Karlsruhe Institute of Technology

Software Design and Quality

Design Space Exploration for Adaptation Planning in Cloud-based Applications

Master's Thesis of

Tobias Pöppke

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf Reussner
Second reviewer:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor:	Dr. rer. nat. Robert Heinrich
Second advisor:	Dipl.-Inform. Kiana Rostami

15. November 2016 – 15. June 2017

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 14.06.2017

. . . .

(Tobias Pöppke)

Abstract

With the widespread adoption of cloud computing and increasing interest by software engineers to develop cloud-based applications, automatic adaptation which optimizes the performance is becoming an important topic. However, this kind of automatic adaptation of applications is an inherently complex task and there are limitations to fully automatic adaptation where a human operator is required to step in. Previous work in this area has neglected these limits and focused on enabling fully automatic adaptation, whereas we propose an approach that integrates automatic adaptation, optimizing the application's performance, as well as an operator-in-the-loop. We do so by using, PerOpteryx, an existing design space exploration tool based on architectural run-time models and combine it with a model-based adaptation planning and execution approach that allows for an operator to take control of the execution if needed. We implement our approach and evaluate it's accuracy and scalability through a series of experiments, which show that our approach is accurate and scales well for most cloud-based applications. Our approach enables the use of automatic adaptation with a focus on performance optimization, while simultaneously allowing operators to step in, which significantly eases the development and maintenance of cloud-based applications with good performance qualities.

Zusammenfassung

Durch die steigende Verbreitung von Cloud Computing und dem wachsenden Interesse von Software Ingenieuren, cloud-basierte Anwendungen zu entwickeln, wird die automatische Adaption zur Leistungsoptimierung dieser Anwendungen zu einem wichtigen Forschungsthema. Diese Art der Adaption einer Anwendung ist jedoch eine inhärent komplexe Aufgabe mit Grenzen für eine vollautomatische Adaption, bei deren erreichen ein menschlicher Bediener eingreifen muss. Vorherige Arbeiten auf diesem Gebiet haben diese Grenzen bisher vernachlässigt und sich vornehmlich der Möglichkeiten zur vollautomatischen Adaption angenommen. Wir stellen jedoch einen Ansatz vor, der die automatische Adaption mit Fokus auf die Leistungsoptimierung mit einem Ansatz zur Bedienerintegration vereint. Unser Ansatz verwendt PerOpteryx, ein existierendes Werkzeug zur automatischen Entwurfsraumexploration, um Laufzeit-Architekturmodelle der Anwendung zu optimieren und mit einem Modell-basierten Ansatz zur Adaptionsplanung und -ausführung zu kombinieren, der Bedienereingriffe während der Adaptionsausführung ermöglicht. Wir implementiern unseren Ansatz und evaluieren seine Korrektheit und Skalierbarkeit mit einer Reihe von Experimenten, die zeigen, dass unser Ansatz korrekt funktioniert und für die meisten cloud-basierten Anwendungen gut skaliert. Der vorgestellte Ansatz ermöglicht daher die Nutzung von automatische Adaptionsplanung mit einem Fokus auf Leistungsoptimierung und erlaubt gleichzeitig einem Bediener bei Problemen einzugreifen. Dieser Ansatz verringert daher den Aufwand für die Entwicklung und Pflege von cloud-basierten Anwendungen mit guten Leistungsmerkmalen.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In recent years, cloud computing has evolved from a hype to an important model to consider when building complex software systems. The NIST definition of cloud computing [27] defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

There are many advantages to adopting a cloud-based architecture for software systems. One of them is the possibility for a customer to provision or release resources, generally through a self-service process. The cloud provider ensures that the requested resource is available for the customer in a matter of minutes or seconds, which creates the impression of an unlimited supply of resources. An application can therefore be designed as if there were no limitations to the available computing resources. Because the resources are shared with other customers, they are also more evenly utilized.

This rapid provisioning, or rapid elasticity [27], of computing resources enables a payment model that is based on the actual resource usage of a customer. This pay-as-you-go payment model is used by most cloud providers and relieves engineers from up-front investments in an application's hardware and software infrastructure. The engineer only has to pay for the resources she uses and can release them at any time, if necessary, to reduce costs. Moreover, the needed infrastructure does not necessarily have to be planned in detail to fit the needs, because it can be adjusted as needed without additional costs other than those charged by the cloud provider.

Most importantly, rapid elasticity enables software engineers to react to performance issues that arise during operation. If a performance bottleneck is detected, additional cloud resources can be acquired in a short time period and integrated into the software system to mitigate the issue. If the performance bottleneck was triggered by a temporary external event, like, for example, a commercial airing on television, the additional resources can again be unprovisioned after the effects of the event have worn off. Thus, with rapid elasticity, it is possible to adjust the performance qualities of a cloud-based application at run time.

Another important aspect of cloud computing is the pooling of resources under the maintenance and management of the cloud provider. This enables companies or individuals to solely focus on the software development process without needing to manage, maintain and secure the, potentially complex, server infrastructure. Because the management of such infrastructures comes with many difficulties and requires a lot of expertise, it may be beneficial and more cost efficient to hand this task over to a cloud provider.

These advantages lead to an increased usage of the services of cloud providers and other third parties. Those services are not necessarily under the control of the software engineers who consume them and include the infrastructure services of cloud providers, as well as software maintained by third parties that can be accessed via the Internet or are included in the cloud providers services. They expose their interfaces to the consumers, which can, in turn, use them to execute specific computation tasks or allocate resources.

According to the NIST definition of cloud computing, there are three service models. The *Software as a Service (SaaS)* service model enables a consumer to use an application, which is managed completely by the cloud provider. The consumer does not have any other means of accessing the service than through the provided interfaces. Another service model, *Platform as a Service (PaaS)*, enables the consumer to deploy her own application onto a provided platform, which is again managed by the cloud provider. In this model, the consumer may have control over some specific configuration settings of the environment, also limited by the cloud provider's interfaces. In the *Infrastructure as a Service (IaaS)* service model, the consumer can acquire cloud resources and use it to deploy whatever applications she wants. Only the underlying infrastructure is managed by the cloud provider.

Although cloud computing brings with it a variety of advantages for software engineers, it also introduces challenging problems which are described further in the following section.

1.2 Problem Statement

The problems that arise by using cloud services are [18, 19]: (I) For SaaS and PaaS services, engineers only have a limited or no view on the inner workings of those services and their infrastructure. Incorporating such services into an application running on an IaaS service can free engineers from implementing commonly needed functionality, like message queues, but it also introduces a dependency on those services. If the service provider chooses to shut down the service or if the service does no longer satisfy the performance requirements, the application has no way of preventing it. Instead it is necessary to adapt the application to the new circumstances. Such changes can normally not be foreseen in advance and have to be dealt with at run time.

(II) The cloud provider may decide that, in order to improve the utilization of hardware resources, it is necessary to move parts or the whole application to another location inside the data center or even into a completely different data center. This may in turn lead to a decreased performance of the application, for example higher response times. Therefore, cloud-based applications have to be able to adapt to such changes in their execution environment.

(III) Cloud-based applications are inherently more complex than monolithic applications because of the distributed nature of cloud computing. They have to be designed to cope with distributed application components as well as with data that is distributed. Most cloud-based applications also have to satisfy performance constraints while being cost-efficient at the same time. Due to the complexity of the applications, it is difficult to find a deployment that best satisfies all of these constraints.

(IV) External events may impact the application as well. Natural disasters, economic difficulties or just a successful sales campaign may overburden the application and lead to decreased performance. Such events can also not be foreseen during the development of a cloud-based application. Moreover, they can not be deducted by only using monitoring data and require additional information from outside the system.

These problems not only raise the difficulty of initially developing a cloud-based system but also limit the applicability of fully automatic adaptation. Especially the problems described in (IV) make fully automatic adaptation a nearly impossible task because of the number of possible scenarios that have to be considered for comprehensive decision making. However, it is imperative for a cloud-based application to adapt to changes in it's execution environment.

This means, that it is necessary to find a way to automatically adapt a cloud-based application, while at the same time leaving room for a human operator to control and influence the process if necessary. This ensures, that external events are considered during the adaptation process as well as other information that is normally not available to a fully automatic approach, such as information about an expected increase in the number of users.

When confronted with a problem of category (I), it may also become necessary to initiate development activities to adapt the application to the changed interfaces or to use a different service. Those activities belong to the realm of software evolution, which we define as a longer sequence of modifications to a software system over its life-time which is manually applied by software engineers (cf. [24]). Such tasks can not be performed in a fully automated way, and must therefore be treated separately. But software evolution is intertwined with software adaptation, because software evolution influences the possibilities for adaptation and the adaptation process delivers information about the problems that can only be solved by evolving the application. It is also possible that a performance bottleneck has the short-term solution of adding more resources, but needs to be tackled by software engineers to remove the bottleneck in the long run.

These processes should therefore rely on the same basis of data and keep it updated, so that each process can use the newest information about the system and not rely on outdated data, such as models that do no longer reflect the actual state of the application.

The combination of automatic adaptation with an operator-in-the-loop for cloud-based applications is an open problem that is being adressed in this thesis. We will focus on adaptations for performance related issues in an IaaS environment, namely (de-)allocating virtual machines, (un-)deploying components, migrating and replicating components. The goals and research questions are described in detail in the following section.

1.3 Goals and Research Questions

Because of the problems mentioned before and the complexity introduced by cloud-based applications, this thesis explores the possibilities for automatic adaptation planning and execution with an operator-in-the-loop approach. Our approach is based on architectural run-time models as the basis for adaptation planning and execution, because they can be understood by humans and can also be used during the software evolution process. We focus on cloud-based applications that use the IaaS service model, because in the context of PaaS, the control over performance related adaptations is mainly in the hands of the cloud provider.

We use the Goal Question Metric (GQM) process [2] to develop our approach. This process directs the development towards fulfilling specific goals in a measurable way. To do so, GQM uses a top down approach by first defining goals on a conceptual level. These goals are further refined by deriving research questions from them, which in turn lead to the definition of metrics. The goals, questions and metrics form the GQM plan. As the first part of this plan, we define the goals of this thesis as follows.

G-1 Enable automatic adaptation planning in cloud-based applications for performance issues.

G-2 Enable automatic execution of adaptations mitigating performance issues in cloud-based applications.

The research questions derived from **G-1** reflect the steps performed by our proposed approach and are closely related to the modules that were implemented.

RQ-1.1 Can architectural run-time models be used to plan accurate performance improving adaptation actions?

RQ-1.2 Is the derivation of an adaptation plan possible in a scalable way?

For **RQ-1.1**, we want to show the accuracy of the approach to derive adaptation plans that improve the application's architecture in a given scenario with respect to several goals like performance and costs. The reason for triggering the adaptation planning is usually a degradation in the application's quality-of-service properties because of internal or external events. Therefore the adaptation planning has to derive a target architecture that improves these properties again or at least prevents further degradation. We want to make sure that an improved architectural run-time model can be transformed into concrete steps to adapt the applications current architecture into the derived target architecture. This is necessary, because the planning process would not be sensible, if there is no possibility to know the steps necessary to actually adapt the current architecture.

The question **RQ-1.2** is concerned with the scalability of the adaptation planning. It is necessary for a cloud-based application to react to changes in the execution environment in a timely manner. If a central system component is no longer available, this may lead to an unavailable application, which in turn can cause a violation of service level agreements (SLAs) or other serious problems for the consumers of the application. It is therefore necessary to have an adaptation planning routine, that is able to minimize the time needed to deal with such problems.

The second goal, **G-2**, relates to the approach's ability to not only plan an adaptation, but also to execute it in an automated way. The following research questions are derived from this goal.

RQ-2.1 Can the adaptation execution be performed automatically for a complex cloud-based application?

RQ-2.2 How scalable is the adaptation execution for a complex cloud-based application?

RQ-2.1 explores the accuracy of our approach to execute the derived adaptation plans in an automatic way. An operator might not always be available, or the operator does not have the required knowledge to make the decisions needed to execute an adaptation plan. Therefore the adaptation plan should be executed as automated as possible and guide the operator, if a problem arises. For a complex cloud-based application, incorrect execution of the adaptation plan could lead, for example, to limited availability of the application or a loss of functionality.

As with **RQ-1.2**, the research question **RQ-2.2** is concerned with the scalability of our approach. It is necessary for the adaptation to be executed in a timely manner, to minimize the disruption to the application caused by such an adaptation.

The metrics and methods used to quantify these research questions are detailed in Section 7.

1.4 Contributions

This thesis contributes an approach to integrate automatic adaptation planning through design space exploration with an operator-in-the-loop. To enable this integration, the following contributions are provided in this thesis:

- A model for representing the available computing resources a cloud provider offers is presented.
- A model for representing actions in an adaptation plan is presented.
- The iObserve megamodel is extended with the steps for adaptation planning and execution.
- An algorithm for deriving degrees of freedom from an architectural run-time model is developed.
- Design space exploration is applied to architectural run-time models in the iObserve context and an algorithm to derive execution steps is developed.
- An algorithm to execute the adaptation steps is developed, including the possibility for a human operator-in-the-loop to interact with it.

1.5 Structure

The remainder of this thesis is structured as follows. In Section 2 an overview of the tools and mechanisms that will be used in this thesis is given and in Section 3 a survey of the current state of the art is conducted. In Section 4, a running example is introduced, which will be referenced throughout the thesis and is used to evaluate our approach. Section 5 introduces the concept of the approach implemented during this thesis. In Section 6, the details of the implementation of our approach are explained. The evaluation of our approach can be found in Section 7 and Section 8 concludes this thesis with an overview of the limitations of our approach and a survey of future work.

2 Foundations

This section gives an overview of the foundations and tools used during the thesis. We first introduce the concept of component-based architectural models in Sec. 2.1. Because a big part of this thesis consists of the modification and extension of existing tools, this section also gives an overview of them. The *iObserve* tool serves as the basis for this thesis and is introduced in Sec. 2.2. To conclude, the *PerOpteryx* tool for automated architecture optimization is introduced in Sec. 2.3.

2.1 Component-based Architectural Models

During the development and maintenance of a software system, many decisions have to be made that together describe the architecture of a system as defined by Taylor et al. (2009, [36]). We use this definition during the thesis, although there is no commonly agreed upon definition of a software architecture.

Software Architecture (Taylor et al. 2009 [36]) A software system's architecture is the set of principal design decisions made about the system.

As principal design decisions, the authors mention the structure of the system, decisions about functional behavior, non-functional properties or the interaction of components as examples. While this definition of software architecture does not require the documentation of principal design decisions, it is beneficial for the description of an architecture to be documented in a specific artifact. These artifacts may be documented in natural language, UML diagrams or in formal models such as the Palladio Component Model (PCM) [4, 32]. The Palladio Component Model describes architectures in the context of component-based software engineering. A component is described by Szyperski et al. (2002, [35]) as a ünit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.D̈uring this thesis, we employ the PCM to model the architecture of software systems and it's terms to describe the different parts of an application.

The composition and deployment of components is modeled with the PCM by employing different sub-models for specific aspects of the system. The basic sub-model is the *Repository* model, which contains all components the system consists of, as well as their provided and required interfaces. All methods defined in the provided interfaces may also have a *Service Effect Specification (SEFF)* or the extended version, a *Resource Demanding Service Effect Specification (RDSEFF)* associated with them. A SEFF describes the abstract internal behavior of a method and an RDSEFF can be used to additionally associate resource demands with a method. This information can be leveraged for analyzing quality properties of a given model.

The *System* model contains information about the composition of a specific system and uses the components defined in the *Repository*. The *Repository*, however, does not have any information about which system uses it's components and may be shared by many systems. The components are encapsulated in the system by so called *AssemblyContext*s. Another sub-model is the *Resource Environment* model, which provides information about the execution environment and available *Resource Container*s. A *Resource Container* is a server, where components can be deployed and executed and they provide hardware specifications like the processing speed of this server. To describe the allocation of components onto resource containers, the PCM provides the *Allocation* model, which connects *AssemblyContext*s to the *ResourceContainer*s they are deployed on. This connection is done by the *AllocationContext* model elements. Additionally the PCM is able to model the usage of a system by employing it's *Usage* model.

2.2 iObserve

The idea of the iObserve approach [18] is to tackle the problems that arise when adapting and evolving cloud-based, distributed software systems. Such systems are subject to constant changes in their execution environment which are not easy, or not possible, to foresee during their development. Therefore iObserve tackles these challenges by employing a MAPE (Monitor, Analyze, Plan, Execute) control loop to causally connect the running application with an architectural runtime model representation. This representation can then be used either by humans for the evolution of the system or by the planning and execution steps to automatically adapt the application's architecture. Because the iObserve approach considers the adaptation and evolution of software systems to be interwoven and interdependent these as shown in Fig. 2.1, it uses one architectural runtime model for both tasks.

The central model to connect both the design time and the run time model is the runtime architecture correspondence model (RAC). The RAC is shown in Fig. 2.2, which depicts the iObserve megamodel. The megamodel shows the relationships between the different models and artifacts used by iObserve. The models are depicted as boxes and transformations are shown by solid line arrows between the models. The dotted line arrows indicate a conforms to relationship between models and metamodels, and an instance of relationship between gathered data and it's data types.

In iObserve, an application is instrumented with so called monitoring probes. These monitoring probes are generated or created by hand and conform to the Instrumentation Aspect Language (IAL). These monitoring probes generate monitoring records that conform to the Instrumentation Record Language (IRL). These monitoring records are aggregated an refined by iObserve and used to update the architectural runtime model, which represents the current state of the application as an instance of the Architecture Meta-Model.

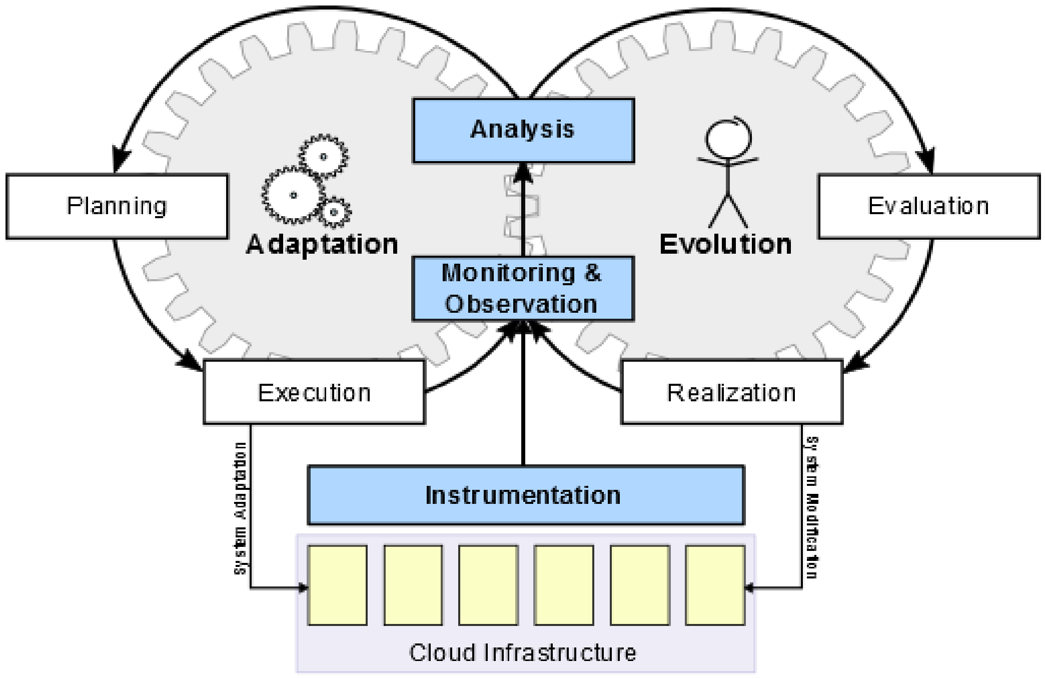


Figure 2.1: iObserve approach to software evolution and adaptation [18]

The iObserve megamodel uses two dimensions to characterize the included models, the design-time vs. run-time and the model vs. implementation level dimensions. For design time, the figure shows the relationship between the monitoring approach used, the architecture model and the corresponding code generation through transformations. The run-time side shows the mapping of monitoring data to their corresponding elements in the RAC which also relate to implementation artifacts.

Currently the iObserve approach only implements the Monitoring and Analyze phase of the MAPE loop. It uses the Palladia Component Model (PCM) [5] as the architecture meta-model to describe both the architectural model and the architectural run-time model.

2.3 PerOpteryx

PerOpteryx [22, 26] is a tool designed to automatically optimize component-based architectural models with respect to performance, reliability and cost of the architecture. It relies on the Palladio Component Model (PCM) to model the architecture of software systems and their costs. It relies on the PCM analysis tools like solvers for Layered Queueing Networks (LQN) [13, 5] or simulators to predict the performance of an architecture model.

It uses an initial architectural model as the starting point for the automatic generation of new architecture candidates, which are generated by employing an evolutionary algorithm. The candidates are then analyzed to get their properties with respect to the given quality criteria and Pareto-optimal [10] candidates are selected to be used during the next iteration or as the output of the process. This process relies on exploring the design space

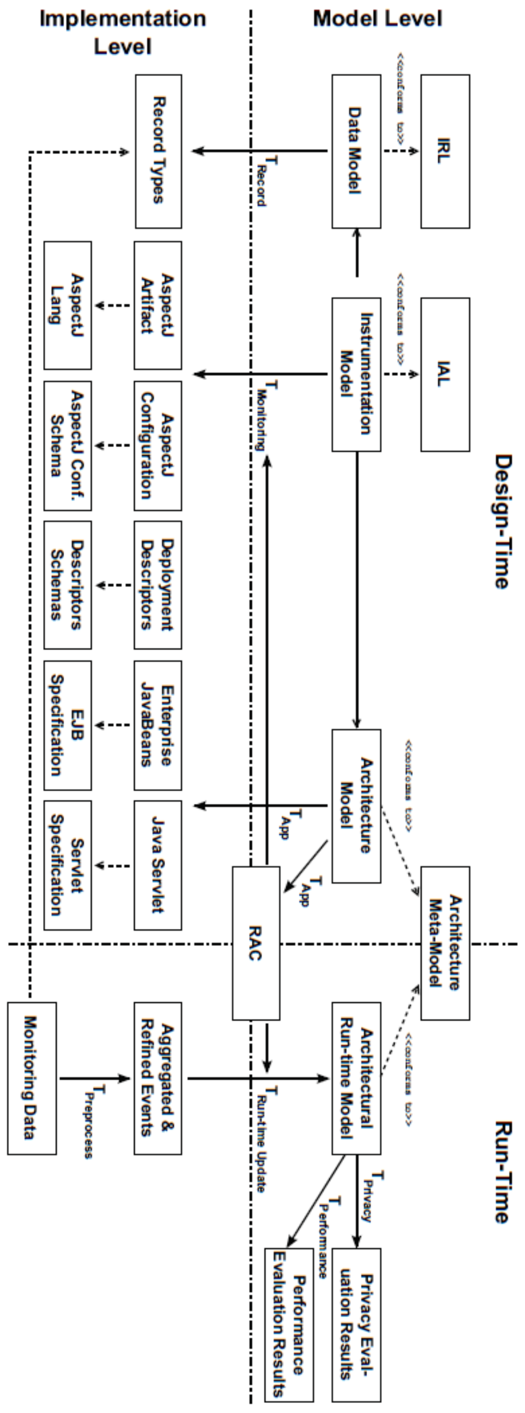


Figure 2.2: The iObserve megamodel in the context of Enterprise Java. [18]

that is spanned by the degrees of freedom of the model. The goal of this exploration is to find a Pareto-optimal architecture for the system that satisfies the given quality criteria as well as optimizes performance, reliability and cost. Because of the multicriterial nature of this problem, a metaheuristic-based approach is used with an evolutionary optimisation algorithm. Originally, this tool was intended to be used by software architects at design time to automatically optimize architectural models.

PerOpteryx defines several models. One model is the *Desing Decision* model, which defines the available degrees of freedom for a specific PCM instance. We only use two of those degrees of freedom for our approach and will introduce them shortly. The first degree of freedom we employ is the *AllocationDegree*, which describes the possible allocations of one *AssemblyContext* and it's encapsulated component onto a set of resource containers. During the optimization, the component is allocated onto one of the possible resource containers according to the heuristics employed by the evolutionary algorithm.

The second degree of freedom that is relevant for our work is the *ResourceContainerReplicationDegree*. This degree of freedom allows the replication of one resource container in a specified range of replications. To this end, the resource container is referenced in the degree and a lower and upper bound for the number of replicas has to be specified. During the optimization, the evolutionary algorithm chooses a number of replicas within that range for a specific candidate and evaluates the candidate.

Another model that is defined by PerOpteryx is the *Cost* model, which is also used during this thesis. This model provides information about the costs of processing resources defined in the *Resource Environment* mode. These costs are used in conjunction with the *QMLDeclarations* model, which describes the quality criteria that have to be optimized. If the costs are defined in the *QMLDeclarations* model as a relevant criterion, the costs of each candidate will be analyzed and taken into account for further processing.

3 State of the Art

Our approach has many similarities with the CloudMF Framework [11, 25]. CloudMF aims to enable the automatic adaptation of cloud-based applications at run-time using CloudML [12] models to model the application. The models are kept in a causal connection with the deployed system by a models@run-time based [6, 30] engine. However, the CloudML modelling language is only used to model the deployment of the application and does not include additional information that may be useful for adaptation. Specifically, the used models for the adaptation planning are decoupled from the models used for the development of the application. This inhibits software evolution processes because the evolved models can not be transferred directly onto the running application and vice versa.

SimuLizar [3] uses performance analysis for self-adaptive systems to enable the analysis of adaptation phases during scaling processes of the system. This approach is based on design-time architectural models and is not designed to leverage run-time information. It is also limited to analyzing the models and therefore provides only limited utility for adaptation planning compared to PerOpteryx.

Metzger et al. [29] introduced FCORE, a model-based approach for the self-adaptation of cloud-based systems. FCORE utilizes run-time feature models and combines them with goal models to derive adaptation actions. All adaptation actions are consolidated to achieve a coordinated adaptation plan. This approach lacks flexibility because it is constrained to the features defined at design time and is difficult to use for systems that do not have many feature sets to choose from.

An approach for the design space exploration of cloud-based applications is provided by Frey et al. [14]. This approach is an evolutionary algorithm specifically designed for cloud-based applications and uses cloud deployment options to define it's search space and the results of it's optimization. It is tightly integrated with CloudMIG [15], an approach for the semi-automatic migration of software to the cloud. This approach focuses on the evolutionary algorithm in a cloud environment, but is not concerned with the automatic adaptation of cloud-based applications.

4 Running Example

As a running example throughout the thesis and for evaluating our approach, we will use the Common Component Modelling Example (CoCoME). CoCoME is a demonstrator for a component based architecture. It is an implementation of the inventory and sales management system of a fictional supermarket chain. There are multiple variants of CoCoME with a cloud variant also available. The cloud variant is available as a PCM based design time model which makes it easy to enrich it with run-time information gathered by iObserve. The application can therefore be used to evaluate the approach developed in this thesis. CoCoME is a typical three-tiered application with a presentation tier, the business logic tier and the data tier. The presentation tier consists of two components. The *PickupShop* component implements a shop system, where a customer can order products online to later pick them up in a selected shop. The *Web* component implements the administrative interface for store and enterprise managers as well as the cash desk interface for cashiers.

CoCoME's business tier consists of the *TradingSystem::CashDeskLine* component, which implements the logic for the cash desks and the *TradingSystem::Inventory* component which is responsible for all other administrative tasks and the booking of sales. The data tier is encapsulated in the *ServiceAdapter* component which is deployed on its own database server. All other components are run either on an enterprise server, responsible for all enterprise related operations, or on a store server. The store server is responsible for store related operations and also holds the The coarse structure of the cloud variant of CoCoME can be seen in Fig. 4.1.

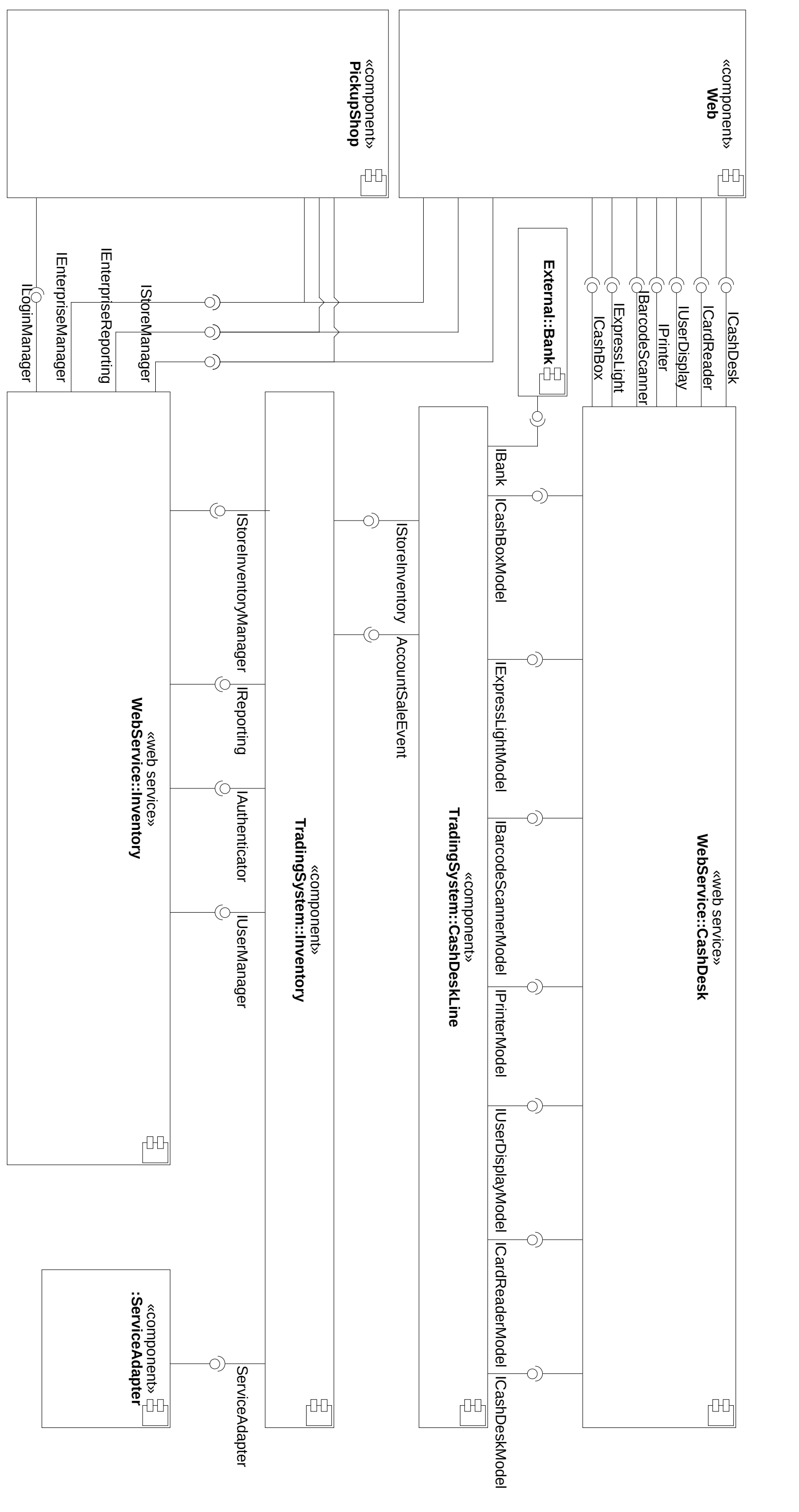


Figure 4.1: Coarse Structure of the Cloud-based Variant of CoCoME [17]

5 Concept

To be able to explore the design space of a particular architecture for performance optimization, it is necessary to first be able to model the application's architecture in a way that accurately represents performance relevant characteristics. In particular, it is necessary to model the deployment and distribution of the application's parts and be able to update the model if a change in the environment occurs. Performance relevant changes in cloud-based applications that happen at runtime are described by Heinrich et al. (2016, [18]) as *workload characterization changes*, *migrations*, *(de-)replications* and *(de-)allocations*. Another requirement is the ability to model the application's deployment in a way that can be used for operator-in-the-loop adaptations. Moreover, the used architectural model has to be usable for analyzing the performance of an architecture.

Developing a framework which covers all these changes and can update an architectural runtime model when such changes appear is outside the scope of this thesis. Therefore we decided to implement our approach as an extension of iObserve, because it is able to provide a component-based architectural runtime model that is causally connected to the application as the basis for the design space exploration and adaptation planning. This enables model based performance analyses and proactive adaptation planning based on models of user behavior which can be used with iObserve. Additionally it facilitates the inclusion of an operator-in-the-loop, because those models are easier to understand for humans than monitoring data. Moreover, it already implements the basic functionalities of the MAPE loop, which is a widely adopted model for feedback loops.

The main idea of our approach is to extend the current version of iObserve and add the planning and execution steps to it's MAPE loop. Currently, iObserve implements the instrumentation and monitoring of cloud-based applications, as well as the processing and integration of those observations into an architectural runtime model. This ensures the causal connection between the application and the runtime model. This runtime model can then be used as the input to the planning and execution steps.

Alternative approaches to causally connect an application with a runtime model like reusing design-time models (e.g. [7, 30] are not suited, because they don't support component-based architectures or do not update the model structure based on runtime information. As pointed out by Heinrich et al. (2016, [18]), the iObserve approach is the only approach which satisfies our requirements.

Because iObserve is based on the PCM, the existing tooling for PCM models can be used to analyze the runtime model for performance bottlenecks. This analysis can then again be used to optimize the architecture and derive a candidate model, as shown in Fig. 5.1. The candidate model is used as the input to the adaptation planning step. In this step, the concrete actions that are necessary to transform the current application's architecture into the target architecture, represented by the candidate model, are planned.

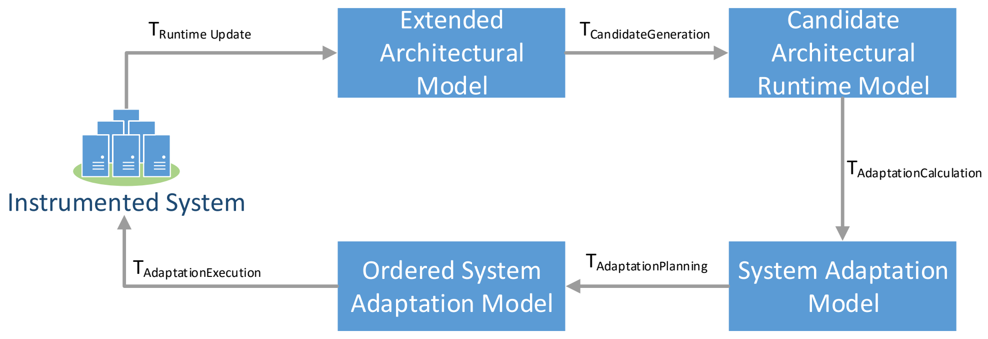


Figure 5.1: Overview of the steps in the iObserve MAPE loop. The arrows stand for model transformations and the boxes represent specific models.

The resulting adaptation plan is then used as input for the execution step. In this step the actions are executed on the application and transform it's architecture into the target architecture. This in turn results in the generation of monitoring events that are again sent to iObserve and integrated into the architectural runtime model.

Because iObserve uses a model-based approach to integrate the architectural models used in development and operations, we use the same approach and extend the iObserve megamodel. The megamodel describes the connections between individual models, metamodels and transformations in the iObserve context. Our extensions to the megamodel are highlighted in Fig. 5.2 and correspond to the stages in the MAPE loop. The models and metamodels are shown as rectangles in the megamodel with the transformations between them depicted as solid line arrows. Diamonds depict multiple models as input or output of a transformation. Dotted lines point out the conformity to a metamodel or an instance of relationship between operational data and their development data type.

In this thesis we implement the $T_{\text{CandidateGeneration}}$ transformation, which uses three models as it's input. The first model is the *Extended Architectural Runtime Model*, generated by iObserve, which represents the current state of the application. This model conforms to the *Extended Architecture Meta Model* which extends the current meta model by adding the elements needed to represent a cloud-based application. The *Cloud Profile Model* is a part of the *Extended Architecture Meta Model* and is used to describe the available cloud resources, which have to be considered during candidate generation. The *Cloud Profile Model* is therefore used in conjunction with the *Degree of Freedom Model* to model the degrees of freedom. The degrees of freedom span the design space within which the optimization is performed.

During the candidate generation, multiple candidates are being generated and each one is evaluated for it's performance properties through the $T_{\text{Performance}}$ transformation. Once a suitable candidate is found, the $T_{\text{AdaptationCalculation}}$ transformation calculates the differences between the current architectural runtime model and the target model generated by the candidate generation. These differences are then rearranged by the $T_{\text{AdaptationPlanning}}$ transformation. This transformation is necessary to ensure that the adaptation actions are ordered correctly.

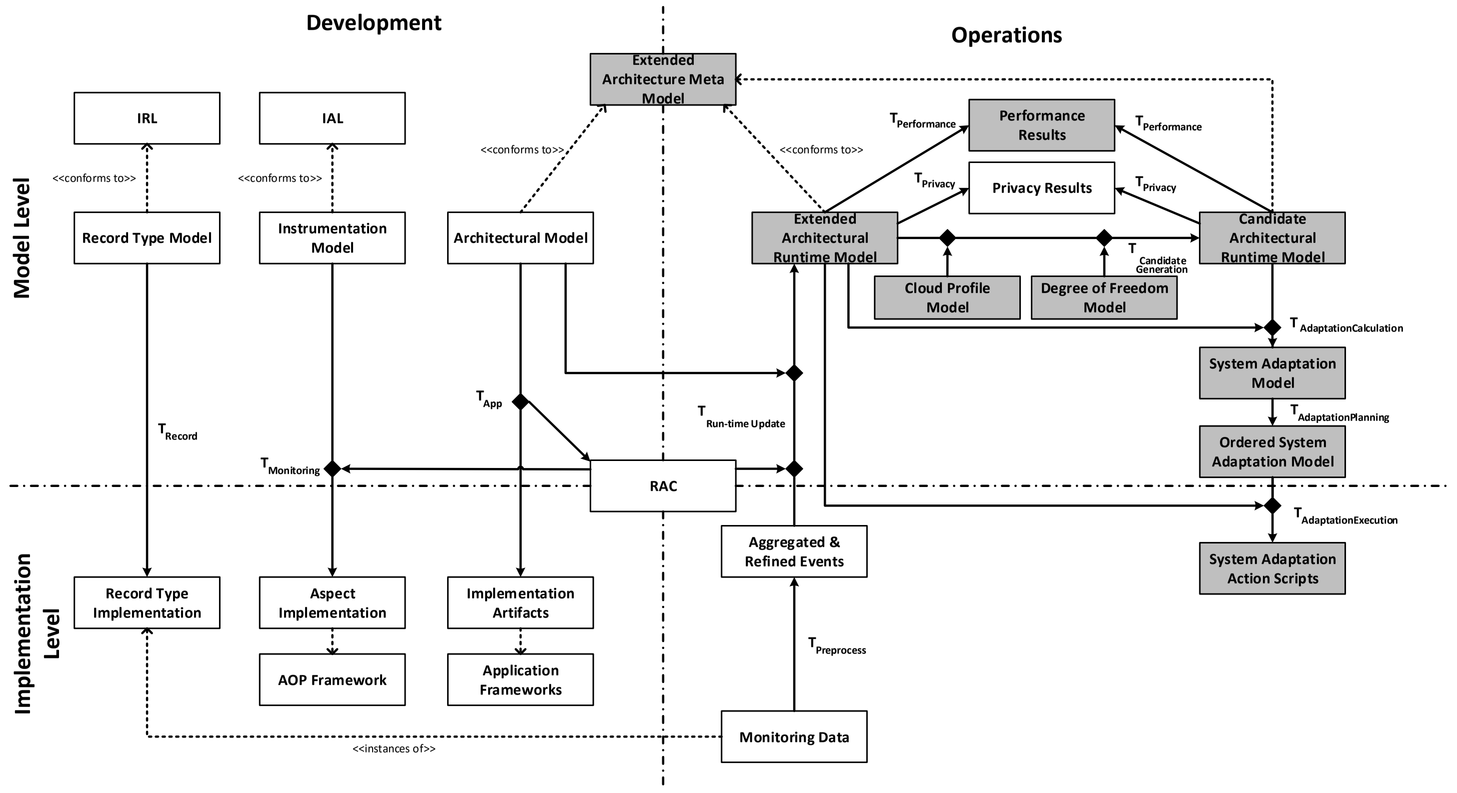


Figure 5.2: Overview of the extensions to the iObserve megamodel. The added or extended elements are highlighted. (Based on Heinrich et al. 2017 [20])

Once the adaptation actions are available in the correct order, the $T_{\text{AdaptationExecution}}$ transformation generates parameterized action scripts and executes them in the given order. If any actions are found that can not be executed automatically, the operator is involved during this step to manually perform the necessary actions.

As a general guideline to our decision processes, which are detailed in the following sections, we chose to employ approaches that are as independent of the used technologies and applications as possible. This improves the portability of our approach and ensures, that even when exchanging some components, the overall approach does not need to change.

In Section 5.1 we first describe the details of the candidate generation and the adaptation calculation is detailed in Section 5.2. The adaptation planning is explained in Section 5.3 and we conclude this chapter by describing the adaptation execution in Section 5.4.

5.1 Candidate Generation

The generation of candidates through design space exploration is at the heart of our approach and is further divided into two steps, *Base Model Extraction* and *Model Optimization* that are executed sequentially. This separation is necessary, because the input model needs to be preprocessed before the design space exploration can be executed, as detailed in Sec. 5.1.2. We introduce the *PCM Cloud Metamodel* as an extension to the PCM architectural meta model in Section 5.1.1, which is used in the *Base Model Extraction* step described in 5.1.2. The central step is the candidate generation and optimization which is described in Section 5.1.3. During this step, new candidates are generated and analyzed for their performance and cost properties. If a candidate with better performance and costs than the current one is found, it is used as input to the *Adaptation Calculation* step, which is described in Section 5.2.

5.1.1 Extended Architectural Meta Model for the Cloud

In this section we propose an extension of the PCM as the architectural meta model used in iObserve. The architectural meta model that is needed for our approach has to provide information about the used types of virtual machines (VMs) and their hardware specifications. This is necessary because the candidate generation has to generate candidates that use valid VM instance types, otherwise the candidate can not be deployed correctly or the performance analyses on the model become invalid. This is because in a cloud context, most cloud providers only offer specific hardware configurations that can not be changed by the consumers.

In addition, the meta model has to provide information on how to access an already deployed VM, for example to (de-)allocate or migrate deployed components. Without this information, it would not be possible to adapt the running system without human interaction during the execution step. This also includes another requirement, the ability to provide information on how to access the cloud provider for acquiring and terminating VMs. These requirements are inspired by the CloudML meta model [11]. Another important information is the pricing of the available instance types to be able to consider the costs of a deployment while analyzing candidate models. This information therefore has to be present to assess the quality of a candidate architecture.

There are several possibilities to satisfy these requirements. One is to use a different meta model that is specifically designed for the purpose of handling cloud-based deployments, for example the one proposed in [12]. This solution has the benefit of being specifically tailored to the needs of a cloud-based application, but is also restricted to this one purpose. Moreover, this approach loses the benefits of an already matured tooling ecosystem, like the one PCM provides with it's tools for performance analysis and model optimization.

PerOpteryx also already provides a cost model for the optimization. This model can however only be used for the cost aspect and it associates costs with single processing resources. For our approach this could be used, but it would be tedious to maintain the cost model for every used processing resource, when there is only a limited number of VM types available. Therefore, to increase the maintainability, we propose to associate the costs with a specific VM type.

The PCM, the architectural meta-model used in iObserve, was developed with an on-premise background in mind and does not natively satisfy all the requirements for cloud-based applications. Thus, we propose an extension to the PCM that includes the information needed to model a cloud-based application. The proposed *PCM Cloud Metamodel* is shown in Figure 5.3. Through this approach we can leverage the existing tooling for the PCM while still being able to provide a description of a cloud-based application that includes the information needed to plan and execute adaptations of it.

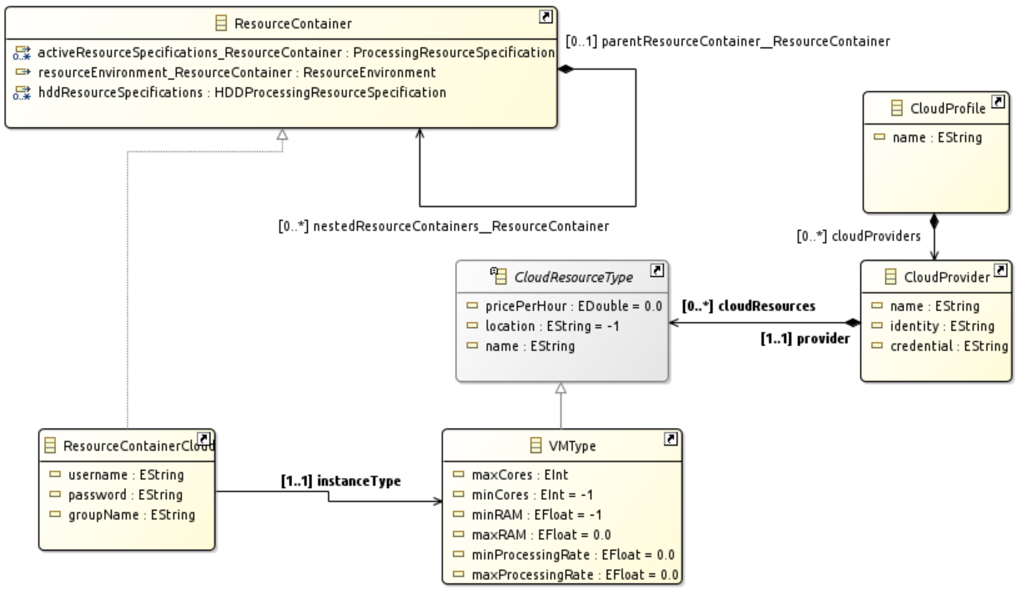


Figure 5.3: The PCM Cloud Metamodel extension to the PCM. The *ResourceContainer* metaclass is a PCM entity.

We extend the PCM in a non-invasive way, as described by Strittmatter et al. (2015) [34], and define the new *ResourceContainerCloud* as a subclass of the *ResourceContainer* type in the PCM. The original *ResourceContainer* of the PCM is extended to include the *username* and *password* attributes, which describe the login credentials that should be used when logging into the resource container to perform an adaptation task. This is used to access the resource container for performing (de-)allocation or migration actions and can be provided to the operator in case an adaptation action can not be performed automatically.

Moreover, we assign each container the attribute *groupName* which reflects the name of the allocation group this container belongs to. The concept of allocation groups is detailed in Section 5.1.2. A container's *groupName* is used to perform adaptation actions on a group of resource containers on which the same components are allocated. This is useful to be able to execute an adaptation action for a complete group in parallel, therefore decreasing the time it takes for the adaptation to complete.

Additionally, each container is assigned an *instanceType* which connects the *Resource-ContainerCloud* to the newly introduced *Cloud Profile* metamodel. The *instanceType* represents a specific type of virtual machine a cloud provider offers to it's customers. This type is represented by the *VMType* metaclass, which provides information about the hardware specifications of a virtual machine type as they are specified by the cloud provider, which is used in the optimization process to predict the performance of a candidate. It also includes information about the location of the virtual machine, as well as pricing information, through its superclass *CloudResourceType*.

The location of the virtual machine and it's type are used needed for acquiring a new VM and are therefore necessary for the execution of this action. The pricing information is, as mentioned before, used to determine the costs of a candidate architecture during optimization. A *CloudResourceType* represents a specific cloud resource offered by a cloud provider and references the *CloudProvider*. This superclass was introduced to facilitate adding new cloud resources that are not virtual machines, like automatically scaling computing resources.

The *CloudProvider* metaclass represents a cloud provider and stores the credentials to access the cloud provider's services, which is needed during the execution of an adaptation to acquire VMs. The name is used to identify the provider, which also identifies how to access the provider's interfaces for resource management during adaptation execution. This mapping is done at the implementation level and to keep the model clean of implementation details, only the name is used. This enables the implementation to add or remove new providers without changing the metamodel. A *CloudProvider* can also contain an arbitrary number of *CloudResourceType* instances, which can be acquired to deploy application components on them. The cloud providers are again contained in the *CloudProfile* metaclass, which eases access to all cloud providers during the planning stage. This is convenient, because the planning stage has to consider all available resource types from all cloud providers, when generating new candidates.

With this extension of the PCM metamodel we can model the environment in which a cloud-based application is executing and also store the necessary information for automated adaptation execution. Moreover, we retain the possibility to leverage the tools developed for the PCM metamodel.

5.1.2 Base Model Extraction

The extended PCM metamodel is used as input for the model processing step. This step is performed for two reasons. The first is that the design space has to be defined before the design space exploration can happen in the *Model Optimization* step. The design space for a cloud-based application is spanned primarily by the available cloud resources and the components that can be allocated on them. The available cloud resources are a discrete set of possible choices, because the available resource containers are normally defined by the cloud provider as a set of VM types. However, the amount of replicas of those types is practically not limited. This poses the problem that the available possibilities for allocating components is practically unlimited as well, because every component may be allocated on any of the replicated resource containers.

This leads to the second reason, why the model is preprocessed before the actual design space exploration. Without constraining the possible allocations of components, the design space to explore would also be unconstrained, which makes the optimization more difficult. One way to constrain the possibilities for allocating a component is to use the introduced types of resource containers and allow the allocation of components only onto types of resource containers instead of individual ones. By replicating the instances of those resource container types, it is then possible to leverage the elasticity of the cloud without the problem of an unlimited number of possible allocations. This strategy leads to an application architecture where one component is deployed onto a group of resource containers of the same type. Another advantage of this constraint is that it limits the number of degrees of freedom for the optimization. We can also limit the number of needed resource containers to the number of resource container types, which further decreases the design space.

The disadvantage is that it is no longer possible to allocate components arbitrarily onto any kind of resource container. However, this kind of arbitrary allocation only complicates the planning process and does not necessarily lead to better performance. The resource demands of a component do not change based on the resource container it is deployed on. Therefore, if a resource container of a specific type fits the resource demands of one component and satisfies the performance constraints, it should be deployed on that specific type. If this component causes a performance bottleneck, the available options are to replicate it and distribute the load or to migrate it onto a resource container with better hardware. Replicating it onto a resource container of a different type than the ones already used would result in a different utilization of the replicated resource container, which may not completely solve the performance bottleneck. Migrating only some components onto a different type of resource container leads to the same problem. Therefore, it is not necessary from a performance perspective to allow arbitrary allocations.

To be able to reflect this kind of constraint, it is necessary to be able to easily determine the number of replications of an allocated component and the involved resource containers as well as their type. This information is available in the PCM, but it is not easily accessible. Therefore we introduce the concept of *Allocation Group*s. *Allocation Group*s are calculated during the *Base Model Extraction* step and represent all allocations of one component onto one specific type of resource container.

Alternatively, the PCM could have been changed to support a basic architecture model which only contains one resource container per allocated component. Additionally, the replicated containers would be modeled in a different architecture model, including an easily accessible method to calculate the number of replicas per resource container in the basic model. This would, however, be a major change in the architecture of the PCM which is outside the scope of this thesis. The advantage of this solution is that it would explicitly model the replications and groups of containers belonging to one component.

The calculation of *Allocation Groups* is done as a preprocessing step and therefore only once per optimization run. This relieves our approach from calculating the number of replications for each component on-the-fly, which would be costly. The algorithm for this calculation is shown in Algorithm 1. For the calculation, we use the allocation group matrix consisting of sets of resource containers that are addressed via the allocated component and the type of the resource containers. The algorithm goes through all allocation contexts in the model and extracts the deployed component from the assembly context which is referenced in the allocation context. It then extracts the resource container that is referenced in the allocation context and queries it's type. As the next step, it adds the resource container to the set in the allocation group matrix addressed by the extracted component and resource container type.

Algorithm 1 Algorithm for calculating allocation groups.

```
  Let $\mathbf{R}$ be the set of all resource containers
  Let $\mathbf{T}$ be the set of all resource container types
  Let $\mathbf{C}$ be the set of all components
  Let $\mathbf{G} \in \mathcal{P}(\mathbf{R})^{\mathbf{C} \times \mathbf{T}}$ be the matrix of allocation groups

Require: $A$, the set of all allocation contexts
  function CalculateAllocationGroups(A)
      for all $a \in A$ do
          $asmCtx \leftarrow a$.assemblyContext
          $component \leftarrow asmCtx$.encapsulatedComponent
          $container \leftarrow a$.resourceContainer
          $containerType \leftarrow container$.instanceType

          $\mathbf{G}_{component,containerType} \leftarrow \mathbf{G}_{component,containerType} \cup \{container\}$
      end for
  end function
```

For example, let there be three instances of the component *TradingSystem:Inventory* which are deployed onto three resource containers of type *AWS m3.medium*, as depicted in Fig. 5.4. During the calculation of all *Allocation Group*s, the three resource containers would be grouped together in the allocation group for the *TradingSystem:Inventory* component on the *AWS m3.medium* resource container type. Another instance of the *TradingSystem:Inventory* component is allocated on a resource container of type *AWS m3.xlarge*, together with an instance of the *Web* component. This results in two different *Allocation Group*s, one for each component.

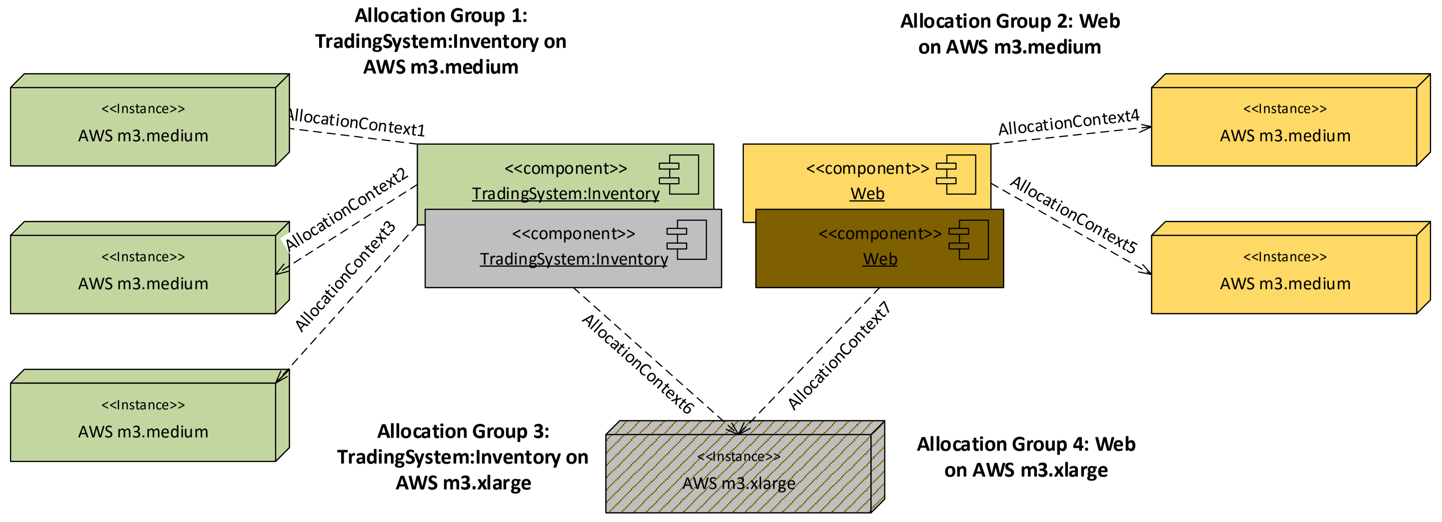

Allocation Group 1: TradingSystem:Inventory on AWS m3.medium
Allocation Group 2: Web on AWS m3.medium
<<Instance>> AWS m3.medium
<<Instance>> AWS m3.medium
<<Instance>> AWS m3.medium
AllocationContext1
AllocationContext2
AllocationContext3
<<component>> TradingSystem:Inventory
<<component>> TradingSystem:Inventory
<<component>> Web
<<component>> Web
AllocationContext4
AllocationContext5
<<Instance>> AWS m3.medium
<<Instance>> AWS m3.medium
AllocationContext6
AllocationContext7
Allocation Group 3: TradingSystem:Inventory on AWS m3.xlarge
<<Instance>> AWS m3.xlarge
Allocation Group 4: Web on AWS m3.xlarge

Figure 5.4: Example for the grouping of components by the types of resource containers they are allocated on.

Furthermore, this grouping also allows to easily determine how many times a component was replicated on a specific container type by simply counting the resource containers in the *Allocation Group*. This is important for determining the degrees of freedom that span the design space.

For this thesis, we employ two degrees of freedom defined by PerOpteryx [26]. The first one is the *Allocation Degree*, which enables the allocation of components onto any of the allocation groups that were calculated from the current runtime model. For the basic case that a component gets allocated onto a different allocation group with the same number of resource containers, this reflects the migration runtime change described in Heinrich et al. [18]. If the number of resource containers in the target allocation group is smaller than the one in the original allocation group, this results in migration and deallocation changes. The migration is executed for the number of resource containers in the new allocation group and the remaining components are deallocated from the resource containers in the old allocation group. In case the number of resource containers in the target allocation group is higher, all original components have to be migrated and the component is newly allocated on all remaining resource containers.

The replication and dereplication changes occur if a component is still allocated onto the same allocation group and the number of containers in this allocation group changes. To model this, we employ the second degree of freedom, which is the *Resource Container Replication Degree*. This degree describes the possibility of a resource container to be replicated. Because we use the concept of allocation groups, it is possible to describe the number of replicas in one allocation group by just using one resource container as a representative of the whole group and enable the replication of this resource container. This way we can reduce the number of needed resource containers and thereby also the number of degrees of freedom.

However, if we want to be able to co-allocate multiple allocation groups onto the same set of resource containers, this has to be taken into account when defining the *Allocation Degree*. In this case, the allocation degree must provide the possibility to not only allocate the component onto all possible resource container types, but also onto all container types of any other allocation group.

If for example, the *TradingSystem:Inventory* component from Fig. 5.4 needs the ability to be co-allocated onto the two *Web* component instances running on the *AWS m3.medium* container type, the allocation context needs to include one set of all available container types where only the inventory is deployed and one set of container types where both components can be allocated. This means, in general, we need to include one complete set of resource container types per allocation group into the allocation degree to allow co-allocation of all components with all other allocation groups.

The PCM does not support any concept like allocation groups. One possibility would be to include this concept into the PCM, which would require a redesign of parts of the PCM. Therefore we propose an approach to transform an instance of the extended PCM into another instance of the extended PCM which is a simplified version of the original model. A resource container in the transformed model is the representative container for one allocation group with one representative allocation context for a specific allocation group deploying one assembly context to the container. This way we create a basic model without replicated resource containers, allocation contexts or assembly contexts. This transformed model is then used as the input to the actual design space exploration.

This transformation results in a loss of information about the actually used resource containers. It is therefore no longer possible to take these resource containers into account when optimizing the deployment. This loss of information would not occur with the concept of a base model in the PCM. This is, however, compensated by the adaptation planning, which does not rely on the transformed base model, but uses the original runtime model to extract the adaptation actions. Therefore, allocation groups are a good compromise.

The transformation itself is done by first deriving the allocation groups from the current runtime model. Then the resource environment model, the cost model and the allocation model are emptied to prevent any unwanted elements to interfere in the transformed model. Now these models are rebuilt, using the information from the generated allocation groups. First, the transformation rebuilds the resource environment. To do so, one resource container for each *VMType* defined in the *Cloud Profile* model is added for each allocation group. At the same time the needed information about the costs of the resource container is added to the cost model and a *Resource Container Replication Degree* for the container is created. The runtime of this approach is in $O(n \cdot m)$, with n the number of resource container types and m the number of allocation groups.

One way to improve this procedure would be to extend PerOpteryx with a way of handling the replication of resource container types. This would require PerOpteryx to use the extended PCM and use the VM types as a type of resource container which can always be replicated. This would be facilitated by also introducing the concept of a basic and a replication model into PerOpteryx. The drawback of this approach is that this would mean to fundamentally change the way PerOpteryx handles replications, which can not be done in this thesis.

Another possible alternative would be to use a different way of generating the candidates, which is able to natively handle replications of resource containers. CDOXplorer [14] is an example for such an approach and also uses evolutionary search algorithms for design space exploration. This approach employs it’s own architectural model and descriptions of deployment options. Therefore, to use it, we would also need to transform the PCM into the input models and also transform the output again. To do this would therefore mean to introduce even more complex transformations than the one we propose to use, without additional benefits.

To create a *Resource Container Replication Degree*, it is necessary to specify the lower and upper bounds for the number of replications. The lower bound is always one, because PerOpteryx relies on the fact that there is always at least one replica available. However, this behavior does not have an influence on the results, because the costs are only applied for resource containers that are actually used and the same applies for the performance analysis. To compute the upper bound for the number of replicas, let $\mathbf{r} \in \mathbb{N}$ be the current number of replicas in an allocation group, then the upper bound $\mathbf{u} \in \mathbb{N}$ is calculated as

$$\mathbf{u} = (\mathbf{r} + \mathbf{replicationOffset}) * \mathbf{replicationFactor}$$

where $\mathbf{replicationOffset} \in \mathbb{N}$ and $\mathbf{replicationFactor} \in \mathbb{N}$ are configurable and are set to $\mathbf{replicationOffset} = 10$ and $\mathbf{replicationFactor} = 1$ by default. This calculation enables the replication to acquire at least a number of **replicationOffset** replications of a resource container, where 10 should be a sensible default value for moderate sized applications, given that replications are possible for each individual allocation group.

5.1.3 Model Optimization

The model optimization step derives new candidate architectures, which optimize the performance and costs of the application's deployment. For this step, we require an approach which is able to do a multi-criterial optimization on the input architectural runtime model. This is necessary, because we want to optimize the model not only targeting the performance properties, but also minimizing the costs of the deployment at the same time. Moreover, it has to be able to handle multiple types of virtual machines, because cloud providers often offer a multitude of virtual machine types. Therefore it should not be constrained to just one or a few types. It should also be able to provide pareto-optimal candidates in a sufficiently fast way. The optimization of deployment options is known to be an NP-hard problem [7], but it should be possible to complete within an acceptable timeframe. Another requirement is the use of the PCM to avoid unnecessary transformations of models.

Such optimizations can be tackled in various ways [22]. One way is to scalarize all objectives but one, thus converting the problem into a single-objective optimization problem which is then solved for systematically varied scalarizations. Examples for this approach are the weighted sum method or the ε-constraint method. They may deliver fast results if the sub-problems can be solved in a fast way. Because architectural optimization can be arbitrarily complex, this is not the case. Therefore, to solve those problems, it is still necessary to rely on numerical methods or simulations, except for models with very strong constraints.

Therefore, to optimize a multi-criterial architecture optimization problem, the use of multi-objective heuristics is an approach that is better suited [22]. There are two types of multi-objective heuristics, trajectory-based and population-based. The trajectory-based heuristics start with a, possibly random, point in the search-space and search for a more optimal solution based on this point and according to the method employed by the heuristic. Examples are hill climbing methods or simulated annealing. These methods only use one point in the search-space and assess it's properties, which can be sub-optimal for multi-criterial problems. For those problems it tends to work better to look at multiple points in the search-space, because the solution is most likely a set of pareto-optimal points.

The population-based approaches use a set of solution points and work with those solutions to generate new solutions in each iteration. Therefore they are well-suited to handle multi-criterial optimization problems. The most important examples of these approaches are evolutionary algorithms, ant colony optimization and particle swarm optimization. Because evolutionary algorithms are a popular and efficient method for architecture optimization, we chose to implement the design space exploration for our approach on the basis of this method.

Our approach achieves the model optimization through design space exploration by using the PerOpteryx tool which was previously described in Section 2.3. There are several reasons why we chose this tool. PerOpteryx is very well integrated into the PCM ecosystem and is capable of using different methods to execute the design space exploration spanned by the degrees of freedom. The most important method is it's constrained evolutionary algorithm which is capable of generating multiple architecture model candidates by using the initial model, varying it's values for the degrees of freedom and crossing it with other candidates. Additionally it is still able to guide the model creation by employing additional heuristics, like server consolidation. Through this evolutionary algorithm, it is also capable of producing good candidates in a fast way.

However, PerOpteryx does not support cloud-based applications out-of-the-box, which results in the necessity to pre-process the input model, as described in Sec. 5.1.2. Yet this can be done with relatively low effort, because PerOpteryx uses the PCM, therefore it does not require additional model transformations as it would be the case when using other tools, like CDOXplorer [14]. By using the pre-processing step it is also possible to use the resource container types specified in the extended PCM with PerOpteryx.

The candidate generation uses a multi-criterial optimization approach and tries to improve on all of the criteria that are specified by a Quality-of-Service Modeling Language (QML) model [16]. This QML is used in PerOpteryx to constrain the evolutionary algorithm to only consider viable options that fulfill the given quality-of-service aspects.

For this thesis, the main goals of the design space exploration are to decrease the response time of the application while minimizing the deployment costs at the same time. But it is also possible to add additional constraints, like the consideration of privacy constraints, to the optimization process if needed. PerOpteryx analyses each candidate for the given quality attributes and ranks them according to the results of this analysis. For the performance analysis, PerOpteryx can use the tools the PCM provides.

These performance analysis tools can be divided into two groups. One group uses a simulation-based approach to analyze the performance properties of a model. This approach simulates the modeled system and derives the performance properties from it's observed behavior. *SimuCom* is the most mature performance analyzer for the PCM based on this approach. It is the reference simulator for the PCM and generates an executable Java representation of the model to simulate it's behavior. *EventSim* [28] for event-based simulation and *QPNSolver* [23] for the simulation of queuing petri nets are other examples of this approach. All of these approaches have in common that they provide good results, although they are rather time and resource intensive because of the simulation they have to perform. This is especially a problem when analyzing systems with many servers, which can occur frequently when exploring deployment options with an evolutionary algorithm. Moreover, these approaches do not support server replicas, which is essential when analyzing cloud-based applications.

The second group consists of analytical approaches to analyze the model. The most prominent representative of this group is the *LQN Solver* [13], which uses a numerical approach to predict the performance of layered queuing network (LQN) models. This tool is specifically designed for distributed systems and supports server replications. However, the PCM model needs to be transformed into an LQN model before it can be analyzed. Another analytical approach is the solver for Stochastic Regular Expressions (SREs) [23], which can be used to calculate the distribution of response times in a single-user setting. These approaches are inherently faster than simulations, yet they may be not as accurate.

However, this is negligible in our case, because with an evolutionary algorithm it is more important to have a fast method to analyze the population, because of the possibly big number of candidates. Moreover, the transformation from the PCM to LQN models is already available in PCM. Therefore we decided to use the *LQN Solver* for the performance analysis of the candidates generated by PerOpteryx. Moreover, using the *LQN Solver*, it is not necessary to explicitly model the load balancing mechanisms between replicas because the solver automatically distributes the load.

PerOpteryx may potentially generate many candidates as the output of the design space exploration process. All of these candidates are pareto-optimal, and therefore equally viable solutions to the optimization problem. To further process the candidate and go on with the adaptation of the system, it is however necessary to choose one of those candidates. One way to do this would be to always ask the operator, which candidate to choose. However, this would hinder the automatic adaptation and if no operator is available, the adaptation would be stalled. Moreover, the evolutionary algorithm has a tendency to distribute all components onto as many servers as possible, which is intensified further by our approach to create a resource container for every available *VMType*.

Therefore we adapted PerOpteryx to output the candidate with the lowest costs, which tends to be the one with the least amount of resource containers in use. This is the case, because each used resource container contributes to the costs of a deployment. Therefore, having fewer resource containers normally results in lower costs as well. We also adapted PerOpteryx to output the selected candidate as a PCM model to use it for the planning of the adaptation steps. Normally PerOpteryx uses it's own output model containing the choices for all given degrees of freedom. However, using the PCM instead of the PerOpteryx specific output has the advantage of being a more general solution. This means it is possible to exchange the candidate generator without modifying the rest of the implementation, as long as the generator uses the PCM as it's output model. The best candidate that was generated by the candidate generator is then used to calculate the necessary adaptation steps. Note that PerOpteryx always regards the input candidate as one possible output candidate. Therefore, if no other candidate can be found by the optimization, the input model will be regarded as the best candidate.

5.2 Adaptation Calculation

In this section we detail the derivation of the necessary adaptation actions to adapt the current application architecture into the optimized architecture. This is done on the basis of the difference between the prescriptive architectural runtime model that was generated during the preceding candidate generation step and the descriptive architectural runtime model of the current application architecture.

An alternative approach to using the model differences would be to directly use the output of PerOpteryx. This output is given in the form of a *DesignDecision* model, which contains the choices that were made for each degree of freedom of a specific candidate. This model can also be used to compute the differences between the current architecture and the candidate architecture. However, the main drawback is that this model is specific to PerOpteryx and therefore not a part of the PCM. For our approach, we do not want to depend on the details of the specific tool used to derive candidates. This leads to a slightly more complex solution, because the model comparison has to extract all the choices from the model, but allows the candidate generation tool to be exchanged without rewriting the comparison.

Another option is the direct observation of the decisions that are made by the candidate generation tool. These decisions could in turn be mapped to adaptation actions. This approach requires some form of observation mechanism included in the candidate generation tool, which is again highly specific to the used tool. Moreover, for PerOpteryx this could lead to a non-optimal sequence of adaptation actions, because the evolutionary algorithm uses random adaptations to generate candidates. Minimizing the number of adaptation actions is important, because every adaptation action can potentially disrupt the functionality of the application. Additionally, it does not make sense to execute an adaptation action, just to undo it in a subsequent action, which is a possibility when using this approach. Therefore, this approach would need a significant amount of post-processing of the resulting adaptation actions, which makes it infeasible for our approach.

We chose the model comparison, even though the difference calculation is a little more complex than the comparison when directly using the *DesignDecision* model. It provides a feasible way to calculate the minimal sequence of adaptation actions and is independent of the used tool. The actions that are derived by our approach are modeled in the *Systemadaptation Model*, which is described in the following Section.

5.2.1 Systemadaptation Model

One possibility to represent adaptation actions is to directly implement them in the used programming language. This approach does not reflect the model-driven approach of iObserve and the PCM by introducing a direct dependency to language-specific artifacts. To be in accordance with the overall approach of iObserve and the PCM, we therefore decided to introduce a new model which represents all available adaptation actions and contains all information needed to execute them. This has the benefit of being technology independent with the disadvantage that such a model introduces an additional overhead for it's creation. Moreover, the technology-dependent approach has to access the information from the PCM Cloud model to access the cloud provider's interfaces. With an additional model directly referencing the PCM Cloud model, we can minimize the effort to acquire all the needed information.

The introduced *Systemadaptation Model* is shown in Fig. 5.5 and contains the *SystemAdaptation* metaclass as the container for all *Action* instances contained in the model. We define a hierarchy for *Action*s, by dividing them into two subgroups: *AssemblyContextActions* and *ResourceContainerAction*s. This reflects the natural hierarchy of these actions. *AssemblyContextAction*s are used for all adaptations that include an *AssemblyContext* and therefore an application component. *ResourceContainerAction*s are used for all actions that include a *ResourceContainer*.

The *AssemblyContextAction*s include the *AllocateAction*, the *DeallocateAction*, the *MigrateAction* and the *ChangeRepositoryComponentAction*. Therefore these actions cover the (de-)allocation and migration cases of changes at runtime [18]. The additional *ChangeRepositoryComponentAction* adds the ability to exchange a component with a different

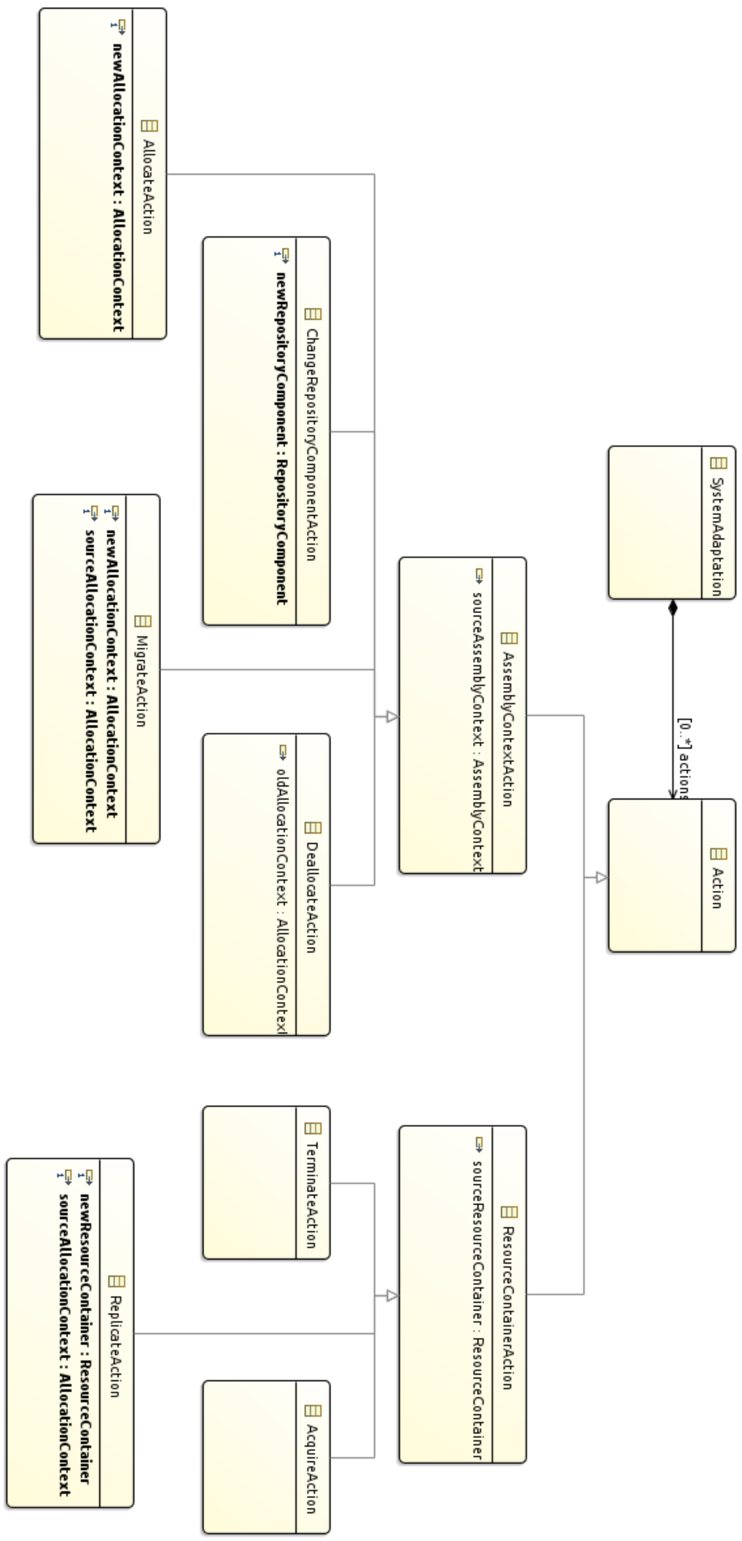

SystemAdaptation
[0..*] actions
Action
AssemblyContextAction
sourceAssemblyContext : AssemblyContext
ResourceContainerAction
sourceResourceContainer : ResourceContainer
AllocateAction
newAllocationContext : AllocationContext
ChangeRepositoryComponentAction
newRepositoryComponent : RepositoryComponent
MigrateAction
newAllocationContext : AllocationContext
sourceAllocationContext : AllocationContext
DeallocateAction
oldAllocationContext : AllocationContext
TerminateAction
ReplicateAction
newResourceContainer : ResourceContainer
sourceAllocationContext : AllocationContext
AcquireAction

Figure 5.5: The Systemadaptation model containing all available adaptation actions.

component which provides and requires the same interfaces. This might make sense if the implementation of a specific component is fast for some use-cases but not for others. Therefore it would be beneficial for the performance of the application to exchange this component if a use-case requires a more constantly performing implementation of the component.

All *AssemblyContextAction* instances have the *sourceAssemblyContext* reference to the *AssemblyContext* instance for which this action is relevant. The *AllocateAction* uses the *newAllocationContext* reference to describe onto which *ResourceContainer* instance the component should be allocated. For the *DeallocateAction*, the *oldAllocationContext* reference describes which allocation of the source component is no longer valid and has to be deallocated.

A *MigrateAction* references the *sourceAllocationContext* to indicate where the component was originally allocated to and the *newAllocationContext* to describe onto which container the component should be migrated. For the *ChangeRepositoryComponentAction* it is sufficient to reference the *newRepositoryComponent* that should be used instead of the component that is encapsulated in the *sourceAssemblyContext*.

The *ResourceContainerAction*s can be divided further into the *TerminateAction* which terminates an instance of a resource container, the *AcquireAction* which acquires a new resource container and the *ReplicateAction* which replicates a component onto a new resource container, including the state of the component. This covers the replication and dereplication case of possible changes at runtime. All *ResourceContainerAction*s have a reference to their *sourceResourceContainer* which is the *ResourceContainer* instance for which this action should be carried out. For the *TerminateAction* and the *AcquireAction* this information is already sufficient to describe the action. Only the *ReplicateAction* needs more information, because it needs to know which component should be replicated and onto which resource container. Therefore, *newResourceContainer* references the resource container onto which the component included in the *sourceAllocationContext* should be replicated.

The *ReplicateAction* and a corresponding de-replication action are, however, special cases which are mapped to an *AcquireAction* and an *AllocateAction* for replication. Dereplication is mapped to a *DeallocateAction* and a *TerminateAction*. This is due to the fact that it is currently not possible to copy the internal state of a component.

With the information provided by these actions, all necessary information for the execution of the actions is provided. Most of the actions have a direct or indirect, via *AllocationContext*, reference to a resource container of the PCM Cloud model which again references the information needed to access the cloud provider and issue the necessary commands to execute the actions.

5.2.2 Model Difference Calculation

To calculate the differences between the descriptive architectural runtime model and the prescriptive architectural runtime model and to derive the adaptation actions, we use a simple algorithm based on the fact that all relevant actions are either related to an *AssemblyContext* or to a *ResourceContainer*.

To calculate the *AssemblyContextAction*s, we save all *AssemblyContext*s of the current model into a dictionary, which is indexed by the context's ID. Next, we iterate over all *AssemblyContext*s present in the target model. If the current *AssemblyContext* is not present in the dictionary, we add a new *AllocationAction* to the list of actions, because the assembly context needs to be allocated in the target model. If the current context is present in the dictionary, there are only two possible actions left. In case the component in the current context has a different ID than the component in the context from the dictionary, a *ChangeRepositoryComponentAction* is added to the list of actions, because there was an exchange of components on this assembly context. If the current context is deployed on a different resource container than the context in the dictionary, we need to add a *MigrateAction* to the list of actions. Next, we delete the context from the dictionary and go on to the next context.

Once this iteration is done, all assembly contexts that remained in the dictionary are no longer used in the target model and we create a *DeallocationAction* for each context left in the dictionary. This way, all *AssemblyContextAction*s are recognized by the algorithm.

We employ a similar algorithm to determine all *ResourceContainerAction*s. First all resource containers of the current model which have components allocated on them are added to a dictionary, again indexed by their IDs. Then we iterate over all resource containers in the target model which have components allocated on them. If the current resource container is not contained in the dictionary, a new *AcquireAction* for this server is generated, because the container needs to be present in the target model. If the container is found in the dictionary, it can be deleted from the dictionary, because the container is still present in the target model. Lastly, we iterate over all containers that are still present in the dictionary and add *TerminateAction*s for each one of them, because they are no longer used in the target model and can therefore be terminated.

These algorithms depend on the fact that the IDs of PCM entities like the resource containers and assembly contexts are not altered when deriving new candidates. An alternative approach to the model comparison would be to use a general purpose model comparison tool like the Eclipse Modelling Framework (EMF) Compare tool. Such a tool does not rely on PCM specific identifiers, however, it would require additional post-processing of the comparison results to extract the adaptation actions, which would result in a similar algorithm as the one described. Additionally it would introduce more complexity to the problem and enlarge the resource consumption, because the tool would need to be loaded and executed in addition to iObserve. Therefore we decided to use an algorithmic solution specifically tailored to the problem at hand.

5.3 Adaptation Planning

The adaptation actions resulting from the adaptation calculation step can not be executed as they are discovered because the order of their execution is important. It is not possible to allocate a component onto a resource container that has not yet been acquired, for example. Therefore the adaptation planning step is needed to order the adaptation actions in such a way, that dependencies between actions are taken into account. Furthermore, the disruptions caused for the application should be minimized. Therefore, it is the goal of the adaptation planning to keep the application in a functional state while executing the adaptation actions.

One option to order the actions is to use the different types of actions to determine their ordering. This approach first executes all actions to acquire resource containers. This way, all subsequent actions are guaranteed to be executed in an environment where all resource containers are already available. Now the order for the change component, migration, replication as well as allocation and deallocation actions has to be determined.

The deallocation and termination actions should be performed last, because deallocating components before their replacements are allocated may lead to a loss of functionality until the adaptation is completed. The same holds true for termination actions. Allocate, migrate and replicate actions are independent of each other and may be performed in any order after the resource containers are available. Component change actions, however, may have an influence on migrate and replicate actions, because a changed component can also be migrated or replicated. Deallocation actions are not impacted by a component change, because it would not make sense to change a component and then deallocate it. Therefore component change actions have to be executed before migrate and replicate actions. Component changes do not affect allocation actions, because an allocation action would just allocate the changed component instead of allocating the current component and changing it afterwards.

After all components are allocated, migrated and replicated, the deallocation and termination actions can be performed to remove the components that are no longer needed and terminate superfluous resource containers. The proposed adaptation sequence is therefore to first execute all acquire actions, followed by allocation actions, component change actions, migration actions and replication actions. As the last two action types, all deallocation actions followed by all terminate actions should be executed. This approach is a simple way to order the adaptation actions only depending on the type of action that are to be performed. It does, however, assume that there are no further dependencies between the components and does not take into account possible changes of other artifacts, like configuration files, build-scripts or test cases.

This might be problematic, if, for example, the *TradingSystem:CashDeskLine* component needs to specify the location of the *TradingSystem:Inventory* component in one of it's configuration files to access the store's inventory. If the inventory component needs to be migrated, it is necessary to change the configuration files of all *TradingSystem:CashDeskLine* components. With the proposed approach, this change has to happen inside the executed action script and is not considered when planning the adaptation.

Therefore a different approach to adaptation planning which takes into account the impact of a change on the complete application architecture is an alternative to the method described above. One approach which is capable of calculating such impacts is the Karlsruhe Architectural Maintainability Prediction (KAMP) tool introduced by Rostami et al. [33]. With KAMP, it is possible to calculate the impact of a change request and to predict the steps that are necessary to implement that change. It uses additional annotations on the architectural model to represent the needed information about dependencies between components and the artifacts associated with them. This approach is also useful to direct an operator in a more detailed way if an adaptation step can not be performed manually. The main focus of KAMP, however, is on software evolution activities in a semiautomated context. Therefore the tool would need modification to be able to use it for automatic adaptation planning. It also increases the complexity of the used architectural model and would need to be integrated into iObserve before using it.

We decided to use the first approach because it does not depend on additional tooling and is sufficient to plan an adaptation sequence. It does, however, rely on the action scripts to consider all dependencies between modules and execute the necessary actions to update them.

5.4 Adaptation Execution

The action sequence generated by the planning step has to be executed for the actual transformation of the application's current architecture into the target architecture. This execution step is highly dependent on the technology that is used to build and deploy the application. Our approach should, however, not depend on a specific technology stack or on application specific build infrastructure. To achieve this kind of solution, some kind of abstraction from those details has to be used instead of directly implementing the actions in iObserve.

One way to achieve this is to add an additional wrapper script between the application specific action script and the adaptation action model. This script can be called by our approach and only executes the application specific script on the resource container it should run on. The application specific action scripts can be specified for each component and for each type of action and perform their respective action. For this approach, the action scripts are only executed on the resource containers, therefore restricting them to be self-contained and be able to retrieve all required artifacts by themselves. This is a valid assumption for most applications, because most software engineers use repositories and version control systems to build and develop their code. In this case, an action script can easily use these repositories to access the relevant artifacts on the resource container. With the use of container technologies, this process can even be simplified further. The drawback of this solution is that all action scripts are executed on the resource containers, which may take some time to complete and blocks the resource container from being used.

Another possibility to achieve this is to use a dedicated separate tool which receives the adaptation plan as input and uses this information to directly execute the adaptation actions on the application. With this approach, application specific technologies can be leveraged and there is room to implement actions that have an influence on multiple resource containers instead of only one. However, this approach requires the implementation of such a tool for every application that wants to use iObserve. Moreover, it requires additional maintenance in case the used technologies change. Therefore we propose the use of wrapper scripts for executing adaptation actions from iObserve.

To execute the action scripts on the cloud provider's infrastructure, it is necessary to have a way to communicate with the cloud provider as well as with the resource containers. One way to achieve this is to implement the communication directly in iObserve. The advantage of this solution is, that the number of new dependencies is kept to a minimum. A big disadvantage, however, is that there are many cloud providers without a common way to access them and the interfaces may change relatively frequently. Efforts to standardize interfaces, like the Open Cloud Computing Interface (OCCI, [9]) or the Cloud Infrastructure Management Interface (CIMI, [8]) exist, but are not yet adopted by many providers. Therefore it would require a lot of effort to support the needed cloud providers with this approach.

Therefore we propose the use of a middleware, which implements the communication with the cloud providers and defines a uniform API for the wrapper scripts to work with. This has the disadvantage of introducing new dependencies, but enables the use of many cloud providers with only one code base. By using this approach the code for the wrapper scripts does not need to be changed when the cloud provider interfaces change, only the middleware needs to be changed.

6 Implementation

We implemented our approach using Java in conjunction with the Eclipse Modeling Framework (EMF) for all modeling related tasks. This approach is used by iObserve as well, so our approach builds on the existing iObserve implementation, which is also available under an open-source license [1]. The candidate generation steps are implemented in the *org.iobserve.planning* packages within iObserve and the adaptation planning as well as the execution is implemented in the *org.iobserve.adaptation* packages.

iObserve uses the Teetime framework to implement it's pipes and filters architecture. Therefore our approach integrates into this framework as well. The candidate generation stage is implemented in the *CandidateGeneration* class, which itself calls the sub-stages implemented in the *ModelProcessing*, *ModelOptimization* and *CandidateProcessing* classes. The extraction of the base model is performed by the *ModelTransformer* class. This class extracts the allocation groups from the resource environment by initializing an instance of the *AllocationGroupsContainer*, and consecutively rebuilding the model. This process, including the following exectuion of PerOpteryx via an instance of the *ModelOptimization* class is shown in Fig. 6.1.

The *ModelOptimization* class builds a wrapper class around the execution of PerOpteryx. This is necessary because PerOpteryx is usable only in conjunction with Eclipse. Therefore, to use PerOpteryx in the iObserve context we developed a standalone version of PerOpteryx, which can be called from iObserve to perform the design space exploration and candidate generation. This is done by packaging PerOpteryx as an Eclipse Application and adding an additional wrapper for configuring PerOpteryx. A native integration of PerOpteryx into iObserve would be beneficial, because the candidate generation could be observed better. But due to the dependencies to Eclipse this was not possible to achieve during the course of this thesis.

The PerOpteryx Eclipse Application is available with iObserve in the form of an Eclipse Plug-in Project called *peropteryx.plugin*. In this plug-in, we automatically configure PerOpteryx according to the given configuration options. The most important being the number of iterations of it's evolutionary algorithm and the number of individuals per iteration. This configuration is done in the *PerOpteryxLaunchConfigurationBuilder* class, which is called from the application's entry point in the *PerOpteryxHeadless* class.

The *CandidateProcessing* class pre-processes the candidate and generates a graph representation from it. This is included in our implementation, because there is simultaneous work on privacy constraints, which uses this representation. Additionally, this stage sets some variables in the *AdaptationData* object, which is passed on to the adaptation stage and includes information about the location of the models.

The adaptation stage is implemented in the *SystemAdaptation* class and is also divided into sub-stages. These sub-stages are implemented in the *AdaptationCalculation*, *AdaptationPlanning* and the *AdaptationExecution* classes. The *AdaptationCalculation* calculates

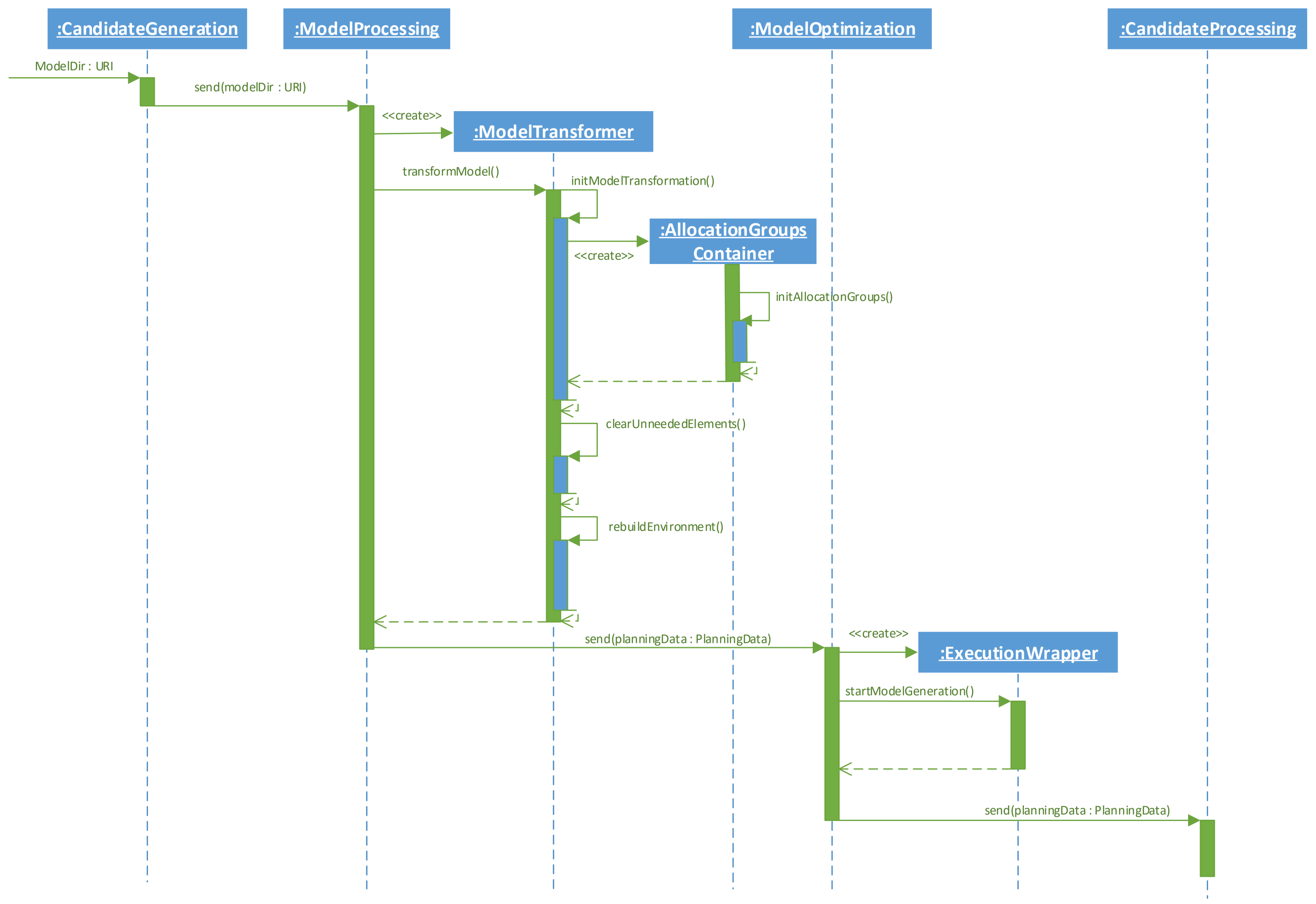


Figure 6.1: The sequence diagram showing the transformation of the model and the execution of the design space exploration.

the adaptation actions resulting from the difference between the original model and the generated candidate model. These actions are represented by the model elements of the *Systemadaptation* model. This model and the generated Java representation is available in the *planning.systemadaptation* Eclipse Plug-in Project located in iObserve.

These model elements are ordered in instance of the *AdaptationPlanning* class before they are passed on to the *AdaptationExecution*. There the adaptation actions are used to retrieve parameterized action scripts from an *ActionScriptFactory* instance. This approach is used to decouple the adaptation execution from the actually used scripts. The adaptation execution does not know the specifics of the scripts generated by the factory. Therefore the scripts are easier to exchange.

For executing the scripts, they are first queried whether they can, in principle, be executed in a fully automatic way or if they may need operator interaction. If a script can not be executed automatically, the operator is asked if she wants to execute the adaptation anyway through an *IAdaptationEventListener* instance. This listener is set from outside and may access a command-line interface, a web interface or in any other way communicate with the operator. If the operator decides to continue, all adaptation actions that can be performed automatically will be executed and in case of a failure or of an action which can not be executed automatically, the operator is prompted to perform the operation.

The adaptation actions are executed on the cloud provider's infrastructure through a middleware. For Java, there are only a few candidates as a middleware with Apache jclouds being the most mature. It supports more than 40 cloud providers at the time of writing. Therefore we chose to implement our approach on the basis of jclouds. However our architecture allows an easy exchange of the middleware by adapting the execution scripts. Once the adaptation is complete, the application is in the target state and the MAPE loop begins again.

7 Evaluation

To be able to answer the research questions posed in Sec. 1.3, we evaluate our approach in this section. The evaluation is divided into four main parts. The first part describes the general evaluation design and is found in Sec. 7.1. The second part in Sec. 7.2 evaluates the accuracy and the scalability of the base model transformation. In Sec. 7.3, the adaptation calculation and planning are evaluated. Finally, in Sec. 7.4, the adaptation execution is evaluated.

7.1 Evaluation Design

As the basis for evaluating our approach, we use the research questions presented in Section 1.3 to derive an evaluation design. The evaluation therefore focuses on the accuracy and scalability of the adaptation planning and execution stages. All experiments are conducted on a virtual machine with 4 virtual CPUs, 22 GB of RAM running a Linux operating system. This setup was chosen, because our approach would probably be deployed in a cloud environment as well and run on a virtual machine with similar hardware. The host system is an Intel Core i7-6820HQ CPU with 8 logical cores, 2.7 GHz and 32 GB RAM.

As an example for virtual machine instances for the accuracy evaluations, we use two instance types, *m3.large* and *m4.2xlarge*, from Amazon's Elastic Compute Cloud (EC2) in the presented scenarios. We use this cloud provider as an example because of it's widespread use and the adoption of it's APIs by other cloud providers. The two instance types were chosen because of their hardware and costs. At the time of writing, the *m3.large* type has a dual core processor with 2.6 GHz processing rate at costs of 0.15$ per hour. This type represents a resource container with low costs and low performance. The *m4.2xlarge* has an octa core processor with 2.4 GHz processing rate at costs of 0.48$ per hour. It represents a resource container with high costs and high performance. We chose only two types, because it is sufficient to use two distinct types as a minimal example for multiple types.

During the evaluation, we do not evaluate the optimization results of PerOpteryx and suppose that it always finds a pareto-optimal candidate. Therefore, the evaluation is divided into three parts. This separation is necessary, because it would be difficult to separate the contributions of each step due to the interdependencies between them and the optimization from PerOpteryx in between. The evaluation is structured according to the order in which the steps are performed in the iObserve pipeline. It is sufficient to evaluate each part independently, because iObserve only adds the connection of the steps which has no influence on the accuracy and only introduces a small constant offset on the time for the scalability evaluation. Therefore, we evaluate the accuracy and scalability of each step in our approach separately. Furthermore, this enables us to pinpoint existing problems more accurately to the specific steps where they occur. The first part in Sec. 7.2 evaluates the transformation of the input model into the base model that is handed over to PerOpteryx for the design space exploration. An evaluation of PerOpteryx is given in [26]. The second part in Sec. 7.3 evaluates the planning of the adaptation steps based on the optimized model. In Sec. 7.4 the execution of the planned steps is evaluated as the last part.

7.2 Base Model Extraction

The base model transformation is the preprocessing step an input model has to go through before it can be used to optimize the deployment. To answer the research questions, we evaluate this step of the extended iObserve pipeline according to it's accuracy and it's scalability. In Sec. 7.2.1, the accuracy is evaluated and in Sec. 7.2.2, we evaluate the scalability of this step.

7.2.1 Accuracy

The accuracy of the input model transformation into the base model is an important aspect of our approach and is needed to answer the research question **RQ-1.1**. The overall accuracy of our approach can be guaranteed if each individual step is performed accurately. For our approach it is therefore sufficient to show, that the model which is used as an input for PerOpteryx is reflecting the current runtime model and accurately describes the available degrees of freedom, costs and computing resources of each resource container. Therefore we want to show that the model transformation introduced in Sec. 5.1.2 is capable of doing this.

Because the transformation focuses on grouping replicated allocations and the respective resource containers, we focus our evaluation on cases where replications are absent or present. These are the only cases where our algorithm steps in. Therefore, we evaluate the accuracy of the transformation by using the following scenarios, which are loosely based on the changes at runtime mentioned in Heinrich et al. (2016, [18]). For each scenario, we describe the current model and the expected output model after the transformation.

As a metric to evaluate the accuracy of the transformation, we use the Jaccard coefficient as a measure of the similarity between the models. The Jaccard coefficient is an easy to calculate similarity metric for unordered sets and therefore a good choice for this task. Other metrics like Kendall's rank correlation coefficient [21] or Spearman's rank correlation coefficient are not a good fit, because there is no need to compare the order of the model elements. The PCM resource environment, allocation and system model have no concept of an order for their elements and we only look at these models for the accuracy.

To use the Jaccard coefficient, we consider the model elements of the PCM resource environment, allocation and system model as elements in a set. For assessing the equality of two model elements, we compare all attributes of the elements, except their names and IDs, because these attributes can differ for newly generated elements. The Jaccard coefficient is defined as follows, with **A** and **B** being the sets that are to be compared.

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Scenario 1: No Replications

This scenario addresses the case when no component in the input model is replicated, but some components are deployed onto resource containers of the same type. It demonstrates the ability to recognize that there are no replicated components. Additionally, it demonstrates that unused resource container types are added to the model so that the components can be allocated to any resource container type during the subsequent design space exploration.

The supermarket chain just recently decided to move their business information system CoCoME into the cloud and used just one virtual machine per top-level component to test the feasibility. Now the chain wants to increase the load on the application and starts to optimize the deployment. This change reflects the runtime change of an increasing workload.

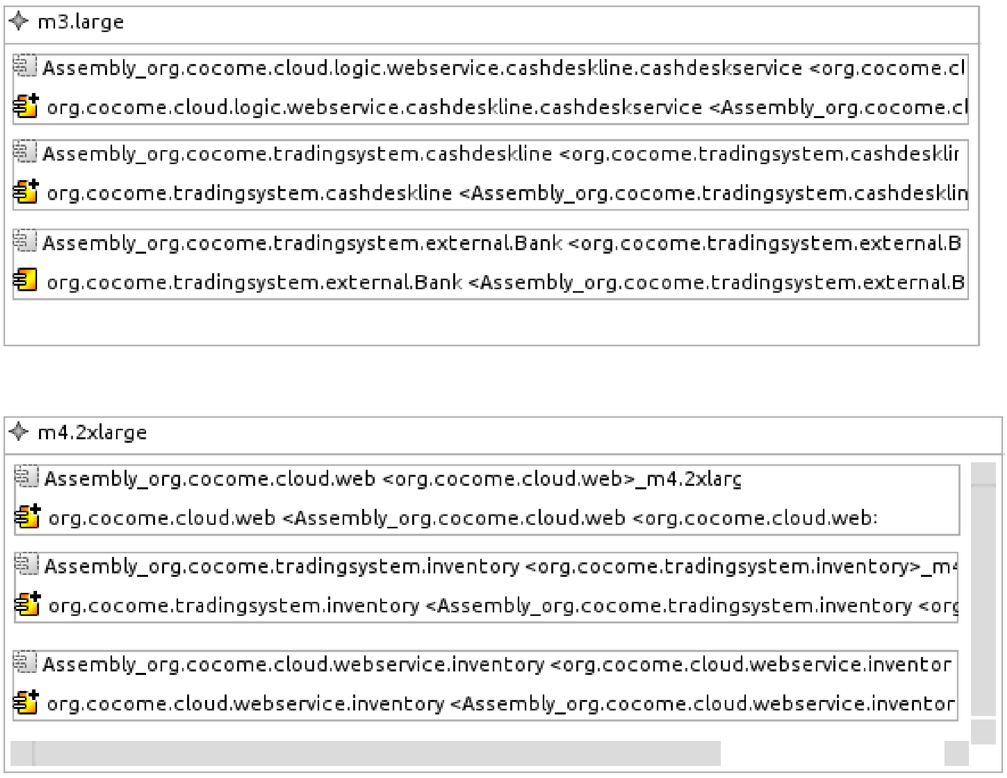
Initial model: The components *TradingSystem:Inventory*, which includes the *ServiceAdapter* component, *WebService:Inventory* and *Web* are each allocated onto an instance of the resource container type *m4.2xlarge*. The *WebService:CashDesk*, *TradingSystem: CashDeskLine* and *External:Bank* components are allocated onto an instance of the resource container type *m3.large*. The corresponding allocation model is depicted in Fig. 7.1.

Expected transformed model: All components are still allocated onto their respective resource containers as in the current model. For each component a new resource container is created. This container uses a different instance type than the one onto which the component is currently allocated. For example, if the component is currently allocated onto a resource contaiern of type *m3.large*, the newly created resource container will have the type *m4.2xlarge*. Additionally, for each newly created container, the processing resource specifications and costs are set in the resource environment and cost model as defined by it's instance type. For each resource container in the resource environment there exists a new *ResourceContainerReplicationDegree* to allow it's replication during the optimization phase. This degree is needed to span the design space for PerOpteryx.

The model resulting from the transformation has a Jaccard coefficient of 1.0, when compared to the expected model, including the generated degrees of freedom, costs and resource specifications. Therefore, all model elements are equal with respect to this metric and the transformation is accurate for this scenario.

Scenario 2: Simple Replicated Components

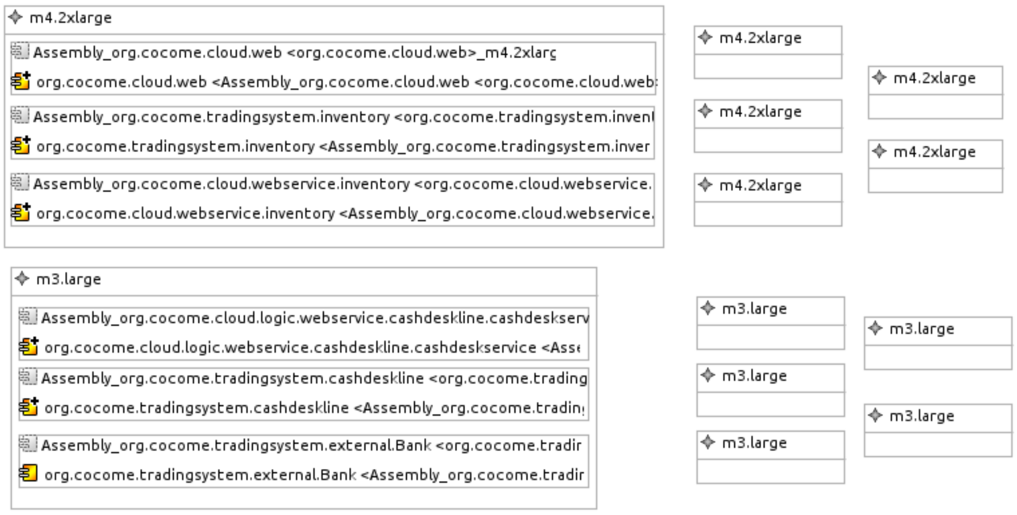
This scenario addresses the case where some components in the input model are replicated and some components are co-allocated with other components. It demonstrates the reduction of all replicated allocations of one component to it's base case with only one allocation on one resource container of the correct type. Moreover, it demonstrates the ability to distinguish between resource container types that are instantiated multiple times but with different allocated components and between types that are used to replicate a component.



```
m3.large
Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <org.cocome.cl
org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <Assembly_org.cocome.cl
Assembly_org.cocome.tradingsystem.cashdeskline <org.cocome.tradingsystem.cashdesklir
org.cocome.tradingsystem.cashdeskline <Assembly_org.cocome.tradingsystem.cashdesklin
Assembly_org.cocome.tradingsystem.external.Bank <org.cocome.tradingsystem.external.B
org.cocome.tradingsystem.external.Bank <Assembly_org.cocome.tradingsystem.external.B

m4.2xlarge
Assembly_org.cocome.cloud.web <org.cocome.cloud.web>_m4.2xlarc
org.cocome.cloud.web <Assembly_org.cocome.cloud.web <org.cocome.cloud.web:
Assembly_org.cocome.tradingsystem.inventory <org.cocome.tradingsystem.inventory>_m
org.cocome.tradingsystem.inventory <Assembly_org.cocome.tradingsystem.inventory <orc
Assembly_org.cocome.cloud.webservice.inventory <org.cocome.cloud.webservice.inventor
org.cocome.cloud.webservice.inventory <Assembly_org.cocome.cloud.webservice.inventor
```

Figure 7.1: The initial allocation model of scenario 1.



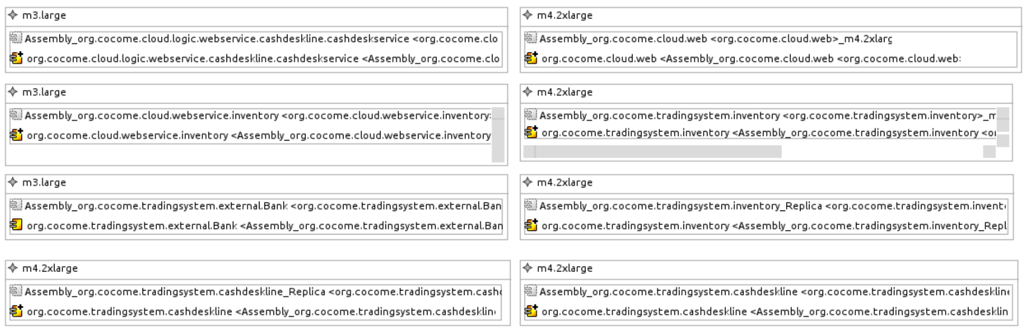
```
m4.2xlarge
Assembly_org.cocome.cloud.web <org.cocome.cloud.web>_m4.2xlarc
org.cocome.cloud.web <Assembly_org.cocome.cloud.web <org.cocome.cloud.web:
Assembly_org.cocome.tradingsystem.inventory <org.cocome.tradingsystem.invent
org.cocome.tradingsystem.inventory <Assembly_org.cocome.tradingsystem.inver
Assembly_org.cocome.cloud.webservice.inventory <org.cocome.cloud.webservice.
org.cocome.cloud.webservice.inventory <Assembly_org.cocome.cloud.webservice.

m4.2xlarge
m4.2xlarge
m4.2xlarge
m4.2xlarge
m4.2xlarge

m3.large
Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cashdeskserv
org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <Asse
Assembly_org.cocome.tradingsystem.cashdeskline <org.cocome.trading
org.cocome.tradingsystem.cashdeskline <Assembly_org.cocome.tradin
Assembly_org.cocome.tradingsystem.external.Bank <org.cocome.tradir
org.cocome.tradingsystem.external.Bank <Assembly_org.cocome.tradir

m3.large
m3.large
m3.large
m3.large
m3.large
```

Figure 7.2: The resulting allocation model of scenario 1 after the transformation. The names of the resource containers have been shortened for better readability.


m3.large
Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <org.cocome.clo
org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <Assembly_org.cocome.clo

m4.2xlarge
Assembly_org.cocome.cloud.web <org.cocome.cloud.web>_m4.2xlarg
org.cocome.cloud.web <Assembly_org.cocome.cloud.web <org.cocome.cloud.web:

m3.large
Assembly_org.cocome.cloud.webservice.inventory <org.cocome.cloud.webservice.inventory:
org.cocome.cloud.webservice.inventory <Assembly_org.cocome.cloud.webservice.inventory

m4.2xlarge
Assembly_org.cocome.tradingsystem.inventory <org.cocome.tradingsystem.inventory>_m
org.cocome.tradingsystem.inventory <Assembly_org.cocome.tradingsystem.inventory <oi

m3.large
Assembly_org.cocome.tradingsystem.external.Bank <org.cocome.tradingsystem.external.Ban
org.cocome.tradingsystem.external.Bank <Assembly_org.cocome.tradingsystem.external.Ban

m4.2xlarge
Assembly_org.cocome.tradingsystem.inventory_Replica <org.cocome.tradingsystem.invent
org.cocome.tradingsystem.inventory <Assembly_org.cocome.tradingsystem.inventory_Repl

m4.2xlarge
Assembly_org.cocome.tradingsystem.cashdeskline_Replica <org.cocome.tradingsystem.cash
org.cocome.tradingsystem.cashdeskline <Assembly_org.cocome.tradingsystem.cashdeskline

m4.2xlarge
Assembly_org.cocome.tradingsystem.cashdeskline <org.cocome.tradingsystem.cashdeskline
org.cocome.tradingsystem.cashdeskline <Assembly_org.cocome.tradingsystem.cashdesklin

Figure 7.3: The initial allocation model of scenario 2.

The last adaptation resulted in the replication of the *TradingSystem:Inventory* and the *TradingSystem:CashDeskLine* components due to an increased load, which was caused by a big sales campaign. Now the sales campaign is over and the optimization is to be executed again to find a more cost-efficient deployment with the decreased load.

Current model: The components *TradingSystem:Inventory* and *TradingSystem:CashDeskLine* are each allocated onto two instances of the resource container type *m4.2xlarge* and the *Web* component is allocated onto one instance of the same type. All other components are allocated onto one instance of the resource container type *m3.large*. The corresponding allocation model is depicted in Fig. 7.3.

Expected transformed model: Each of the components *TradingSystem:Inventory*, *TradingSystem:CashDeskLine* and *Web* is allocated to one resource container of the type *m4.2xlarge* and there is one resource container of the type *m3.large* with no allocations on it for each of those components. All other components are each allocated on one resource container of the type *m3.large*, with an additional unallocated resource container of the type *m4.2xlarge* present. For each component there is a new *AllocationDegree* which allows the allocation of the component onto every resource container in the resource environment. Additionally, for each newly created container, the processing resource specifications and costs are set in the resource environment and cost model as defined by it's instance type. For each resource container in the resource environment there exists a new *ResourceContainerReplicationDegree* to allow it's replication during the optimization phase. This degree is needed to span the design space for PerOpteryx.

The model resulting from the transformation has a Jaccard coefficient of 1.0, when compared to the expected model, including the generated degrees of freedom, costs and resource specifications. Therefore, all model elements are equal with respect to this metric and the transformation is accurate for this scenario.

Scenario 3: Co-allocated And Replicated Components

This scenario addresses the case where all components are allocated and replicated onto the same resource container type. It demonstrates the ability to correctly recognize co-allocated components even when using the same resource container type.

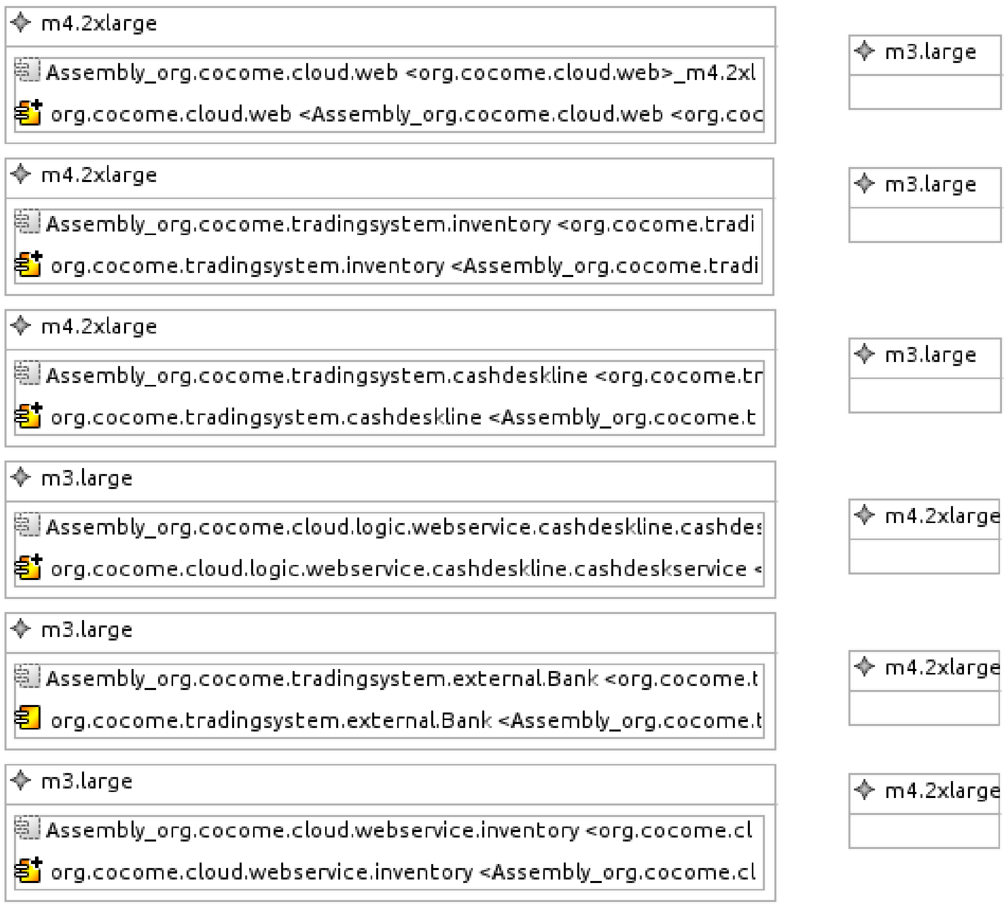
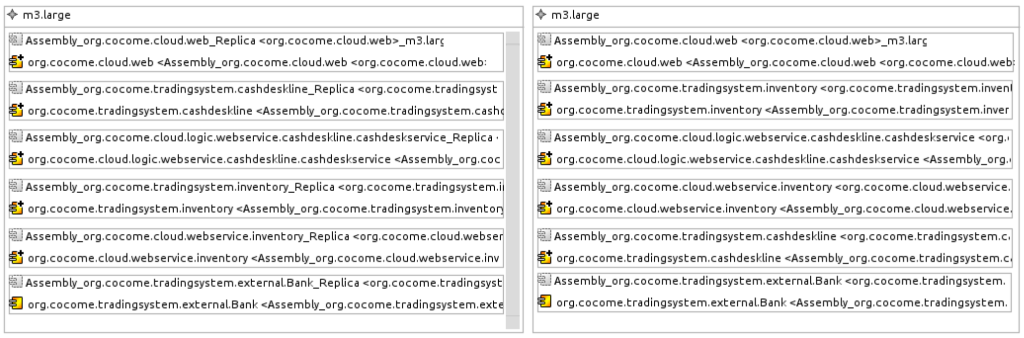

m4.2xlarge
Assembly_org.cocome.cloud.web <org.cocome.cloud.web>_m4.2xl
org.cocome.cloud.web <Assembly_org.cocome.cloud.web <org.coc
m4.2xlarge
Assembly_org.cocome.tradingsystem.inventory <org.cocome.tradi
org.cocome.tradingsystem.inventory <Assembly_org.cocome.tradi
m4.2xlarge
Assembly_org.cocome.tradingsystem.cashdeskline <org.cocome.tr
org.cocome.tradingsystem.cashdeskline <Assembly_org.cocome.t
m3.large
Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cashdes
org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <
m3.large
Assembly_org.cocome.tradingsystem.external.Bank <org.cocome.t
org.cocome.tradingsystem.external.Bank <Assembly_org.cocome.t
m3.large
Assembly_org.cocome.cloud.webservice.inventory <org.cocome.cl
org.cocome.cloud.webservice.inventory <Assembly_org.cocome.cl
m3.large
m3.large
m3.large
m4.2xlarge
m4.2xlarge
m4.2xlarge

Figure 7.4: The resulting allocation model of scenario 2 after the transformation. The names of the resource containers have been shortened for better readability.



m3.large

Assembly_org.cocome.cloud.web_Replica <org.cocome.cloud.web>_m3.larg
org.cocome.cloud.web <Assembly_org.cocome.cloud.web <org.cocome.cloud.web:

Assembly_org.cocome.tradingsystem.cashdeskline_Replica <org.cocome.tradingsyst
org.cocome.tradingsystem.cashdeskline <Assembly_org.cocome.tradingsystem.cashd

Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice_Replica ·
org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <Assembly_org.coc

Assembly_org.cocome.tradingsystem.inventory_Replica <org.cocome.tradingsystem.i
org.cocome.tradingsystem.inventory <Assembly_org.cocome.tradingsystem.inventory

Assembly_org.cocome.cloud.webservice.inventory_Replica <org.cocome.cloud.webser
org.cocome.cloud.webservice.inventory <Assembly_org.cocome.cloud.webservice.inv

Assembly_org.cocome.tradingsystem.external.Bank_Replica <org.cocome.tradingsyst
org.cocome.tradingsystem.external.Bank <Assembly_org.cocome.tradingsystem.exte

m3.large

Assembly_org.cocome.cloud.web <org.cocome.cloud.web>_m3.larg
org.cocome.cloud.web <Assembly_org.cocome.cloud.web <org.cocome.cloud.web:

Assembly_org.cocome.tradingsystem.inventory <org.cocome.tradingsystem.invent
org.cocome.tradingsystem.inventory <Assembly_org.cocome.tradingsystem.inver

Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <org.
org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <Assembly_org.

Assembly_org.cocome.cloud.webservice.inventory <org.cocome.cloud.webservice.
org.cocome.cloud.webservice.inventory <Assembly_org.cocome.cloud.webservice.

Assembly_org.cocome.tradingsystem.cashdeskline <org.cocome.tradingsystem.c
org.cocome.tradingsystem.cashdeskline <Assembly_org.cocome.tradingsystem.c

Assembly_org.cocome.tradingsystem.external.Bank <org.cocome.tradingsystem.
org.cocome.tradingsystem.external.Bank <Assembly_org.cocome.tradingsystem.

Figure 7.5: The initial allocation model of scenario 3.

We now assume that the last adaptation resulted in a consolidation of the CoCoME deployment, allocating all components onto one resource container type, with two replicas. Now a new store opens and the optimization process is started, so the application is able to satisfy it's performance requirements with the new store attached.

Current model: All components are allocated together onto two instances of the resource container type *m3.large*. The corresponding allocation model is depicted in Fig. 7.5.

Expected transformed model: All components are allocated together onto one instance of the resource container type *m3.large*. There are five additional resource containers of the type *m3.large*, one for each component, as well as six resource containers of the type *m4.2xlarge*. For each component there is a new *AllocationDegree* which allows the allocation of the component onto every resource container in the resource environment. Additionally, for each newly created container, the processing resource specifications and costs are set in the resource environment and cost model as defined by it's instance type. For each resource container in the resource environment there exists a new *ResourceContainerReplicationDegree* to allow it's replication during the optimization phase. This degree is needed to span the design space for PerOpteryx.

The model resulting from the transformation has a Jaccard coefficient of 1.0, when compared to the expected model, including the generated degrees of freedom, costs and resource specifications. Therefore, both models are equal with respect to this metric and the transformation is accurate for this scenario.

7.2.2 Scalability

With research question **RQ-1.2** we want to evaluate the scalability of the adaptation planning process. The first part to answering this question is to evaluate the scalability of the model transformation. The parameters that influence the execution time of the transformation are the following.

T: Number of instance types available in the cloud profile

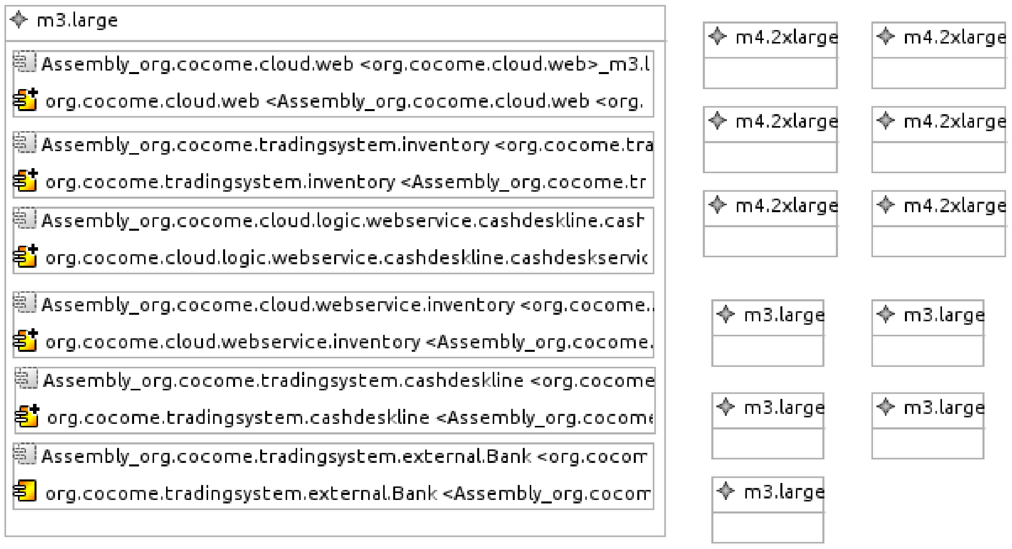

m3.large
Assembly_org.cocome.cloud.web <org.cocome.cloud.web>_m3.l
org.cocome.cloud.web <Assembly_org.cocome.cloud.web <org.
Assembly_org.cocome.tradingsystem.inventory <org.cocome.tra
org.cocome.tradingsystem.inventory <Assembly_org.cocome.tr
Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cash
org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservic
Assembly_org.cocome.cloud.webservice.inventory <org.cocome.
org.cocome.cloud.webservice.inventory <Assembly_org.cocome.
Assembly_org.cocome.tradingsystem.cashdeskline <org.cocome
org.cocome.tradingsystem.cashdeskline <Assembly_org.cocome
Assembly_org.cocome.tradingsystem.external.Bank <org.cocom
org.cocome.tradingsystem.external.Bank <Assembly_org.cocom
m4.2xlarge
m4.2xlarge
m4.2xlarge
m4.2xlarge
m4.2xlarge
m4.2xlarge
m3.large
m3.large
m3.large
m3.large
m3.large

Figure 7.6: The resulting allocation model of scenario 3 after the transformation. The names of the resource containers have been shortened for better readability.

C: Number of allocated components in the model

We first evaluate the scalability of both parameters separately by generating models with 10, 100, 1.000 and 10.000 allocated components or instance types respectively. For this scenario, it would not make sense to start with only 1 allocation context or 1 instance type. It would only evaluate the time it takes to load the model from disk, as the algorithm would not be doing much as there would be no replications or differen allocation groups. The number of resource containers in the model is kept constant at 100 and all components are allocated onto a random resource container, which is of a randomly chosen type. We chose 10000 allocated components as the upper limit, because this should be a sufficiently high number for most applications, given that an application like the content-delivery network of the Netflix video streaming service uses around 5000 servers [31]. We assume the number of instance types per cloud provider to be at around 100. This means, we represent an estimated 100 cloud providers with a total of 10000 instance types, which is sufficient for most applications.

To create the resource environment for the transformed model, our algorithm has to create a maximum of $\mathbf{T} \cdot \mathbf{C}$ resource containers. Therefore we evaluate scaling both of these parameters together. For the evaluation, we generate models with 10, 100, 1.000 and 10.000 allocated components, each allocated on a random resource container and the same number of resource container types in the cloud profile.

We use the average time it takes for 5 executions of the base model transformation to evaluate the scalability of our approach. We chose this metric to compensate for runtime effects of the Java Virtual Machine like garbage collection. The number of repetitions is enough to average out such effects.

The results of our evaluation are shown as a graph in Fig. 7.7. We use a logarithmic scaling for both axes of the graph, where the y-axis shows the average execution time

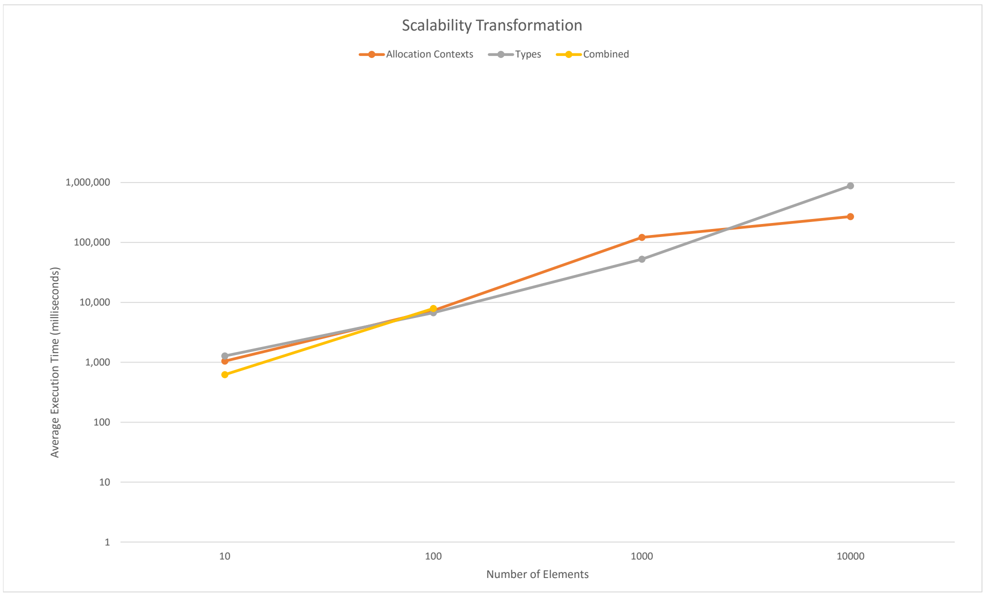


Figure 7.7: Results of the scalability measurements for the base model extraction.

in milliseconds and the x-axis the number of elements for the specific parameter. The orange line shows the average execution times when varying the number of allocation components in the allocation model, the gray line the same when varying the number of instance types and the yellow line for varying both numbers simultaneously. All other graphs in the evaluation are structured similarly.

The measured times for the scaling of allocation contexts are shown in Table 7.1, the times when scaling the instance types are shown in Table 7.2 and the times for scaling both parameters combined are displayed in Table 7.3. In these tables, the times measured for each execution are shown in milliseconds and the average time is shown in the last column.

As can be seen from the graph, the runtime of our approach increases linearly when increasing the number of allocated components or the number of instance types separately. When scaling both together, however, we encountered problems regarding the memory usage. A model with 1000 allocated components and 1000 instance types already uses more than the available 22GB of memory, because the design decision model that is being built needs that memory. This is due to the fact that in this model, there is an allocation degree for each allocation group. This means there are 1000 allocation degrees in the case of 1000 allocation groups, and one resource container replication degree for each resource container. Because there are 1000 allocation groups and 1000 instance types, the transformation will create 1000000 resource containers. The created allocation degrees need to reference all resource containers as possible deployment options. Therefore, the memory usage of the design decision model is in $O(n^2 \cdot m)$, with **n** the number of allocation groups and **m** the number of instance types. This means, that the memory usage grows rapidly, practically limiting the number of possible resource containers in the resource environment. However, this limit is hard to reach for a normal application, mainly because the use of 1000 instance types implies using 10 cloud providers with an average of 100 available instance types each, which is much more than most applications would need.

As can be seen from the results for scaling the number of allocation contexts or instance types separately, our approach scales as expected in a linear way, although the effect of the memory consumption can be seen when scaling up the instance types. For 10000 instance types, the design decision model is already at a size of 26 GB, which is more than the available memory. Therefore the operating system starts to swap out parts of the model, which leads to a slower execution of the transformation in this case.

To answer research question **RQ-1.2** for this part of the evaluation, we can conclude that our approach scales linearly with the number of allocation contexts or instance types, if they are increased separately. However, when increasing them simultaneously, the structure of the design decision model results in a very high memory consumption, so we could not determine the behavior of our approach for a larger number of allocation contexts and instance types. For most applications, however, this limitation will not be an issue.

7.3 Adaptation Calculation and Planning

The calculation and planning of adaptation steps is done to derive the adaptation actions that are necessary to transform the current application architecture into the optimized target architecture. The accuracy of this step is evaluated in Sec. 7.3.1 and in Sec. 7.3.2 we evaluate the scalability of our approach.

7.3.1 Accuracy

To answer **RQ-1.1**, the accuracy of the complete planning process has to be evaluated. The first step to do this, is to evaluate the accuracy of the base model transformation, which is done in Sec. 7.2. The next step is to evaluate the accuracy of the adaptation calculation and planning steps. We evaluate both steps at once because they are closely related. The planning step can only be accurately performed if the preceding calculation was accurate. The available adaptation actions are defined in the Systemadaptation metamodel, see Section 5.2.1.

We evaluate the accuracy of these steps by using scenarios which reflect the structural changes at runtime in Heinrich (2016, [18]). Therefore our scenarios have to cover the following changes.

- Migration of a component
- Replication of a component
- Dereplication of a component

Allocation Contexts						Average
10	1017.25	1048.93	1002.68	1061.09	1103.57	1046.70
100	7771.82	7635.07	6844.94	7149.79	7237.22	7327.77
1000	121623.67	118107.48	122952.26	127950.00	117378.48	121602.38
10000	279350.40	270473.17	264699.01	269433.24	268782.78	270547.72

Table 7.1: Times measured in milliseconds for scaling the number of allocation contexts.

Instance Types						Average
10	1191.49	1312.35	1522.55	1079.56	1272.81	1275.76
100	6599.53	6748.61	6537.41	6762.92	7009.80	6731.65
1000	53592.67	53612.99	50650.12	53481.61	51849.88	52637.45
10000	965097.04	847239.23	868442.41	869695.35	862640.81	882622.97

Table 7.2: Times measured in milliseconds for scaling the number of instance types.

Combined						Average
10	628.58	646.22	599.06	586.04	651.94	622.37
100	8728.83	7937.20	7414.10	7655.94	7817.06	7910.63
1000	-	-	-	-	-	-
10000	-	-	-	-	-	-

Table 7.3: Times measured in milliseconds for scaling the number of allocation contexts and instance types simultaneously.

- Allocation of a component
- Deallocation of a component

The replication and de-replication cases are special cases which are mapped to an acquire and an allocate action for replication. Dereplication is mapped to a deallocate and a terminate action. To evaluate the accuracy, we therefore employ the following scenarios which cover all possible changes at runtime. After each scenario, we describe the corresponding reference adaptation plan that is compared against the actual output of the adaptation planning process. For comparing the output and the reference model, we use the Jaccard coefficient as a measurement of equality of both sets. Because the order of the actions is important, we also employ Spearman's rank correlation coefficient to measure the similarity regarding the ordering of the lists. The order of the actions in the reference adaptation plan is given by the considerations in Sec. 5.3. Spearman's rank correlation coefficient is computed as follows.

$$SRCC(L_1, L_2) = 1 - \frac{6\sum_{e \in E}(rank(L_1,e)-rank(L_2,e))^2}{n(n^2-1)}$$

Scenario 1: Migration

This scenario addresses the increase of a workload on the application and the subsequent migration of components on new resource containers. It demonstrates the ability to plan the migration of components.

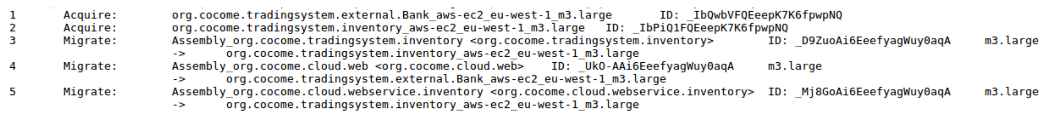
The supermarket chain just recently decided to move their system into the cloud and used just one virtual machine to test the feasibility. Now the chain decides to use a store with 10 cash desks as a front-runner to further test the application.

Runtime model: All components of CoCoME are deployed onto one resource container of the type *m3.large*.

Target model after optimization: The *TradingSystem:CashDeskLine*, *WebService:CashDesk* and *External:Bank* components are deployed on one container of the type *m3.large*. The *Web* component is allocated on another container of the type *m3.large*. The *TradingSystem:Inventory* and *WebService:Inventory* components are allocated on another container of the type *m3.large*.

Reference Adaptation Plan

1. Acquire Action: Acquire a new instance of type *m3.large*
2. Acquire Action: Acquire a new instance of type *m3.large*
3. Migrate Action: Migrate the *Web* component to the first newly acquired *m3.large* resource container.
4. Migrate Action: Migrate the *TradingSystem:Inventory* component to the second newly acquired *m3.large* resource container.



```
1    Acquire:   org.cocome.tradingsystem.external.Bank_aws-ec2_eu-west-1_m3.large       ID: _IbQwbVFQEeepK7K6fpwpNQ
2    Acquire:   org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m3.large   ID: _IbPiQ1FQEeepK7K6fpwpNQ
3    Migrate:   Assembly_org.cocome.tradingsystem.inventory <org.cocome.tradingsystem.inventory>        ID: _D9ZuoAi6EeefyagWuy0aqA     m3.large
                ->      org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m3.large
4    Migrate:   Assembly_org.cocome.cloud.web <org.cocome.cloud.web>    ID: _UkO-AAi6EeefyagWuy0aqA     m3.large
                ->      org.cocome.tradingsystem.external.Bank_aws-ec2_eu-west-1_m3.large
5    Migrate:   Assembly_org.cocome.cloud.webservice.inventory <org.cocome.cloud.webservice.inventory>  ID: _Mj8GoAi6EeefyagWuy0aqA     m3.large
                ->      org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m3.large
```

Figure 7.8: The resulting adaptation plan for scenario 1.

5. Migrate Action: Migrate the *WebService:Inventory* component to the second newly acquired *m3.large* resource container.

The actually derived adaptation plan for this scenario is shown in Fig. 7.8. The Jaccard coefficient for this plan is 1.0, indicating that the planned adaptation actions are the same as in the reference adaptation plan. The Spearman rank correlation coefficient is 0.9, indicating a strong correlation. This value is below 1, because there are only five observations and two observations differ in their position. This difference in position is not relevant in this case, because the differing observations are both migrate actions which swapped their position. This swapping of positions within a group does not affect the execution of the adaptation plan. Therefore, we conclude that the adaptation planning for this scenario is accurate, except for irrelevant changes in the execution order within the same group of adaptation actions.

Scenario 2: Replication, Acquisition, Allocation

With this scenario we address the replication of components following an increased workload. Because replication is seen as the sequence of an acquisition followed by an allocation, these cases are addressed with this scenario as well. It demonstrates the ability to accurately plan these steps while simultaneously also migrating components to increase the complexity of the scenario. It also demonstrates the ability to terminate resource containers that are no longer needed because no component is allocated on them.

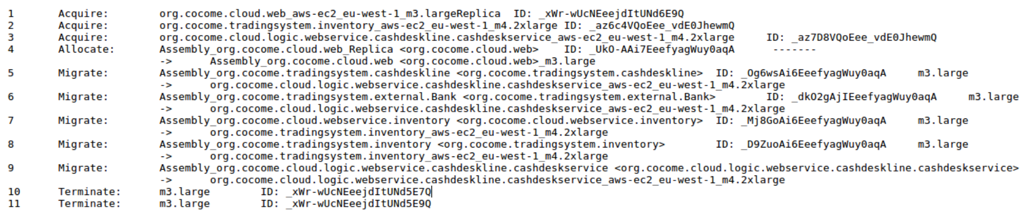
The supermarket chain decides to launch a big sales campaign, leading to a rise in the number of customers. In addition to the 10 cash desks already used in the store, 5 more express cash desks are opened and the application has to adapt to the higher demand.

Initial model: The *TradingSystem:Inventory* and *WebService:Inventory* components are deployed onto one virtual machine of the type *m3.large*. The *TradingSystem: CashDeskLine*, *WebService:CashDesk* and *External:Bank* components are deployed on another *m3.large* virtual machine. The frontend in the *Web* component is also deployed on an *m3.large* instance.

Target model: The *TradingSystem:Inventory* and *WebService:Inventory* components are migrated to one virtual machine of the type *m4.2xlarge*. The *TradingSystem: CashDeskLine*, *WebService:CashDesk* and and *External:Bank* components are migrated onto a *m4.2xlarge* virtual machine. The frontend in the *Web* component is replicated on a second *m3.large* instance.

Reference Adaptation Plan: Scenario 2

1. Acquire Action: Acquire a new instance of type *m3.large*



```
1       Acquire:        org.cocome.cloud.web_aws-ec2_eu-west-1_m3.largeReplica  ID: _xWr-wUcNEeejdItUNd6E9Q
2       Acquire:        org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m4.2xlarge ID: _az6c4VQoEee_vdE0JhewmQ
3       Acquire:        org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice_aws-ec2_eu-west-1_m4.2xlarge     ID: _az7D8VQoEee_vdE0JhewmQ
4       Allocate:       Assembly_org.cocome.cloud.web_Replica <org.cocome.cloud.web>    ID: _UkO-AAi7EeefyagWuy0aqA      -------
                        ->      Assembly_org.cocome.cloud.web <org.cocome.cloud.web>_m3.large
5       Migrate:        Assembly_org.cocome.tradingsystem.cashdeskline <org.cocome.tradingsystem.cashdeskline>  ID: _Og6wsAi6EeefyagWuy0aqA     m3.large
                        ->      org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice_aws-ec2_eu-west-1_m4.2xlarge
6       Migrate:        Assembly_org.cocome.tradingsystem.external.Bank <org.cocome.tradingsystem.external.Bank>        ID: _dkO2gAjIEeefyagWuy0aqA     m3.large
                        ->      org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice_aws-ec2_eu-west-1_m4.2xlarge
7       Migrate:        Assembly_org.cocome.cloud.webservice.inventory <org.cocome.cloud.webservice.inventory>  ID: _Mj8GoAi6EeefyagWuy0aqA     m3.large
                        ->      org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m4.2xlarge
8       Migrate:        Assembly_org.cocome.tradingsystem.inventory <org.cocome.tradingsystem.inventory>        ID: _D9ZuoAi6EeefyagWuy0aqA     m3.large
                        ->      org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m4.2xlarge
9       Migrate:        Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice>
                        ->      org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice_aws-ec2_eu-west-1_m4.2xlarge
10      Terminate:      m3.large        ID: _xWr-wUcNEeejdItUNd5E7Q
11      Terminate:      m3.large        ID: _xWr-wUcNEeejdItUNd5E9Q
```

Figure 7.9: The resulting adaptation plan for scenario 2.

2. Acquire Action: Acquire a new instance of type *m4.2xlarge*

3. Acquire Action: Acquire a new instance of type *m4.2xlarge*

4. Allocate Action: Allocate the *Web* component to a new instance of type *m3.large*

5. Migrate Action: Migrate the *TradingSystem:Inventory* component to the first newly acquired *m4.2xlarge* instance.

6. Migrate Action: Migrate the *WebService:Inventory* component to the first newly acquired *m4.2xlarge* instance.

7. Migrate Action: Migrate the *TradingSystem:CashDeskLine* component to the second newly acquired *m4.2xlarge* instance.

8. Migrate Action: Migrate the *WebService:CashDesk* component to the second newly acquired *m4.2xlarge* instance.

9. Migrate Action: Migrate the *External:Bank* component to the second newly acquired *m4.2xlarge* instance.

10. Terminate Action: Terminate the *m3.large* instance from which the inventory components were migrated.

11. Terminate Action: Terminate the *m3.large* instance from which the cash desk components were migrated.

The actually derived adaptation plan for this scenario is shown in Fig. 7.9. The planned adaptation actions are the same as in the reference adaptation plan, which is reflected by the Jaccard coefficient of 1.0 for this adaptation plan. The Spearman rank correlation coefficient is 0.8909, indicating a strong correlation. The value below 1 is again caused by the fact that the positions within the group of migration actions are swapped. Therefore, we conclude that the adaptation planning for this scenario is accurate.

Scenario 3: Dereplication, Deallocation, Termination

This scenario addresses the dereplication of components following a decrease in the workload. Because the dereplication is mapped as the sequence of a deallocation followed by a termination, these cases are addresses with this scenario as well. It demonstrates the ability to accurately plan these steps while simultaneously also migrating components to increase the complexity of the scenario.

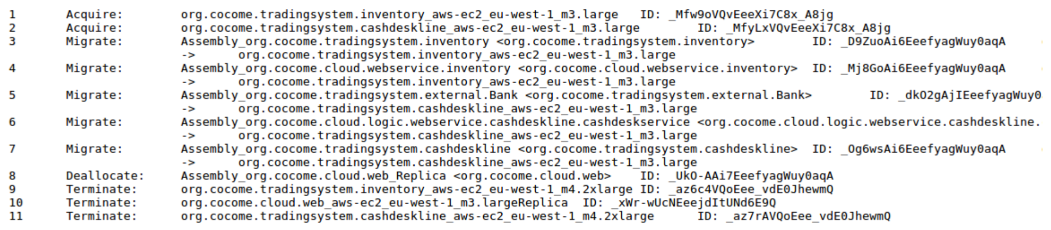
After the sales campaign is over, the number of customers declines and the additional 5 express cash desks are closed again. The application therefore has to adapt to the decreased workload to optimize the costs of the deployment.

Initial model: The *TradingSystem:Inventory* and *WebService:Inventory* components are deployed on one resource container of the type *m4.2xlarge*. The *TradingSystem: CashDeskLine*, *WebService:CashDesk* and *External:Bank* components are deployed onto a *m4.2xlarge* resource container. The frontend in the *Web* component is deployed on two *m3.large* instances.

Target model: The *TradingSystem:Inventory* and *WebService:Inventory* components are migrated onto one resource container of the type *m3.large*. The *TradingSystem: CashDeskLine* and *WebService:CashDesk* components are migrated onto another *m3.large* virtual machine. The frontend in the *Web* component stays deployed on only one *m3.large* instance and the other *m3.large* instance is dereplicated.

Reference Adaptation Plan: Scenario 3

1. Acquire Action: Acquire a new instance of type *m3.large*
2. Acquire Action: Acquire a new instance of type *m3.large*
3. Migrate Action: Migrate the *TradingSystem:Inventory* component to the first newly acquired *m3.large* instance.
4. Migrate Action: Migrate the *WebService:Inventory* component to the first newly acquired *m3.large* instance.
5. Migrate Action: Migrate the *TradingSystem:CashDeskLine* component to the second newly acquired *m3.large* instance.
6. Migrate Action: *WebService:CashDesk* component to the second newly acquired *m3.large*instance.
7. Migrate Action: Migrate the *External:Bank* component to the second newly acquired *m3.large* instance.
8. Deallocate Action: Deallocate the *Web* component from the replicated *m3.large* instance.
9. Terminate Action: Terminate the *m4.2xlarge* instance from which the inventory components were migrated.
10. Terminate Action: Terminate the *m4.2xlarge* instance from which the cash desk components were migrated.



```
1       Acquire:        org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m3.large   ID: _Mfw9oVQvEeeXi7C8x_A8jg
2       Acquire:        org.cocome.tradingsystem.cashdeskline_aws-ec2_eu-west-1_m3.large        ID: _MfyLxVQvEeeXi7C8x_A8jg
3       Migrate:        Assembly_org.cocome.tradingsystem.inventory <org.cocome.tradingsystem.inventory>        ID: _D9ZuoAi6EeefyagWuy0aqA
                        ->      org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m3.large
4       Migrate:        Assembly_org.cocome.cloud.webservice.inventory <org.cocome.cloud.webservice.inventory>  ID: _Mj8GoAi6EeefyagWuy0aqA
                        ->      org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m3.large
5       Migrate:        Assembly_org.cocome.tradingsystem.external.Bank <org.cocome.tradingsystem.external.Bank>        ID: _dkO2gAjIEeefyagWuy0
                        ->      org.cocome.tradingsystem.cashdeskline_aws-ec2_eu-west-1_m3.large
6       Migrate:        Assembly_org.cocome.cloud.logic.webservice.cashdeskline.cashdeskservice <org.cocome.cloud.logic.webservice.cashdeskline.
                        ->      org.cocome.tradingsystem.cashdeskline_aws-ec2_eu-west-1_m3.large
7       Migrate:        Assembly_org.cocome.tradingsystem.cashdeskline <org.cocome.tradingsystem.cashdeskline>  ID: _Og6wsAi6EeefyagWuy0aqA
                        ->      org.cocome.tradingsystem.cashdeskline_aws-ec2_eu-west-1_m3.large
8       Deallocate:     Assembly_org.cocome.cloud.web_Replica <org.cocome.cloud.web>    ID: _UkO-AAi7EeefyagWuy0aqA
9       Terminate:      org.cocome.tradingsystem.inventory_aws-ec2_eu-west-1_m4.2xlarge ID: _az6c4VQoEee_vdE0JhewmQ
10      Terminate:      org.cocome.cloud.web_aws-ec2_eu-west-1_m3.largeReplica  ID: _xWr-wUcNEeejdItUNd6E9Q
11      Terminate:      org.cocome.tradingsystem.cashdeskline_aws-ec2_eu-west-1_m4.2xlarge      ID: _az7rAVQoEee_vdE0JhewmQ
```

Figure 7.10: The resulting adaptation plan for scenario 3.

11. Terminate Action: Terminate the *m3.large* instance to which the web component was replicated.

Fig. 7.10 depicts the derived adaptation plan for this scenario. For this adaptation plan, the Jaccard coefficient is 1.0 and shows that the actions are the same for the derived plan and the reference plan. The Spearman rank correlation coefficient is 0.9545, which indicates a strong correlation. Here the positions within the migration actions and the termination actions differ slightly, which results in the value below 1. This is, however, as mentioned before not a problem for the adaptation execution. Therefore, we conclude that the adaptation planning for this scenario is accurate.

7.3.2 Scalability

To evaluate the scalability of the complete planning process and answer research question **RQ-1.2**, we evaluate the scalability of the adaptation calculation and planning as the second part.

The important parameter for evaluating the adaptation calculation and planning is the number of adaptation actions that have to be calculated and ordered. To get to a specific number of adaptation actions, we use one generated initial model and generate a derived model with random adaptations. The concrete type of adaptation actions which is used is not relevant for the scalability, because no action requires special computations when calculating or ordering it. Therefore we uniformly distribute the number of overall adaptation actions onto all available action types and generate the derived model.

We chose to evaluate the scalability by using target models with 10, 100, 1000 and 10000 adaptation actions. We chose to omit a target model with 1 adaptation action, because the time taken to load the model from disk would distort the measurements in this case. These adaptation actions are generated on the basis of a valid generated model with no semantic meaning. We use a tool to derive a modified model with the specified number of adaptation actions. The derived model is again a valid model and therefore these two models form a valid basis for the evaluation of our approach. As mentioned in Sec. 7.2.2, 10000 adaptation actions involving at least the same number of resource containers is more than the majority of applications would require. For our evaluation, we use the average time it takes for 5 executions of the adaptation planning to evaluate the scalability of our approach. Again, we chose this metric to compensate for runtime effects of the Java Virtual Machine.

In Fig. 7.11, the results of the evaluation are shown as a graph and the underlying measurements can be seen in Table 7.4. The graph shows that our approach is performing very

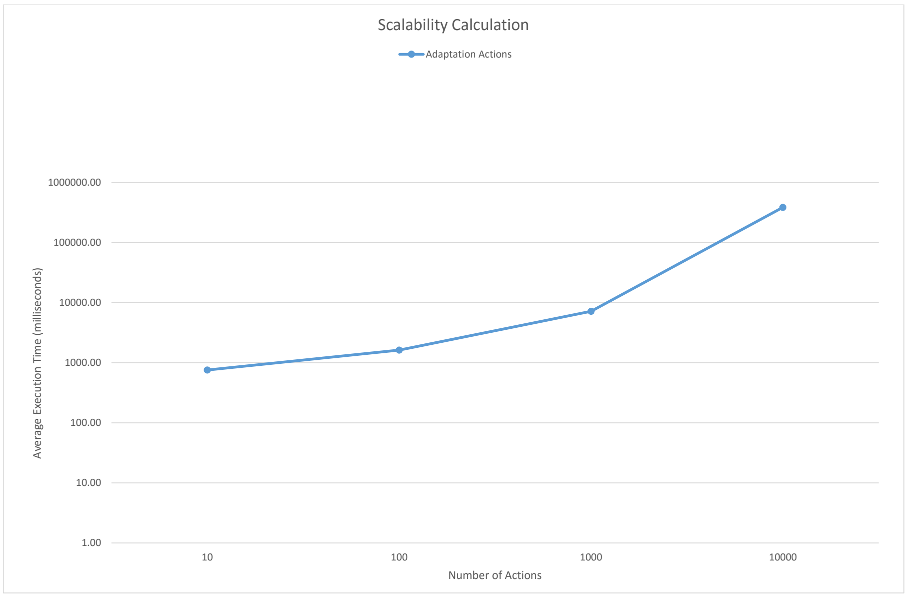


Figure 7.11: Results of the scalability measurements for the adaptation calculation.

well for up to 1000 adaptation actions, where it needs even less time than expected, when looking at the execution time for 10 adaptation actions. For 10000 adaptation actions, however, the execution time increases sharply. Further investigation revealed that more than 60% of the time is spent loading the original model and the target model from disk, whereas less than 40% of the time is spent in the planning algorithm itself. We therefore conclude that the adaptation planning algorithm itself scales linearly, as expected, although the loading of the models is an issue for the performance.

However, when considering the absolute values, the time for planning the execution of 10000 adaptation actions is still in $O(n)$, with **n** the number of adaptation actions. With an average execution time of 758.58 ms for 10 adaptation actions, it could be expected that the execution time for 10000 adaptation actions is at about 750000 ms. Therefore the actual average execution time of 388475.95 ms is in an acceptable range. With these results, we can answer this part of the research question **RQ-1.2**. The adaptation calculation step scales linearly with the number of adaptation actions.

7.4 Adaptation Execution

We evaluate the adaptation execution in order to answer the research questions **RQ-2.1** and **RQ-2.2**. First, we evaluate the accuracy of our approach in Sec. 7.4 and we conclude this section by evaluating the scalability of our approach in Sec. 7.4.2.

Adaptation Actions						Average
10	613.66	709.80	785.39	861.58	822.48	758.58
100	2062.82	1580.67	1554.45	1448.26	1500.03	1629.24
1000	7081.14	7001.65	7287.40	7370.13	7295.85	7207.23
10000	383364.24	388608.08	395989.88	390971.43	383446.10	388475.95

Table 7.4: Times measured in milliseconds for scaling the number of adaptation actions when planning the adaptations.

Adaptation Actions						Average
10	11.42	11.64	17.33	14.94	14.20	13.91
100	45.98	45.11	48.89	40.82	36.19	43.40
1000	204.23	229.12	189.24	163.40	219.18	201.03
10000	1072.71	1173.58	1123.08	1231.02	1212.21	1162.52

Table 7.5: Times measured in milliseconds for scaling the number of adaptation actions when executing the adaptations.

7.4.1 Accuracy

The accuracy with which a derived adaptation plan is executed provides an answer to **RQ-2.1**. To evaluate this, we use the reference adaptation plans introduced in Sec. 7.3 which describe adaptation actions that have to be performed on the target system. These plans already cover the set of changes at runtime for performance related adaptations.

Therefore, to evaluate the accuracy of the adaptation execution, we track the actions that are executed by this step and compare them with the expected executions based on the adaptation plans. The tracking is done by issuing a log statement for each executed action. This is possible, because the adaptation plans can be mapped directly to the executed adaptations. Again, we employ the Jaccard coefficient and the Spearman rank correlation coefficient, because they are easy to calculate and show the level of similarity of the two lists. The Spearman rank correlation coefficient is useful here, again because the order of the elements in the list is used in it's calculation. This is necessary because here, as well as in Sec. 7.3.1, the ordering of elements in the list is important to enable the accurate execution of adaptations.

The accuracy of the execution is shown by the Jaccard coefficients of 1.0 for each of the three scenarios, when compared to the reference adaptation plans. For all scenarios, the Spearman rank correlation coefficient is equal to the one for the adaptation planning, resulting in a value of 0.9 for the first scenario, 0.8909 for the second scenario and 0.9545 for the third. This indicates that the adaptation execution is accurately executing the adaptation plans derived from the step before.

7.4.2 Scalability

To evaluate the scalability of the adaptation execution and answer research question **RQ-2.2**, we use a similar approach as with the evaluation of the scalability for adaptation planning. The important parameter for evaluating the adaptation execution is the number of adaptation actions that has to be performed. We assume that all adaptation actions have the same effect on the scalability because, all actions only perform a constant number of calls to the service providers interfaces.

The evaluation of the execution process' scalability therefore relies on generated Systemadaptation models with a specific number of adaptation actions. For our measurements we exclude the execution of the calls to the cloud provider's interfaces, as well as the execution times of the actual scripts. Both are highly dependent on external factors like network latency or workload on the cloud providers infrastructure. Moreover, the execution scripts that are executed on the servers are dependent on the application and as such not part of this thesis. Therefore we assume that these scripts behave as intended by their engineers. Their execution times also heavily depend on the used technologies and the hardware of the virtual machine. Therefore these steps are not evaluated.

The design of this evaluation is similar to the design in Sec. 7.3.2. We use adaptation plans with 10, 100, 1000 and 10000 elements. These numbers cover adaptations for very large applications and are therefore sufficient to show the scalability of our approach. We use randomly generated adaptation actions, because the concrete type of action does not influence the runtime of the execution. As a metric we use the average time it takes

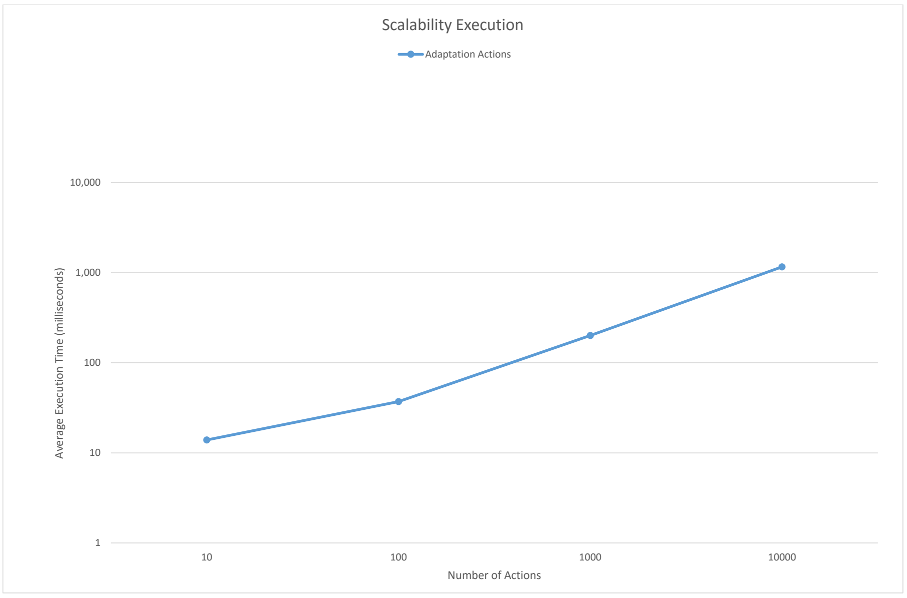


Figure 7.12: Results of the scalability measurements for the adaptation execution.

for 5 executions of the adaptation planning to evaluate the scalability of our approach to compensate for runtime effects of the Java Virtual Machine.

We show the results of our measurements in Fig. 7.12 and the exact values for each iteration as well as the average are listed in Table 7.5. The graph shows that the adaptation execution is performed in $O(n)$ with n being the number of adaptation actions. Because we do not consider the execution times that result from the actual execution on the resource containers and from calling the cloud provider's interfaces, the absolute values will be higher in practice. Therefore as the answer to research question **RQ-2.2**, our evaluation shows that the adaptation execution scales linearly with the number of adaptation actions.

8 Conclusion

In this thesis, we presented and evaluated our approach for automatic adaptation planning in cloud-based applications with an operator-in-the-loop using component-based architectural models. We showed the benefits of using an evolutionary algorithm for the design space exploration when optimizing the architecture of cloud-based applications and developed an approach to integrate PerOpteryx into the MAPE loop of iObserve. Our approach is both accurate and scalable and therefore fulfills the goals formulated in Sec. 1.3. To conclude this thesis, we discuss the limitations of our approach in Sec. 8.1 and point out possibilities for future work in Sec. 8.2.

8.1 Limitations

Our approach currently considers all components in one system to be independent of each other and also to be independently deployable. This is consistent with the idea of component-based software engineering. However, in practice it might often be useful to specify constraints on components. For example it may be beneficial for the *WebService:CashDesk* component of CoCoME to be deployed on the same resource container as the *TradingSystem:CashDeskLine*, because it results in less overhead for the communication.

Another limitation is that the approach is not able to plan adaptations according to dependencies between them. That means if there are dependencies between two or more components, those dependencies will not be taken into account when calculating the order of adaptation actions. All components are again regarded as independent in this regard.

Moreover, our approach is limited with respect to the number of instance types and allocation contexts it supports. The evaluation showed that the current approach is not able to handle 1000 allocation contexts and the same number of instance types in the same model, because of the memory used in this scenario.

8.2 Future Work

During the course of this thesis, several possibilities for future work were identified. Specifically, we encountered some possible enhancements for the PCM and PerOpteryx from which our approach could benefit. Moreover, during the evaluation, we discovered performance limitations that should be investigated further. We therefore present the following list of open tasks for future work.

KAMP Integration As one alternative to the adaptation calculation, we proposed to integrate KAMP into our approach. This would benefit not only the adaptation calculation by providing information on dependencies between artifacts. It could also provide an operator with better information about tasks that have to be done manually. Therefore we propose the integration of KAMP into our approach as a further step.

PCM Replication Model We introduced the transformation of a PCM model into it's base model representation, because the PCM is not built to handle replicated resource containers. Our approach would, however, benefit greatly from a version of the PCM which supports some form of basic model of the application and that can be enriched with information about replicated resource containers.

PerOpteryx Integration One of the problems discovered by our evaluation is directly related to the design decision model of PerOpteryx and could be prevented by improving the way degrees of freedom are saved. Another task is to integrate PerOpteryx into our approach not as a call to a completely separate application, but by directly integrating it's source code. At the moment this is not possible because of the dependencies from PerOpteryx to Eclipse. Therefore we propose a redesign of PerOpteryx, to decouple it's core functionality from Eclipse.

Artificial Neural Networks Artificial neural networks have been successfully employed to explore large search spaces, for example for the board game Go. Recent research indicates that it may be possible to tackle many complex tasks with such an approach [37]. However, to the best of our knowledge, there is not a lot of research on how to use such networks for design space exploration of architectural models.

PaaS integration We currently only support the IaaS service model for the cloud. It would be beneficial to also include support for the PaaS service model to be able to leverage these types of deployments as well.

Bibliography

[1] Various Authors. *iObserve Github Repository*. URL: `https://github.com/research-iobserve/iobserve-analysis%20(Last%20accessed:%20June%202017)`.

[2] V. R. Basili and D. M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984), pp. 728–738. ISSN: 0098-5589. DOI: `10.1109/TSE.1984.5010301`.

[3] Matthias Becker, Steffen Becker, and Joachim Meyer. “SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems.” In: *Software Engineering* 213 (2013), pp. 71–84.

[4] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “The Palladio Component Model for Model-driven Performance Prediction”. In: *Journal of Systems and Software* 82 (2009), pp. 3–22. DOI: `10.1016/j.jss.2008.03.066`. URL: `http://dx.doi.org/10.1016/j.jss.2008.03.066`.

[5] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[6] Gordon Blair, Nelly Bencomo, and Robert B France. “Models@ run. time”. In: *Computer* 42.10 (2009), pp. 22–27.

[7] Gerardo Canfora et al. “A framework for QoS-aware binding and re-binding of composite web services”. In: *Journal of Systems and Software* 81.10 (2008), pp. 1754–1769.

[8] Cloud Management Working Group Distributed Management Task Force. *Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol.* Aug. 2016. URL: `http://www.dmtf.org/standards/cloud`.

[9] Andy Edmonds et al. “Toward an open cloud standard”. In: *IEEE Internet Computing* 16.4 (2012), pp. 15–25.

[10] Matthias Ehrgott. *Multicriteria optimization*. Springer Science & Business Media, 2006.

[11] Nicolas Ferry et al. “Cloud MF: Applying mde to tame the complexity of managing multi-cloud applications”. In: *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society. 2014, pp. 269–277.

[12] Nicolas Ferry et al. “Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems”. In: *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE. 2013, pp. 887–894.

[13] Greg Franks et al. “Enhanced modeling and solution of layered queueing networks”. In: *IEEE Transactions on Software Engineering* 35.2 (2009), pp. 148–161.

[14] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. “Search-based genetic optimization for deployment and reconfiguration of software in the cloud”. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 512–521.

[15] Sören Frey and Wilhelm Hasselbring. “The cloudmig approach: Model-based migration of software systems to cloud-optimized applications”. In: *International Journal on Advances in Software* 4.3 and 4 (2011), pp. 342–353.

[16] Svend Frølund and Jari Koistinen. *Qml: A language for quality of service specification.* Hewlett-Packard Laboratories, 1998.

[17] R Heinrich, K Rostami, and R Reussner. “The CoCoME platform for collaborative empirical research on information system evolution”. In: *Karlsruhe Reports in Informatics, Tech. Rep* (2016).

[18] Robert Heinrich. “Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications”. In: *ACM SIGMETRICS Performance Evaluation Review* 43.4 (2016), pp. 13–22.

[19] Robert Heinrich et al. “Architectural run-time models for operator-in-the-loop adaptation of cloud applications”. In: (2015).

[20] Robert Heinrich et al. “Software Architecture for Big Data and the Cloud”. In: to appear. Elsevier, 2017. Chap. An Architectural Model-Based Approach to Quality-aware DevOps in Cloud Applications.

[21] Maurice G Kendall. “A new measure of rank correlation”. In: *Biometrika* 30.1/2 (1938), pp. 81–93.

[22] Anne Koziolek. *Automated improvement of software architecture models for performance and other quality attributes.* Vol. 7. KIT Scientific Publishing, 2014.

[23] Heiko Koziolek et al. “Evaluating performance of software architecture models with the Palladio component model”. In: *Model-Driven Software Development: Integrating Quality Assurance, IDEA Group Inc* (2008), pp. 95–118.

[24] Manny M Lehman and Laszlo A Belady. *Program evolution: processes of software change.* Academic Press Professional, Inc., 1985.

[25] Maksym Lushpenko et al. “Using Adaptation Plans to Control the Behavior of Models@ runtime”. In: ().

[26] Heiko Martens et al. “Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms”. In: *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM. 2010, pp. 105–116.

[27] Peter Mell, Tim Grance, et al. “The NIST definition of cloud computing”. In: (2011).

[28] Philipp Merkle and Jörg Henss. “EventSim–an event-driven Palladio software architecture simulator”. In: *Palladio Days* (2011), pp. 15–22.

[29] Andreas Metzger et al. "Coordinated run-time adaptation of variability-intensive systems: an application in cloud computing". In: *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*. ACM. 2016, pp. 5–11.

[30] Brice Morin et al. "Models at runtime to support dynamic adaptation". In: *Computer* 42.10 (2009).

[31] Amy Nordrum. *Researchers Map Locations of 4,669 Servers in Netflix's Content Delivery Network*. Aug. 2016. URL: `http://spectrum.ieee.org/tech-talk/telecom/internet/researchers-map-locations-of-4669-servers-in-netflixs-content-delivery-network%20(Last%20accessed:%20June%202017)`.

[32] Ralf Reussner et al. "The Palladio Component Model". In: (2011).

[33] Kiana Rostami et al. "Architecture-based assessment and planning of change requests". In: *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. IEEE. 2015, pp. 21–30.

[34] Misha Strittmatter et al. "A Modular Reference Structure for Component-based Architecture Description Languages." In: *ModComp@ MoDELS*. 2015, pp. 36–41.

[35] Clemens Szyperski, D Gruntz, and S Murer. "Component software: beyond object-oriented programming. 2002". In: *ISBN: 0-201-74572-0* (2002).

[36] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[37] Fei-Yue Wang et al. "Where does AlphaGo go: from Church-Turing thesis to AlphaGo thesis and beyond". In: *IEEE/CAA Journal of Automatica Sinica* 3.2 (2016), pp. 113–120.