    <targetConnectors href="redeploymentmodel#_895EAA1xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:AllocateNodeAction">
    <targetResourceContainer href="redeploymentmodel#_K6XZkFnMxEEiXof1znnLDQw"/>
    <targetConnectors href="redeploymentmodel#_895EAA1xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:ConnectComponentAction">
    <targetAllocationContext href="redeploymentmodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
    <sourceAllocationContext href="runtimemodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:MigrateComponentStateAction">
    <targetAllocationContext href="redeploymentmodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
    <sourceAllocationContext href="runtimemodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:DeployComponentAction">
    <targetAllocationContext href="redeploymentmodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:MigrateComponentStateAction">
    <targetAllocationContext href="redeploymentmodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
    <sourceAllocationContext href="runtimemodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:ConnectComponentAction">
    <targetAllocationContext href="redeploymentmodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
    <sourceAllocationContext href="runtimemodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:FinishComponentAction">
    <targetAllocationContext href="runtimemodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:DisconnectComponentAction">
    <targetAllocationContext href="runtimemodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
    <sourceAllocationContext href="runtimemodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:UndeployComponentAction">
    <targetAllocationContext href="runtimemodel#_1LRP0AA6xEEiMaMdqMnouQ"/>
  </actions>
  <actions xsi:type="org.iobserve:DeallocateNodeAction">
    <targetResourceContainer href="runtimemodel#_rohggAA1xEEiMaMdqMnouQ"/>
    <targetConnectors href="runtimemodel#_895EAA1xEEiMaMdqMnouQ"/>
    <targetConnectors href="runtimemodel#_895EAA1xEEiMaMdqMnouQ"/>
  </actions>
</org.iobserve:ExecutionPlan>
```

78
5.6. Results

T2: Application of an Execution Plan to the System  Listing 5.10 shows the Kubernetes services during the experiment. The JPetStore service shown in line 2 of Listing 5.10 remained active throughout the entire experiment.

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpetstore</td>
<td>ClusterIP</td>
<td>10.97.178.220</td>
<td>&lt;none&gt;</td>
<td>8080/TCP</td>
<td>56s</td>
</tr>
</tbody>
</table>

Listing 5.10. Output of `kubectl get services` during migration scenario

Listing 5.11 shows how the Kubernetes deployments changed during the experiment. In the first state shown in lines 1-5 we see the cluster’s initial configuration before the execution plan was executed. We have one instance of each deployment running. In the second state shown in lines 7-11 we see the first monitored state after the migration has taken place. The account deployment has disappeared and instead there is an account2 deployment in line 8 which is not up to date yet. In the third and final monitored state in lines 13-17 we see the same deployment as in the previous state. In this state, however, the account2 deployment is up-to-date now as well.

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>57s</td>
</tr>
<tr>
<td>catalog</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>57s</td>
</tr>
<tr>
<td>frontend</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>57s</td>
</tr>
<tr>
<td>order</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>57s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1s</td>
</tr>
<tr>
<td>catalog</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>58s</td>
</tr>
<tr>
<td>frontend</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>58s</td>
</tr>
<tr>
<td>order</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>58s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2s</td>
</tr>
<tr>
<td>catalog</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>59s</td>
</tr>
<tr>
<td>frontend</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>59s</td>
</tr>
<tr>
<td>order</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>59s</td>
</tr>
</tbody>
</table>

Listing 5.11. Outputs of `kubectl get deployments` during migration scenario

In Listing 5.12, we see the different states of the pods throughout the experiment. The first state in lines 1-5 shows the cluster’s initial configuration before the execution plan was executed. One pod of each component is running. In the second state shown in lines 7-12 we see the first monitored state after the migration has taken place. A new pod with the name account2 was created in line 9. This new pod is not ready yet and its status indicates that a container is currently being created for it. The third state in lines 14-19 shows that
the pod with the name account is currently terminating in line 15. The account2 is still in
the container creation state and not ready yet. In the fourth state in lines 21-26 this changes.
The account2 pod in line 23 is running now. The account pod is still terminating. In the
final state in lines 28-32 the account pod terminated and was removed. All other pods are
running.

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>account-546cddf8f-4fjlp</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>56s</td>
</tr>
<tr>
<td>catalog-67c666c6f-ss7s4</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>56s</td>
</tr>
<tr>
<td>frontend-69b8d9cf76-26h8r</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>56s</td>
</tr>
<tr>
<td>order-59d6d9bdf4-z42l6</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>56s</td>
</tr>
<tr>
<td>account-546cddf8f-4fjlp</td>
<td>1/1</td>
<td>Terminating</td>
<td>0</td>
<td>57s</td>
</tr>
<tr>
<td>account2-59f54f677b-qdvsf6</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>0s</td>
</tr>
<tr>
<td>catalog-67c666c6f-ss7s4</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>57s</td>
</tr>
<tr>
<td>frontend-69b8d9cf76-26h8r</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>57s</td>
</tr>
<tr>
<td>order-59d6d9bdf4-z42l6</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>57s</td>
</tr>
<tr>
<td>account-546cddf8f-4fjlp</td>
<td>1/1</td>
<td>Terminating</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>account2-59f54f677b-qdvsf6</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3s</td>
</tr>
<tr>
<td>catalog-67c666c6f-ss7s4</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>frontend-69b8d9cf76-26h8r</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>order-59d6d9bdf4-z42l6</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>account-59f54f677b-qdvsf6</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>33s</td>
</tr>
<tr>
<td>catalog-67c666c6f-ss7s4</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>frontend-69b8d9cf76-26h8r</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>order-59d6d9bdf4-z42l6</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
</tbody>
</table>

Listing 5.12. Outputs of `kubectl get pods` during migration scenario
T3: Expected Behavior of the Service Based Architecture  We encountered no problems after we started the planning, adaptation, and execution services. The mocked planning service terminated after it had sent the predefined PAM and CAM to the adaptation service. The models were stored in the service’s working directory. The adaptation service sent the execution plan, as well as PAM and CAM to the execution service in whose working directory they were stored. The adaptation and execution service continued running before being terminated by our execution script.

5.7 Discussion

In this section we discuss our evaluation results presented in the previous section.

T1  In the replication scenario our input models described a replication and in our dereplication scenario they specified a dereplication. In the migration scenario, our input models caused an allocation followed by a migration and a deallocation. We expected that these composed actions were recognized and transformed into the correspondent atomic adaptation actions as defined in Section 4.2.3. Looking at the execution plans of all three scenarios, we can see that each execution plan contains the expected atomic adaptation actions. Additionally, the atomic action’s attributes reference the correct model components of PAM and CAM for each execution plan. For this reason, we can say that the execution plans were generated correctly for the given scenarios. We can not make certain statements about other scenarios. However, our scenarios cover all adaptation actions except for the replacement of a repository component. Our evaluation scenarios are minimalistic and therefore arbitrary composed adaptation scenarios can be constructed from our scenarios. Therefore, we can suppose that the generation of execution plans may also deliver correct results in composed scenarios.

T2  In the replication and dereplication scenarios, the respective actions were applied to the running JPetStore as expected. New instances were deployed and undeployed without a problem. However, we discovered an unexpected state in the migration scenario. It is depicted in lines 7-11 of Listing 5.11: The account deployment is not present any more. The account2 deployment has already been created but is not available yet. Taking a look at the pods in Listing 5.12 confirms this problem. In lines 7-12 we see that while the account pod (line 8) is still running while the account2 pod (line 9) is being created. There is no problem with this, so far. However, in lines 14-19 we see that the status of the account pod (line 15) has changed to terminating while the account2 pod is still being creating. At this point of time, there is no running instance of the JPetStore’s account component. In the following state in lines 21-26 we see that the account2 component is finally running. Nevertheless, looking at the age of the account2 in the described states, we can see that it took three seconds to get it running. With the account pod terminating at the second described state, there is an interval of at least one second where the JPetStore had no active
5. Evaluation

account component. As mentioned with respect to T1, the execution plan for the migration scenario is correct. The fault results from the fact that our execution implementation for Kubernetes does not receive feedback from the cluster on whether the adaptation actions finished. It simply executes them in the given order. To fix the described challenge, the deployment action has to wait until the deployment becomes available. We implemented waiting as an adhoc solution to the challenge after this evaluation. An alternative future solution is discussed in Section 7.2.

T3 In all three scenarios the mocked planning service, the adaptation and the execution services worked as expected. The models and execution plan were serialized and sent without issues. When models or execution plan were sent multiple times to simulate multiple iterations of iObserve’s control loop their old versions were overwritten. With iObserve running on the Kubernetes master, there were no issues accessing Kubernetes.

5.8 Threads to Validity

A limitation of our evaluation is that we only used a mocked planning phase which outputs predefined system adaptation. Due to technical issues with the PerOpteryx application, there was no other way to provide PAM and CAM instances and to use predefined scenarios. Real CAMs created by PerOpteryx after the application of simulated workloads on the observed system could lead to unexpected adaptation scenarios. Additionally, they were closer to real world scenarios than our minimal adaptation scenarios. Nevertheless, the latter represent all adaptation actions except for the exchange of repository components as well. The lack of a scenario for this action is another thread to validity. In the JPetStore application no exchangeable component instances were present which is why this scenario could not be evaluated. Thus, we cannot assume anything about the correctness of our approach concerning the exchange of repository components.

We only used the relatively small JPetStore application. There may occur performance issues with larger systems. The PerOpteryx optimization algorithm, the initialization of the used rule engine, the sequential atomic action computation and the sequential execution of adaptation actions are potential performance bottlenecks. With our logging of the Kubernetes cluster’s states of services, deployments, and pods the logging interval of one second is another thread to validity. We might have missed unexpected states between the logging outputs. Nevertheless, we were able to capture the key states with this interval.

5.9 Summary

The feasibility study shows that our approach works correctly in terms of the generation of execution plans for the given scenarios. We encountered a problem during the execution
of these plans when the JPetStore account component was briefly unavailable during the migration scenario. The root cause of the fault is that the execution service does not wait until a deployed component is actually available before it continues to process the execution plan. We fixed this issue after the evaluation. iObserve’s service based architecture worked correctly throughout this evaluation.
In this chapter, we present related work with respect to our approach. The approaches presented in this chapter focus to the computation of system adaptations and relate to the MAKE-K approach.

6.1 SLAstic

The SLAstic framework [Van Hoorn et al. 2009] focuses on the reduction of operating costs of software systems while remaining conform to certain service level agreements (SLAs). Like iObserve, the SLAstic framework therefore promotes the runtime adaptation of software systems.

We referred to the SLAstic approach during the definition of our composed adaptation actions in Section 4.2.3. Like iObserve, SLAstic builds upon a model-based approach using PCM architecture models for the computation of system adaptations. SLAstic is based on the MAPE-K approach and contains the the three technology independent phases observation, analysis, and adaptation in its control loop. Unlike iObserve, the loop does not contain the monitoring and execution phases. These technology dependent phases implemented as separate components in SLAstic. As described in Section 4.3.3, we also addressed the topic of technology dependent implementations in the context of iObserve’s execution service by providing an appropriate interface. Additionally, the framework features advanced components such as SLAstic.SIM enabling the complete simulation of model-based system architectures [Von Massow et al. 2011]. Such components are not part of iObserve yet.

6.2 DiVA

The DiVA\(^1\) project is another approach focusing on adaptive software systems. Within the project’s life cycle different approaches were presented to one of which we referred in this thesis. The project is targeted towards dynamic software product lines and therefore also

addresses topics such as managing co-existent and co-dependent feature configurations of a software system. In their contributions to the project Morin et al. [2008] conduct research on the use of a model-based approach. As iObserve, their approach is also based around a source and a target model.

Being focused on dynamic software product lines, possible adaptations in this approach relate to adapting provided features at runtime. The concept of aspect oriented modeling is used to weave new aspects into the source model and thereby creating the target model which represents a system with those new features. Morin et al. [2009] use EMF Compare\(^2\) to create a diff model and a match model containing differences and similarities between the source and target model. Then they also derive adaptation commands from the differences and similarities.

The approach contains several similar approaches to iObserve such as the source and target models and the concept of comparing the models to derive adaptation commands. However, this approach is focused on the adaptation of features in dynamic software product lines. Therefore, it adapts a system to modify certain functionalities while iObserve adapts a system to reach certain goals such as performance or privacy goals.

6.3 Adaptive Knowledge Bases for Self-Adaptive Systems

Klös et al. [2018] present an approach based on adaptive knowledge bases. Like iObserve, the approach features a MAPE-K loop. The key components are rules of the style $r : g \& c_1; c_2; \ldots; c_n \rightarrow \text{effect after time}$. This means that a rule $r$ is applicable if its guard $g$ is satisfied. If the rule is chosen, the commands $c_1; c_2; \ldots; c_n$ are applied to the observed system. A certain effect is assumed to be observable after a certain amount of time. They extend the MAPE-K loop by adding an evaluation and a learning phase.

The approach uses models of the current system environment and specific system goals. In the analysis phase, they use distance functions to capture the distance between a system’s current state and the desired system goals. Additionally, the evaluation is invoked in this phase to evaluate the previous effects of rules to the system and can disable rules which do not cope with the current state of the environment. If the analysis has detected the need for an adaptation, the best available rule is chosen in the planning phase. If no suitable rule is found for a certain system state or if the rules have to be updated due to topology changes, the learning phase is triggered. The learning phase adapts and refines the rule base. Learning new rules dynamically enables the use of specific rules for specific components. This is not possible in an approach with a static rule base. Finally, the commands from the chosen rule are applied to the system in the execution phase [Klös

6.4 CDOXplorer

A key motivation of Klös et al. [2018] is to reduce the complexity of common planning and adaptation logics. Due to the explicit application condition and expected effects of timed adaptation rules, they are able to make adaptation decisions comprehensible and explicit. With their approach of timed adaptation rules they also address problems like we faced in our evaluation, where a deployment was only available after a certain amount of time. Especially the focus on comprehensible decision making is an interesting aspect of this approach. The complexity of adaptation decisions is still a challenge with common approaches such as the evolutionary algorithm of PerOpteryx. Overall, the approach of Klös et al. [2018] is more formal and more focused on rules than iObserve.

6.4 CDOXplorer

The genetic algorithm CDOXplorer aims at finding cloud deployment options (CDOs) [Frey et al. 2013]. CDOs describe how software systems can be deployed in a cloud based environment with respect to the environment, the deployment architecture, and the runtime reconfiguration rules. The algorithm is capable of finding Pareto-optimal solutions and an evaluation showed that these solutions surpass those of other state-of-the-art techniques by up to 60%.

The CDOXplorer might be a reasonable replacement for the PerOpteryx algorithm which we tried to use without success in this thesis. However, the algorithm requires an architectural model, a so-called status-quo deployment model, a workload profile, and cloud profile as inputs. We do not know if generating these models from the PCM models used by iObserve is a reasonable approach. CDOXplorer outputs the optimized architecture represented by a so-called cloud deployment model. Like the input models, this model would have to be converted to a PCM model in iObserve. Additionally, the output contains so called reconfiguration rules for cost-efficient dynamic resource scaling according to observed usage patterns. PerOpteryx does not provide similar output. However, these reconfiguration rules can be of interest with respect to our rule based approach to the computation of execution plans.
Conclusions and Future Work

In this chapter, we conclude our thesis by reflecting on the goals and research questions we defined in Section 1.2. Furthermore, we present topics which can be addressed in future works.

7.1 Conclusions

Our first goal was the integration of the existing PerOpteryx optimization tool into the iObserve architecture. With reference to our first research question RQ 1.1, we must admit that we were not able to integrate the PerOpteryx into iObserve to compute a candidate architecture as described in Section 4.1. While attempts to an integration as a dependency failed completely, an integration with Gradle appeared more promising. However, we were not able to provide a functioning solution, either. The automated export of an Eclipse RCP application product still fails due to problems with the large number of plug-in dependencies and even a manually exported product resulted in an unstable version of PerOpteryx. At least, the RCP application project is now automatically imported and compiled with Gradle during iObserve’s building process. Architecture wise we were able to provide the planning phase as a separate service. We simplified the internal Pipe-and-Filter architecture. Despite the problems with PerOpteryx, the planning phase itself with the model preprocessing and the call of PerOpteryx remains functional. It may be replaced by filters for an alternative optimizing approach. Referring to RQ 1.2, we provide a service-based planning phase whose Pipe-and-Filter architecture is flexible enough to support alternative planning algorithms. However, we provide by no means a framework for exchangeable optimization algorithms.

Our second goal aimed at the improvement of the execution plan computation method from the existing approach to address the observed system’s availability. With respect to RQ 2.1, we presented dependencies of the observed systems in 4.2 and pointed out why the order of execution is also relevant to ensure the system works properly. In our evaluation in Chapter 5 we got a non properly working system because the deployment action from our execution plan returned before its execution had finished. Thus, we present several scenarios where an execution plan which does not take into account all dependencies leads to a not properly system in this thesis. We identified a basic set of specific rules to
detect adaptation actions as demanded in RQ 2.2. We replaced the nested comparison algorithm Pöppke [2017] and Weimann [2017] used to identify adaptation actions with these rules. So far, our rule based approach does not provide more functionalities than the existing approach. However, it provides more extendability for the future. Additionally, we found an error in the existing approach during our rule based implementation. The semantics of assembly and allocation contexts were misinterpreted and mixed which we fixed in this thesis. We addressed dependencies between different components in the observed system by updating the existing composed adaptation actions and introducing the new set of atomic adaptation actions. With the adaptation and execution services, we created two separate services for the computation of execution plans and their actual execution. The execution service provides a dedicated interface for cloud-provider specific implementations as requested in RQ 2.3. We developed a concrete implementation for Kubernetes. Even though the adaptation service does not actively support an interface for different computation mechanisms, its internal Pipe-and-Filter architecture supports an easy implementation of alternative approaches.

As described in Chapter 5, we evaluated our approach which was our third goal. We conducted a feasibility study in three different scenarios with the JPetStore web application. Due to the problems with PerOpteryx, the planning service was mocked and we used predefined models for the different scenarios. The evaluation showed that the amount of time it takes for a component to actually become available after its deployment is triggered has to be taken into account as well. We adapted our Kubernetes implementation to take this into account and wait until a deployment actually becomes available after its execution. Besides this issue, the evaluation showed that the adaptation and execution services of our approach work as expected.

Summary We had several issues with the integration of PerOpteryx as an architecture optimization algorithm. Therefore, we did not manage to finally provide a functioning planning service in this thesis. However, we provide a foundation for future attempts with our efforts towards an integration with Gradle and the new service based architecture. Our implementation of the adaptation and execution services evolve on the existing approach by taking into account component dependencies. Both services make use of new concepts such as a rule based approach with Drools and the execution on a Kubernetes cluster.

7.2 Future Work

PerOpteryx Integration We still believe that the automated export of an PerOpteryx RCP application with the Goomph plug-in for Gradle is possible. It worked without problems in a minimalistic example application. However, the amount of dependencies PerOpteryx requires consumed too much time to finish the debugging of the build scripts during the course of this thesis. This could be the goal of future works. However, we have
to also note that a manually exported product of the PerOpteryx RCP application crashed due to unknown reasons. Finding a solution here is another goal.

Isolating the required PerOpteryx core functionalities into a more lightweight version with less dependencies or directly into iObserve filters can be an alternative. Pöppke [2017] and Weimann [2017] already proposed this in their theses. At the same time, they also stressed the immense difficulty of such an approach due to the plug-in’s complexity. We fully agree with them in this regard. The iObserve approach would benefit hugely from a direct integration. Not only would it eliminate the problems with the integration of an eclipse plug-in into a Java program. It would remove the need to call external executables for PerOpteryx and the required LQN solver, as well. However, such an integration project does not add many new scientific insights for the amount of risk and complexity which come with it.

Change of Repository Components  We implemented the change of PCM repository components similar to a migration action in terms of the required atomic adaptation actions. However, we could not ensure its correctness because the JPetStore we used does not provide alternative component implementations. Possibly, an entirely different solution is needed because Oreizy et al. [1998] state that “both components must not be simultaneously active during the change”. With the same atomic adaptation actions as for a migration the old and the new component would be active simultaneously in our approach. An exact evaluation and alternative solution may be addressed in future works.

Rules for Detecting Certain Patterns  Our rule based approach contains basic rules to detect composed adaptation actions. So far, there is no order of adaptation actions of the same type. However, such an order would make sense in certain scenarios. An example are tightly coupled components deployed on the same resource container which are both migrated to one other resource container. It makes sense to migrate both components directly after each other to prevent increased network traffic while they are located on different resource containers. Future works can refine our rule based approach and add rules to detect such scenarios.

Runtime Execution Evaluation  In our approach the execution simply executes the adaptation actions in the received execution plan. One problem with this approach became obvious in our evaluation: Our first implementation of a deployment returned before the deployed component had actually become available. Generally, there is no mechanism to evaluate if the observed system is transformed to the expected state, yet. Future works can implement such a mechanism to check the conformance between candidate-architecture-model and the observed system after the execution of adaptation actions.
7. Conclusions and Future Work

Compatibility to Different Cloud Infrastructures  Pöppke [2017] and Weimann [2017] used the JClouds middleware to ensure their approach remains compatible to different cloud provider infrastructures. Our execution provides an interface for infrastructure specific implementations. We created an implementation for Kubernetes clusters which are not supported by JClouds at this point of time. However, the evolution of the adaptation and execution services in our approach with the introduction of atomic actions and a dedicated execution plan are not compatible with the old implementation anymore. Therefore, the old JClouds implementation has to be adapted to the new execution service architecture in the future.
Bibliography


Bibliography


Bibliography


Appendix
Appendix

A.1 LQN Solver Installation Instructions

The following instructions explain how the LQN solver\(^1\) required by PerOpteryx can be installed.

Required tools:

- make
- autoconf
- automake
- autoreconf
- gcc
- flex
- bison

Commands:

```
git clone https://github.com/layeredqueuing/V5.git
cd V5
```

The solver can be installed afterwards with the following commands:

```
autoreconf -install
autoconf
automake
./configure -prefix=<path>
```

A `bin` and a `lib` directory are created at the specified path. The default path is `/usr/local/` if no path is specified.

```
make all
```

If this fails (e.g. due to `parasol`), try the following:

```
cd lqiolib
make all
(sudo) make install
```

Use `sudo` if you did not set a prefix. Only the root user is allowed to write to `/usr/local/`.

\(^1\)Layered Queuing V5 LQN Solver. URL: https://github.com/layeredqueuing/V5 (visited on 16/03/2018).
A.2 Required PerOpteryx and Palladio Libraries

The following libraries were used in the attempt to the integration of PerOpteryx as a dependency as described in Section 4.1.3:

- de.uka.ipd.sdq.dsexplore.analysis.cost_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore.analysis.lqn_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore.analysis.reliability_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore.analysis.simucom_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore.bayesnets_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore.qml.edit_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore.qml.editor_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore.qml.handling_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore.qml_1.0.0.201709290946.jar
- de.uka.ipd.sdq.dsexplore_1.3.0.201709290946.jar
- de.uka.ipd.sdq.simucomframework_4.1.0.201709290901.jar
- de.uka.ipd.sdq.simulation_4.1.0.201709290901.jar
- org.eclipse.core.contenttype.source_3.6.0.v20170207-1037.jar
- org.eclipse.core.contenttype_3.6.0.v20170207-1037.jar
- org.eclipse.core.jobs.source_3.9.1.v20170714-0547.jar
- org.eclipse.core.jobs_3.9.1.v20170714-0547.jar
- org.eclipse.core.resources.source_3.12.0.v20170417-1558.jar
- org.eclipse.core.resources_3.12.0.v20170417-1558.jar
- org.eclipse.core.runtime.source_3.13.0.v20170605-1534.jar
- org.eclipse.core.runtime_3.13.0.v20170605-1534.jar
- org.eclipse.debug.core.source_3.11.0.v20170605-1534.jar
- org.eclipse.debug.core_3.11.0.v20170605-1534.jar
- org.eclipse.equinox.app.source_1.3.400.v20150715-1528.jar
- org.eclipse.equinox.app_1.3.400.v20150715-1528.jar
- org.eclipse.equinox.preferences.source_3.7.0.v20170826-2132.jar
- org.eclipse.equinox.preferences_3.7.0.v20170826-2132.jar
- org.eclipse.equinox.registry.source_3.7.0.v201708222-1344.jar
- org.eclipse.equinox.registry_3.7.0.v201708222-1344.jar
- org.eclipse.osgi.source_3.12.50.v20170828-1321.jar
- org.eclipse.osgi_3.12.50.v20170828-1321.jar
- org.palladiosimulator.edp2.dao_1.0.0.201608091414.jar
- org.palladiosimulator.edp2_2.0.0.201608091414.jar
- org.palladiosimulator.recorderframework.edp2_2.0.2.201603031355.jar
- org.palladiosimulator.recorderframework_2.0.1.201603031355.jar
- org.palladiosimulator.reliability_3.1.0.201511051309.jar