Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language

Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, *Member, IEEE* and Manuel Bähr

Abstract—Modern IT systems have increasingly distributed and dynamic architectures providing flexibility to adapt to changes in the environment and thus enabling higher resource efficiency. However, these benefits come at the cost of higher system complexity and dynamics. Thus, engineering systems that manage their end-to-end application performance and resource efficiency in an autonomic manner is a challenge. In this article, we present a holistic model-based approach for self-aware performance and resource management leveraging the Descartes Modeling Language (DML), an architecture-level modeling language for online performance and resource management. We propose a novel online performance prediction process that dynamically tailors the model solving depending on the requirements regarding accuracy and overhead. Using these prediction capabilities, we implement a generic model-based control loop for proactive system adaptation. We evaluate our model-based approach in the context of two representative case studies showing that with the proposed methods, significant resource efficiency gains can be achieved while maintaining performance requirements. These results represent the first end-to-end validation of our approach, demonstrating its potential for self-aware performance and resource management in the context of modern IT systems and infrastructures.

Index Terms—Autonomic, Self-Aware, Adaptation, Model-Based, Modeling Language, Performance, Efficiency

1 INTRODUCTION

MODERN software systems have increasingly distributed architectures composed of loosely-coupled services that are typically deployed on virtualized infrastructures. Such system architectures provide increased flexibility by abstracting from the physical infrastructure and supporting a flexible mapping of virtual to physical resources. This enables the dynamic consolidation of system resources and their sharing among multiple independent applications, thus making it possible to significantly improve system efficiency. However, these benefits come at the cost of higher system complexity and dynamics, due to the introduced gap between physical and virtual resource allocations as well as the complex interactions between the applications sharing the physical infrastructure. The inability to predict the effects of such interactions and the inherent semantic gap between application-level metrics and resource allocations at the physical and virtual layers significantly increase the complexity of managing end-to-end application performance. Thus, many researchers in industry and academia are working on techniques for designing self-adaptive systems that automatically manage their performance, resource efficiency, and other quality-of-service (QoS) properties during operation.

State-of-the-art industrial approaches for automated performance and resource management in virtualized environments generally follow a trigger-based approach when it comes to enforcing application-level service-level agreements (SLAs) [1], [2], [3]. Custom triggers can be configured that fire when a metric reaches a certain threshold (e.g., high resource utilization or load imbalance) and execute certain predefined reconfiguration actions until a given stopping criterion is fulfilled. The problem is that application-level metrics (such as response time) normally exhibit a non-linear behavior on system load and they typically depend on the behavior of multiple virtual machines (VMs) across several application tiers. Generally, it is hard to predict how changes in the application workloads (e.g., varying request arrival rates and/or transaction mix) propagate through the layers and tiers of the system architecture down to the physical resource layer. Therefore, it is hard to determine general thresholds when triggers should be fired given that such triggers are typically highly dependent on the architecture of the hosted services and their workloads.

To tackle the above-mentioned challenges, techniques for *online* performance prediction are needed. Such techniques should make it possible to continuously predict at run-time: a) changes in the application workloads, b)

- *N. Huber, F. Brosig, S. Spinner and S. Kounev are with the Chair of Software Engineering, Department of Computer Science, University of Würzburg, Würzburg, Germany. E-mail: {firstname.lastname}@uni-wuerzburg.de*
- *M. Bähr is with Blue Yonder GmbH & Co. KG., Karlsruhe, Germany. E-mail: manuel.baehr@blue-yonder.com*

This work was funded by the Deutsche Forschungsgemeinschaft (DFG) under grant No. KO 3445/11-1 and grant No. KO 3445/15-1. Parts of this work have been done during Fabian Brosig's summer internship at Salesforce.com. We acknowledge the many fruitful discussions with Jürgen Walter and Simon Eismann from the Chair of Software Engineering, University of Würzburg.

the effect of such changes on the system performance and resource efficiency, and c) the impact of possible adaptation actions at run-time. Recent approaches proposed in the research community have applied different types of models to provide such prediction capabilities. Existing work in this area mainly uses coarse-grained performance models that typically abstract systems and applications at a high level, e.g., [4], [5], [6], [7], [8], [9]. Such models do not explicitly model the software architecture and execution environment to distinguish performance-relevant behavior at the virtualization level vs. at the level of applications hosted inside the running virtual machines. The individual effects and complex interactions between the application workloads and the system components and layers are considered as static and viewed as a black box in such models. This hinders fine-grained performance predictions that are necessary for efficient resource management, e.g., predicting the effect on the response times of different services, if a virtual machine in a given application tier is to be replicated or migrated to another host, possibly with a different configuration.

In the software performance engineering community, a number of modeling approaches for building architecture-level performance models of software systems have been proposed over the last decade [10]. Such models provide modeling constructs to capture the performance-relevant behavior of a system's software architecture as well as some aspects of its execution environment. However, they are designed for use at system design-time in an offline setting. Therefore, they typically assume a static system architecture and execution environment and are thus unsuitable for use as a basis for system adaptation in an online setting.

In autonomic computing, software models play an important role in managing the complexity of dynamic and self-adaptive systems and supporting the adaptation decisions in such environments [11], [12]. However, approaches that employ architectural models, such as [13], [14], [15], usually focus only on adaptation at the application level and do not include the system's operational environment within the scope of the considered adaptation possibilities. Furthermore, adaptation decisions are typically based on simple policies without the possibility to predict the impact of possible adaptation actions at run-time on the end-to-end system performance, and feed this into the decision process.

In summary, we argue that sophisticated modeling and prediction techniques are needed, specifically designed for performance and resource management in online scenarios. Such techniques should allow to capture both static and dynamic system aspects including all relevant influences of the system's resource landscape, its architecture, as well as its adaptation space, adaptation strategies and processes, in a generic, human-understandable and reusable way. Moreover, we need model-based system adaptation mechanisms that apply such modeling and prediction techniques end-to-end, to drive autonomic decision making at run-time [11].

In our previous work [16], [17], [18], we proposed the Descartes Modeling Language (DML), an end-to-end modeling formalism for online performance and resource management specified by metamodels based on OMG's Meta-object Facility (MOF) [19]. This work has been focused on finding suitable abstractions for describing the performance-relevant aspects of applications [18], the influence of their resource environment [16] and the processes for adapting the system according to the high-level goals specified in SLAs [17]. However, the existing work does not cover the question how these models can be used to build an end-to-end control loop for proactive performance and resource management. In particular, sophisticated online prediction techniques for evaluating the impact of changes in the workload or system configuration on the application performance are missing.

This article uses DML as a foundation and makes the following contributions on top of it: i) an end-to-end approach for self-aware performance and resource management based on a holistic model-based adaptation control loop exploiting the reflective capabilities of DML, ii) a novel online performance prediction process that dynamically tailors the model solving taking into account the requested performance metrics as well as goals in terms of accuracy and overhead, and iii) the first real-world evaluation and validation of our approach in the context two industrial case studies demonstrating the benefits of model-based adaptation control loop and the online performance prediction process. for performance and resource management in modern dynamic IT systems, infrastructures and services.

We apply and evaluate our approach end-to-end in the context of two different case studies conducted in cooperation with industrial partners. The first case study is based on a Customer Relationship Management (CRM) application of a large Software-as-a-Service (SaaS) provider. The results show that our novel online performance prediction process provides the flexibility to dynamically trade-off between prediction accuracy and overhead enabling the system to efficiently reason on its performance behavior under different conditions. In the second case study, we apply our approach for self-aware performance and resource management to a software system of a leading provider of predictive analytics and big data cloud services. The results show the capabilities of our approach to trade-off different performance requirements of multiple customers in a heterogeneous system environment while maintaining resource efficiency

The remainder of this article is structured as follows: First, Section 2 lays the foundation with a coherent presentation of the concepts of DML. In Section 3, we present the concepts and realization of our model-based adaptation approach. Section 4 describes the online performance prediction process. Section 5 presents and discusses the results of our case studies. Section 6 gives

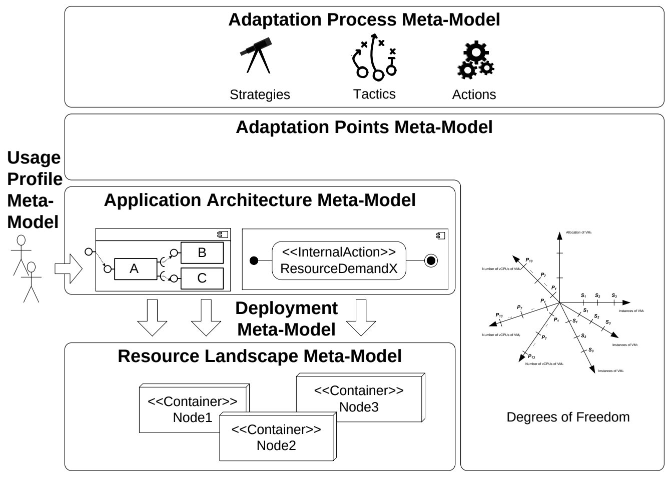

Adaptation Process Meta-Model
Strategies
Tactics
Actions
Adaptation Points Meta-Model
Usage Profile Meta-Model
Application Architecture Meta-Model
A
B
C
<<InternalAction>> ResourceDemandX
Deployment Meta-Model
Resource Landscape Meta-Model
<<Container>> Node1
<<Container>> Node2
<<Container>> Node3
Degrees of Freedom

Fig. 1. Structure of the Descartes Modeling Language.

an overview of related approaches in the area of model-based system reconfiguration and system adaptation, before we conclude with Section 7.

2 The Descartes Modeling Language

We start by presenting an overview of the Descartes Modeling Language (DML), which is the basis for our approach. Technically, DML is comprised of several sub-languages, each of them specified using OMG's Meta Object Facility (MOF) [19] and referred to as *meta-model* in OMG's terminology. We present DML's structure, followed by an introduction of its core concepts and innovative aspects. The complete DML specification is available as a technical report [20]. A realization of DML in EMF Ecore[1] is available from the DML website.[2]

DML has a modular structure designed to reflect the major aspects relevant for modeling performance and resource management of modern IT systems (Fig. 1). In the following, we describe the purpose of each meta-model.

Resource Landscape: The *resource landscape meta-model* is used to describe the structure and the properties of both the physical and the logical resources of the IT system infrastructure. A common pattern reoccurring in modern IT infrastructures is the nested containment of system entities, e.g., data centers contain racks, racks contain servers, servers typically contain a set of virtual machines (VMs), servers and VMs run an operating system (OS), which may contain a middleware layer, and so on. DML provides constructs to model this hierarchy of nested resources as well as their configuration, e.g., memory, bandwidth, and so on.

Application Architecture: The application architecture of the adapted system is modeled after the principles of component-based software systems. A software component is defined as a unit of composition with explicitly defined provided and required interfaces [21]. For convenience, we also use the term *service* to refer to a signature of a software component's interface.

To describe the performance behavior of a service offered by a Component, the *application architecture* meta-model supports multiple (possibly co-existing) behavior abstractions at different levels of granularity. The behavior descriptions range from a BlackBoxBehavior abstraction (a probabilistic representation of the service response time behavior), over a CoarseGrainedBehavior abstraction (capturing the service behavior as observed from the outside at the component boundaries, e.g., frequencies of external service calls and amount of consumed resources), to a FineGrainedBehavior abstraction (capturing the service's internal control flow, covering performance-relevant actions). The novelty of the support for multiple abstraction levels is that one can choose the modeled abstraction level depending on the information that monitoring tools can obtain at run-time, e.g., to what extent component-internal information is available. More importantly, the model is usable in different online performance prediction scenarios with different goals and constraints, ranging from quick performance bounds analysis to detailed system simulation. Multiple different abstraction levels may co-exist for the same service. Thus, the appropriate abstraction level to use can be selected at run-time on a case-by-case basis trading-off between accuracy and speed of the performance prediction. Currently, it is assumed that alternative abstraction levels are consistent with one another and only defer in the granularity at which the system is modeled. In future extensions, we plan to introduce consistency checks between the abstraction levels.

Deployment: To capture the interactions of resource landscape and application architecture, one must model the connection between hardware and software. The *deployment* meta-model associates instances of software component types of the application architecture meta-model with container instances of the resource landscape meta-model.

Usage Profile: To model user interactions with the system (i.e., the usage profile), DML provides a *usage profile* meta-model. A usage profile model contains one or more usage scenarios, which can be seen as a combination of UML use cases and UML activities. A usage scenario describes the workload type (e.g., open or closed workload), the workload intensity (e.g., request arrival rates), and the user behavior, i.e., which services are called and in what sequence.

Adaptation Points and Processes: The *adaptation points* meta-model is an addition to the resource landscape and application architecture meta-models to describe the elements of the resource landscape and the application architecture that can be adapted (i.e., reconfigured) at run-time. Based on this, the *adaptation process meta-model* allows to describe the way the system adapts to changes in its environment. The meta-model consists of three main elements used to describe the adaptation process

1. http://www.eclipse.org/modeling/emf/
2. http://descartes.tools/dml

at three different levels of abstraction.

Actions are the elements on the lowest level of the adaptation process. They capture the execution of an adaptation operation at the model-level. Examples for such actions are to increase/decrease the processing resources of a VM, to clone/remove a VM, to migrate a software component, etc.

Tactics allow to describe more complex adaptations by composing a set of actions to an AdaptationPlan. To describe an adaptation plan, tactics can use control flow elements like Branches and Loops. The purpose of a tactic and its adaptation plan is to achieve an intermediate goal, e.g., increasing the amount of allocated resources. An example of a tactic adding resources could be *"if possible, increase the processing resources of the VM, otherwise, start another VM."*

Strategies are defined on the highest level of abstraction and they capture the logical aspects of the system adaptation process. A strategy defines the Objective that needs to be accomplished and conveys an idea for achieving it. A strategy can be a complex, multi-layered plan using one or more tactics for accomplishing the objective. Which tactic is applied depends on the current state of the system and on the predicted impact of the tactic on the system performance. The advantage here is that in contrast to the individual deterministic adaptation actions of the tactics, the sequence of applied tactics is not pre-defined. This enables the flexibility to react with different tactics in unforeseen situations. For example, a defensive strategy could be *"add as few resources as possible stepwise until response time violations are resolved"*, whereas an aggressive strategy would be *"add a large amount of resources in one step so that response time violations are eliminated, ignoring resource inefficiencies."*

A more complete description of the individual parts of DML can be found in [16], [17], [18]

3 The Descartes Approach

In the 17th century, the French philosopher and mathematician René Descartes described the bidirectional link between the mind and the body as the *dualism principle* ("the mind controls the body, but the body can also influence the mind"), paraphrased by his famous words "cogito, ergo sum". With the approach we propose here, we pursue our vision of self-aware computing systems that have built-in online prediction and self-adaptation capabilities to address the challenges of autonomic performance and resource management [22]. The consensus at the 2015 Dagstuhl Seminar 15041[3] was that *self-aware computing systems* have two main properties [22], [23]. They

- *learn models*, capturing *knowledge* about themselves and their environment (such as their structure, design, state, possible actions, and runtime behavior) on an *ongoing basis*; and
- *reason* using the models (to predict, analyze, consider, or plan), which enables them to act based on their knowledge and reasoning (for example, to explore, explain, report, suggest, self-adapt, or impact their environment)

and do so in accordance with *high-level goals*, which can change.

A major application domain for self-aware computing is the runtime management of modern IT systems. In this context, an IT system is considered *self-aware* [22], [24], [25] if it possesses three properties or can acquire them at runtime - ideally to an increasing degree:

- *Self-reflective* - is aware of its software architecture and execution environment, the hardware infrastructure on which it runs, and its operational goals, such as performance requirements.
- *Self-predictive* - can predict the effects of dynamic changes, such as changing service workloads, and of possible adaptation actions, such as adding or removing resources.
- *Self-adaptive* - proactively adapts as the environment evolves to ensure that it always meets its operational goals.

The three properties in the above definition are obviously not binary, and different systems may satisfy them to a different degree. However, in order to speak of "self-awareness", all three properties must apply to the considered system.

With our approach, we provide a novel, model-based method to design and engineer self-aware systems from the ground up. Such systems have built-in self-reflective and self-predictive capabilities, encapsulated in the form of online system architecture models. The latter are assumed to capture the relevant influences (with respect to the system's operational goals) of the system's software architecture, its configuration, its usage profile, and its execution environment (e.g., physical hardware, virtualization, and middleware). The models are also assumed to explicitly capture the system's operational goals and policies (e.g., QoS requirements, service level agreements, efficiency targets) as well as the system's adaptation space, adaptation strategies and processes. In the context of this article, where our focus is on self-awareness with respect to performance and resource management aspects, we use the Descartes Modeling Language (DML) as a representation for the online system architecture models. In analogy to Descartes' dualism principle, the models are intended to serve as a *mind* to the system (the *body*) with a bidirectional link between the two.

3.1 Concept

The software engineering community and the autonomic computing community both use the notion of a control loop (also called feedback loop) as an essential generic concept to build (self-)adaptive systems [12], [26]. In the software engineering community, the generic

3. http://www.dagstuhl.de/15041

concept of a feedback loop consists of four distinct phases: *Collect*, *Analyze*, *Decide*, and *Act* [12]. These four phases are based on the concept of a control loop used by the autonomic computing community since 2003, which specifies four similar phases MONITOR, ANALYZE, PLAN, and EXECUTE, commonly referred to as MAPE-K control loop [26]. An additional important element of the MAPE-K control loop is the KNOWLEDGE (BASE) which is shared by all other four parts and can be considered as the *mind* of the system.

Our model-based adaptation approach is based on this generic control loop concept and leverages DML's online performance prediction capabilities to implement adaptation processes at the model level. In the following, we explain our refinements to each phase of MAPE-K and describe how we integrate and leverage DML as a basis for self-aware performance and resource management.

3.1.1 Monitor

The adaptation control loop starts by collecting monitoring data from the managed system. This includes mainly performance metrics like service response time, throughput or resource utilization, that can be obtained with monitoring frameworks. In addition to performance metrics, it is important to collect further data from the system environment. For example, in our approach we require workload data in the ANALYZE phase to be able to forecast changes in the workload intensity. Another example is information relevant to the goals of the adaptation, e.g., changed customer constraints like SLAs. Such information is important to ensure that the adapted system is aware of its operational goals.

3.1.2 Analyze

The general purpose of this phase is to analyze the monitored data to detect and anticipate violations of the system's operational goals (e.g., SLA violations, inefficient resource usage). If a problem is detected, the system state is analyzed to identify its causes (e.g., a resource bottleneck) such that suitable adaptation strategies of the subsequent PLAN phase can be triggered. This scenario describes an example of *reactive* system adaptation. To enable *proactive* system adaptation, we must be able to anticipate performance problems before they have actually occurred. In [27], we showed an approach that uses workload classification and forecasting techniques to predict changes in the workload intensity at runtime. In the approach presented here, we apply the predicted workload changes to the usage profile model of the DML instance and use our online performance prediction techniques to analyze the impact on metrics like service response time or resource utilization. This allows to detect emerging system bottlenecks or inefficient resource usage and proactively trigger the search for a solution to the anticipated problem.

3.1.3 Plan

In the PLAN phase, we search for a feasible solution to the problems identified in the ANALYZE phase. Depending on the specific adaptation options supported by the system and the possible solutions to a detected problem, the search process can become complex and comprise several iterations. In general, one can imagine any mechanism, from rule-based approaches (e.g., Amazon AWS auto scaling) to complex algorithms (e.g., using reinforcement learning [28]) and optimization heuristics (e.g., hill-climbing, or evolutionary programming [29]), that can be applied here. However, designing and implementing such mechanisms is a major challenge that is usually addressed using custom low-level solutions tailored for the specific system. Our holistic model-based approach addresses this issue by abstracting from technical and system-specific details and leveraging techniques from model-driven software engineering to reduce the complexity of this phase. With DML, it is possible to specify the adaptation process at the model level, shifting the complexity of system-specific and technical details to the adaptation framework.

The system designer can focus on modeling the adaptation process using the adaptation possibilities provided by the system (adaptation actions). The designer is not required to have extensive experience with the low level technical details of the underlying platforms and mechanisms used to execute the adaptation actions themselves. For example, the adaptation action of "virtual machine migration" is conceptually the same whether considered in the context of a Xen- or VMware-based virtualization platform. The adaptation process designer can work at the model level and can easily apply an adaptation process in the context of different underlying platforms provided that they support the required adaptation actions. The abstraction of technical details, as well as the resulting better separation of concerns, fosters the reuse of adaptation logic in different contexts.

We realize the PLAN phase as two main steps that are executed iteratively to find a suitable solution to a predicted problem. In the first step (*Adapt System Model*), we automatically generate a new system configuration at the model level by applying the adaptation strategy selected as part of the ANALYZE phase in order to avoid the problem. A strategy might contain several alternative tactics (e.g., add VM instance, or replicate software component) to choose from. Each tactic has an associated weight. To decide which tactic to apply next, a strategy chooses the tactic that has the highest weight. The weight of a tactic is determined using a weighting function $f : T \times S \to \mathbb{R}$, where T is the set of tactics and S is the set of all possible system states. The idea is that any existing and well-established optimization algorithms or meta-heuristics (like tabu search or simulated annealing) can be used here to determine the weights depending on the current state of the system, possibly also considering its previous states stored in a trace. In the second step

(*Predict Adaptation Impact*), after applying the tactic with the highest weight at the model level, we analyze the adapted model using our online performance prediction techniques presented in Section 4. The prediction results are compared with the previous metric values to analyze the impact of the adaptation, e.g., by comparing response time or resource efficiency metrics. If the applied adaptation was successful, i.e., the detected problem has been solved, we can derive a concrete system adaptation plan that is executed on the system in the following EXECUTE phase. If the problem is not solved, the PLAN phase continues executing, iteratively generating a different system state while taking into account the determined impact of previously executed adaptations.

This loop ends either when a solution to the detected problem has been found or when a given time limit for finding a solution has been reached. The realization of this phase including the automatic modification of the system model and the impact analysis using online performance prediction techniques is implemented as part of our adaptation framework presented in Section 3.2.

3.1.4 Execute

In this phase, we perform the actual adaptation on the real system by replaying the adaptation actions that have been successfully applied at the model level. To adapt the system and bring it into the desired state, we execute the actions of the concrete adaptation plan derived in the PLAN phase using the interfaces provided by the real system (e.g., virtualization platforms or middleware). We note that actions in a tactic may be retried in case of failures. If a tactic fails several times, it would be deactivated in the adaptation model, and the PLAN phase will be repeated in order to find a new solution using alternative tactics.

3.1.5 Model Calibration and Refinement

To make effective adaptation decisions, it is important that the online system architecture model provides up-to-date and accurate information about the system [11]. Uncertainty in the knowledge about the system structure and behavior, or in the runtime measurements, may negatively impact the robustness of the overall approach. Therefore, we have included an extra step in the overall approach to continuously validate and refine the online architecture model. All predictions will be compared to actual measurements as soon as the real system reaches the time corresponding to a predicted future state. The additional control loop triggers a separate model refinement step that calibrates the model such that it reflects the real system behavior within an acceptable margin of error (for more details on the refinement see [30, Chapter 6.5]). By continuously refining and updating the model, we realize the self-reflective and self-predictive properties that are essential for predicting the effect of dynamic changes in the environment and possible adaptation actions.

3.2 Realization

To realize the control loop presented in Section 3.1, we have implemented a framework for model-based system adaptation. The framework takes as input a DML instance as described in Section 2. The framework interprets the adaptation process described as part of the DML instance and applies the modeled changes on the application architecture, resource landscape, and deployment models. The output is an adapted model of the system with a configuration that solves the problem that triggered the adaptation process. In contrast to performance models which can be extracted automatically, the creation of adaptation processes requires the effort of a system designer [17].

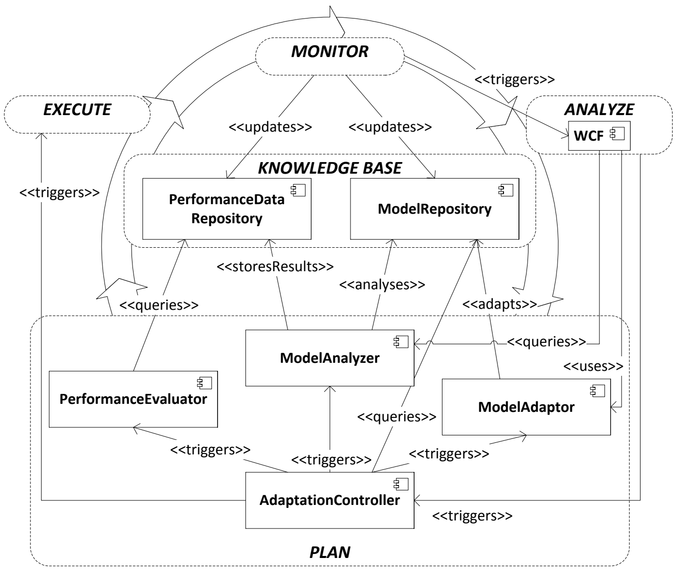


Fig. 2. Architectural overview of the implemented components constituting our adaptation framework.

Figure 2 depicts the different software components of our framework. The `WCF` component forecasts future workload intensities based on the monitored workload data. We use the `ModelAdaptor` to apply the forecasts to the usage profile model of the DML instance stored in the `ModelRepository`. Next, we query the `ModelAnalyzer` employing its online performance prediction techniques to evaluate the impact of the forecast workload changes on the system performance. In case the changes have a negative impact causing a performance problem, `WCF` triggers the `AdaptationController` to start an adaptation process.

The `AdaptationController` is the core component of our framework. Once initialized, the `AdaptationController` listens for events from the ANALYZE phase, indicating current or forecast future violations of the system's operational goals (e.g., SLA violations or inefficient resource usage). When a problem has been signaled through an event from the ANALYZE phase, the `AdaptationController` launches the adaptation process to find a model configuration that resolves the identified issues. To start this process, the AdaptationController queries the ModelRepository for the strategy that has been designed to be triggered upon observation of the respective event. Next, it selects the tactic with the currently highest weight from the set of tactics assigned to the strategy. Then, the AdaptationController passes the selected tactic to the ModelAdaptor, which adapts the DML instance in the ModelRepository according to the tactic's adaptation plan. The ModelAdaptor employs the adapter classes generated by EMF and the EMF reflection API to interpret the adaptation plan and execute the defined adaptation actions. Furthermore, the ModelAdaptor employs EMF's Change Model API to track the changes performed on the DML instance. In this way, we can ensure the transaction semantics of tactics (see Section 2) and roll back the model to its previous state if the application of a tactic fails.

The ModelRepository provides access to the current DML instance. It also uses the adapter classes generated by EMF to query and inspect the model instances. The ModelRepository also uses EMFs Change Model API to maintain a model history by tracing all changes that have been applied to the models. Thereby, we are able to switch back to a previous model state or use the trace to generate an adaptation plan that can be replayed on the real system in the EXECUTE phase.

After the ModelAdaptor has applied the selected tactic, the AdaptationController calls the ModelAnalyzer to analyze the changed model instance. Depending on the performed adaptations, the ModelAnalyzer generates a performance query and triggers the online performance prediction process described in Section 4. The results of the model analysis are stored in the PerformanceDataRepository.

The PerformanceDataRepository is implemented as an instance of a meta-model specified in EMF Ecore. It manages the performance-relevant metrics measured in the system or predicted by the ModelAnalyzer. After model analysis, the AdaptationController decides if the detected problem has been solved or if further adaptations of the model with different tactics are required. If further adaptations are necessary, the AdaptationController triggers the PerformanceEvaluator to evaluate the impact of the tactic. The PerformanceEvaluator queries the PerformanceDataRepository for the affected metrics M of the model state s at the current and previous time points i and $i-1$, respectively. The quantification of the achieved impact of tactic t is based on user-defined weighting functions. The user can specify a weight w_m for each metric $m \in M$ to define the relative importance of the metric with respect to the overall adaptation goal. For example, we can calculate the weight of a tactic as

$$w_t = \sum_{m \in M} w_m \cdot (v_{m,s_i} - v_{m,s_{i-1}})$$

with v_{m,s_i} representing the value of metric m obtained for model state s_i. The PerformanceEvaluator assigns this value w_t as new weight to the executed tactic. In the next iteration of the PLAN phase, the AdaptationController chooses the tactic with the highest weight. This iterative process continues until the evaluation of the current model configuration reveals that the problem detected in the ANALYZE phase has been solved, or until the AdaptationController decides to stop the process. If a valid solution was found, the AdaptationController triggers the adaptation of the real system. To adapt the system, we use the changes recorded by the ModelAdaptor to generate an adaptation plan, which we then execute on the system using the adaptation interfaces it provides.

In collaboration with the ModelAdaptor, the AdaptationController interprets and executes the adaptation process specified with DML. Thus, the implementation of these components defines the semantics of the specified adaptation process model instance. For example, when modeling the scale out of a VM, there is a difference if the used adaptation point refers to a Container or a ContainerTemplate. The meaning of the former is to scale the instances of the Container on the parent model element, whereas new instances of a scaled ContainerTemplate can be deployed on any model element referred to by the template. If there are multiple possible targets for the new container when scaling a ContainerTemplate, the process must decide where to allocate the new instance by using heuristics, e.g., picking a random target or the least/most utilized first. This example shows the inherent complexity of adaptation processes. By using a model-based adaptation process, we cannot eliminate this challenge. However, we reduce complexity by using models to abstract from technical details and define adaptation processes at the model level, thus shifting some of the complexity into our adaptation framework. Thereby, we ease the work of system developers and administrators when designing and implementing adaptation processes.

All components—with exception of the workload classification and forecasting component WCF—are implemented in Java as OSGi components, using the Eclipse Equinox OSGi infrastructure. For code generation and editing of the model instances, we use the Eclipse Modeling Framework (EMF). The WCF component is designed as a stand-alone Java application that is independent of the rest of the framework and is used in the ANALYZE phase for workload forecasting. All components are available as stand-alone Java applications or Eclipse plug-ins[4].

4. http://www.descartes.tools/

4 Online Performance Prediction

A central element of our model-based performance and resource management approach is the use of online system architecture models described with the Descartes Modeling Language (DML) to predict the impact of adaptation actions by adapting and analyzing the online models on-the-fly. Since this analysis has to be performed at run-time, where only limited time and restricted monitoring data may be available, our approach supports tailored online performance prediction techniques. In contrast to black-box approaches to online performance prediction, such as [9], [31], the techniques we propose allow us to vary and analyze the impact of multiple degrees of freedom such as system configurations, service compositions, and resource allocations. The online performance prediction techniques we present in the following are able to answer *performance queries* that can be derived from questions such as: What performance would a new service deployed on the infrastructure exhibit? How much resources should be allocated to it? How should the workloads of the services be partitioned among the available resources? If any service experiences a load spike or a change of its workload, how would this affect the system performance? Which parts of the system architecture would require additional resources? What would be the effect of migrating a service or an application component? However, when answering such queries, there is a trade-off between prediction accuracy and time-to-result. There are situations where the prediction results need to be available in a short period of time such that the system can be adapted *before* SLAs are violated. Accurate, fine-grained performance prediction comes at the cost of a higher prediction overhead and longer time-to-result, whereas coarse-grained performance predictions allow speeding up the prediction process. The challenge is to balance the trade-off between prediction accuracy and prediction speed.

4.1 Performance Queries

The approach we present here allows to conduct performance predictions on-the-fly where each prediction is tailored to answering a given performance query. A performance query describes which specific performance metrics of which entities are of interest. For instance, when triggering a performance prediction, the 90th percentile response time of a specific service or the utilization of a specific resource such as the database server may be of interest. Furthermore, a performance query specifies a desired trade-off weight between prediction accuracy and prediction speed. In situations where the prediction speed is critical, the prediction process provides the option to speed up the prediction. However, this comes at the cost of reduced prediction accuracy. Note that it is not our intention to specify real-time constraints for the prediction process but rather to provide a way to customize the process to the concrete online scenario. Therefore, we use trade-off weights ranked in an ordinal scale and defined as an ordered set $W = \{w_k : k = 1...K\}$. Weight w_K has the semantics of highest prediction accuracy compared to the other $w \in W$. Weight w_1 has the semantics of fastest prediction speed compared to the other $w \in W$. A demanded trade-off weight is given by a selected $dw \in W$. To sum up, the prediction process is tailored to the required performance metrics as well as to a given trade-off weight between prediction accuracy and speed.

```
SELECT r.utilization, s.avgResponseTime
CONSTRAINED AS FAST
FOR RESOURCE 'AppServerCPU1' AS r,
    SERVICE 'newOrder' AS s
USING dmlConnector@modelLocation;
```

Listing 1. Constrained Query

Our notion of a performance query, formalized in [32], provides a declarative interface to performance prediction techniques to simplify the process of using architecture-level performance models for performance analysis. The query language [5] provides a notation to express the required performance metrics for prediction as well as the goals and constraints of a specific prediction scenario. An illustrative example of such a performance query is depicted in Listing 1. It queries for the average response time of a 'newOrder' service as well as the average utilization of an application server CPU, and requests a 'FastResponse' prediction (equivalent to w_K).

4.2 Tailored Prediction Process

Figure 3 provides an overview of the prediction process showing the individual steps and their inputs and outputs. The prediction process is triggered by a performance query referring to a DML instance specified in the `USING` clause of the query.

The model composition step marks those parts of the DML instance relevant for answering the query. These markings are kept in a *composition mark model* which serves as input for the next step. For instance, if a service is described with multiple service behavior descriptions such as a fine-grained behavior, a coarse-grained behavior, and a black-box description, the model composition step chooses a behavior description that provides adequate means to predict the requested performance metrics considering the specified trade-off weight.

The next step traverses the DML instance starting with the usage scenarios specified as part of the usage profile model. First, it resolves the probabilistic characterizations of the parameter dependencies. Parameter dependencies in DML describe the performance-relevant behavior of a service's implementation depending on input parameters passed directly or indirectly upon service invocation (e.g., the resource demand of order

5. Implementation at http://www.descartes.tools/dql

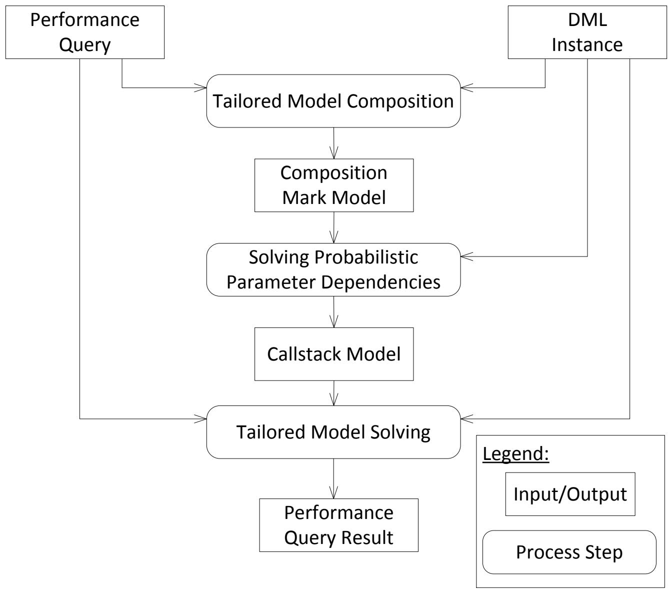


Fig. 3. Overview of online prediction process.

requests in a web shop may depend on the number of items in the shopping cart). A probabilistic description of the distributions of such parameter dependencies is extracted from empirical monitoring data collected at runtime. Second, it parameterizes (e.g., resource demands, branching probabilities) the performance model on-the-fly using the monitoring data stored in the KNOWLEDGE (BASE). The output is a call graph together with the corresponding model parameter values, denoted as *callstack model*. The call graph determines how the performance model has to be traversed for the performance prediction.

The next step is the tailored model solving, i.e., it predicts the requested metrics considering the given trade-off weight. It uses existing model solving techniques based on established stochastic modeling formalisms. The model solving decides which concrete model solving technique to apply. In addition, model solving techniques also come with their own configuration options and can also be tailored to the performance query.

4.3 Tailored Model Composition

As described in Section 2, DML supports up to three different behavior descriptions $behavior(s)$ of different granularity for a service s, namely fine-grained (f), coarse-grained (c), and black-box behavior (b) descriptions, i.e, $behavior(s) \subseteq \{f, c, b\}$. This ambiguity is resolved by the tailored model composition step. In the following, we describe how the selection of an appropriate service behavior description is done. The selection is stored in a mark model, formalized as function $mark(s) : s \mapsto \{f, c, b\}$, that maps a service to a service behavior level. The performance query (dw, DM) is given as a set of requested metrics DM and a trade-off specification $dw = w_k \in W$. The selection consists of three steps: *initialization*, *weighting*, and *truncation*.

Initialization. Using the ordering $f < c < b$, for each service, the service behavior marking is set to the most detailed available service behavior. For each service s, we set $mark(s) = \min(behavior(s))$.

As helper function, for a service s we define

$$\mathit{calledServices}(s) := \{s' | s' \text{ is service triggered by } s\},$$

where the (transitively) called services are derived via a depth first traversal of the marked service behaviors, starting from the $clientBehavior$ modeled as part of the usage profile, navigating through the modeled service calls. Based on $calledServices(s)$, we also define

$$\mathit{resources}(s) := \bigcup_{\substack{s' \in \mathit{called-} \\ \mathit{Services(s)}}} \{r | r \text{ is resource stressed by } s'\},$$

i.e, the set of passive and active resources that is stressed when service s and its subsequent services are processed.

Weighting. Algorithm 1 determines the target service behavior level for service s, depending on the performance query (dw, DM). Mapping function $dw = w_k \mapsto ((k-1) \text{ div } \lceil K/3 \rceil)$ maps $dw = w_k \in W$ to the levels fine-grained ($= 0$), coarse-grained ($= 1$) and black-box ($= 2$). If the target level is fine-grained, we can directly proceed with the *truncation* step. If the target level is coarse-grained, for all services where both fine-grained and coarse-grained behavior descriptions are available, the coarse-grained behavior is marked. If the target level is black box, an available black-box behavior is marked unless the service call path starting with s does not contain a service for which a metric is requested in *DM* and does not stress a resource for which a metric is requested in *DM*.

Truncation. Service calls of the $clientBehavior$ that do not involve services or resources that affect a demanded metric can be truncated in the performance prediction. Such calls do not contribute to the result of the performance query and can therefore be omitted. Let S be the set of directly called services of the $clientBehavior$ as it is modeled in the usage profile model. We partition S in two sets

$$S' := \{s \in S | \forall s' \in \mathit{calledServices}(s) : \\ \neg \exists dm_i = (l_i, m_i, a_i) \in \mathit{DM} : s' = l_i\}$$

and $S_{\mathit{DM}} := S \setminus S'$. Thus, S' denotes the set of services called by the $clientBehavior$ whose call paths do not include a requested metric and S_{DM} denotes the set of services called by the $clientBehavior$ whose call paths *do* include a demanded metric. Each $s' \in S'$ where expression

$$\forall r' \in \mathit{resources}(s') : \neg \exists dm_i = (l_i, m_i, a_i) \in \mathit{DM} : r' = l_i \\ \wedge \forall s_{\mathit{DM}} \in S_{\mathit{DM}} : \mathit{resources}(s') \cap \mathit{resources}(s_{\mathit{DM}}) = \emptyset$$

holds, can then be truncated since requested metrics are not affected.

Algorithm 1: Model Composition: Weighting

```
$Weighting(s : \text{Service}, (dw, DM) : \text{Query})$
begin
    $behaviorLevel \leftarrow (k-1)$ div $\lceil K/3 \rceil$
    if $behaviorLevel = 0$ then
        return
    else if $behaviorLevel = 1$ then
        if $\{f, c\} \subseteq behavior(s)$ then $mark(s) \leftarrow c$
    else
        if $b \in behavior(s)$ then
            if $\forall s' \in calledServices(s)$ :
                    $\neg\exists dm_i = (l_i, m_i, a_i) \in DM : s' = l_i$
                and
                    $\forall r' \in resources(s)$ :
                    $\neg\exists dm_i = (l_i, m_i, a_i) \in DM : r' = l_i$
            then
                $mark(s) \leftarrow b$
            end
        else if $\{f, c\} = behavior(s)$ then
            $mark(s) \leftarrow c$
        end
    end
end
```

4.4 Tailored Model Solving

The tailored model solving step gets the truncated performance model and uses existing model solving techniques based on established stochastic modeling formalisms to predict the requested performance metrics. In the following, we first provide a brief overview of the solving techniques, and then describe the tailoring mechanism itself that derives parametrization of analysis approaches, based on the query and its constraints.

4.4.1 Solving Techniques

As analytical solving techniques, we apply asymptotic bounds analysis [33], and make use of the analytical solver tool LQNS [34], [35]. As simulation technique, we use the SimQPN simulation engine [36].

DML2BoundsAnalysis. A bounds analysis based on Little's Law and Utilization Law can quickly provide asymptotic bounds for the average throughput and the average response time, but this comes at the cost of lower accuracy. However, the results can still be accurate enough to make quick decisions when approximate performance results are sufficient [33].

DML2LQNS. The LQNS solver implements several analytical solving techniques such as Mean Value Analysis (MVA) [33] and combines the advantages of other existing analytical solvers, namely SRVN [37] and MOL [38]. Given that LQNS is a solver for Layered Queueing Networks (LNQs), a transformation from DML to LQN has to be provided. For details of the transformation, we refer the reader to [39].

DML2SimQPN. As a combination of Queueing Networks and Colored Generalized Stochastic Petri Nets, Queueing Petri Nets (QPNs) can model hardware contention and scheduling strategies as well as software contention, simultaneous resource possession, synchronization, blocking, and asynchronous processing. SimQPN is an established simulator for QPNs [36] that provides fine-grained options to control what type and amount of data is logged during the simulation run. For the transformation from DML to QPNs, we refer the reader to [40].

4.4.2 Tailoring

In this section, the focus is on tailoring the solving techniques to the given performance query. On the one hand, it is decided which of the available model solving techniques is appropriate for the performance query. On the other hand, each model solving technique itself comes with its own configuration options and thus itself can be tailored to the query.

Given a performance query (dw, DM), the target *behaviorLevel* is given by the mapping function $dw = w_i \mapsto ((i-1)$ div $\lceil K/3 \rceil)$, that maps trade-off specification dw to the levels fine-grained $(= 0)$, coarse-grained $(= 1)$ or black-box $(= 2)$. Figure 4 illustrates the tailoring mechanism.

- Bounds analysis can be used to quickly derive asymptotic bounds for average throughput and average response time of the *clientBehavior*, but comes at the cost of potentially reduced prediction accuracy. Furthermore, in case of an open workload, resource utilization can be predicted.
- LQNS is also limited to mean value predictions, and may suffer from the assumption of exponentially distributed service times and inter-arrival times, in particular. While approximations of service times are considered acceptable for mean-value analysis (e.g., [31]), approximations of the inter-arrival time distribution in case of an open workload may easily lead to considerable prediction errors. Furthermore, the support for analyzing blocking behavior at passive resources is limited [39].
- SimQPN can provide the most detailed performance predictions, e.g., predictions of response time distributions.

While, e.g., the bounds analysis does not have additional degrees-of-freedom, the transformation to QPN and solving with SimQPN provides multiple configuration options. These configuration options can be used to further tailor the solution to a given query (dw, DM). SimQPN provides fine-grained options to control what type and amount of data is logged during the simulation run. The more data is logged, the longer the simulation run takes. Depending on the desired metrics, each place in the resulting QPNs can be annotated with a so-called stats-level, e.g., differentiating if only throughput statistics are to be collected or also utilization measurements, or even residence time statistics. For the details of the stats-level settings, we refer to [30].

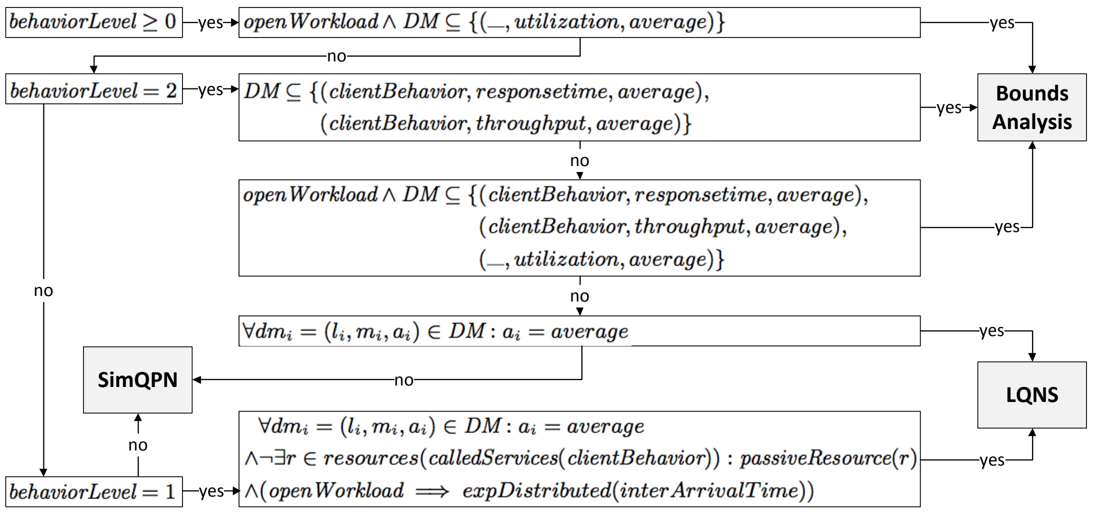

$behaviorLevel \geq 0$
yes
$openWorkload \wedge DM \subseteq \{(\_, utilization, average)\}$
yes
no
$behaviorLevel = 2$
yes
$DM \subseteq \{(clientBehavior, responsetime, average), (clientBehavior, throughput, average)\}$
yes
Bounds Analysis
no
$openWorkload \wedge DM \subseteq \{(clientBehavior, responsetime, average), (clientBehavior, throughput, average), (\_, utilization, average)\}$
yes
no
no
$\forall dm_i = (l_i, m_i, a_i) \in DM : a_i = average$
yes
SimQPN
no
LQNS
no
$\forall dm_i = (l_i, m_i, a_i) \in DM : a_i = average \wedge \neg \exists r \in resources(calledServices(clientBehavior)) : passiveResource(r) \wedge (openWorkload \implies expDistributed(interArrivalTime))$
yes
$behaviorLevel = 1$
yes

Fig. 4. Tailored Model Solving

Furthermore, SimQPN's simulation stopping criteria are configured to reflect trade-off specification dw. SimQPN uses non-overlapping batch means [41] for estimating the variance of mean residence times, and stops when a certain confidence level is reached. The significance level and the desired width of the confidence interval are configurable. The significance level is set to $\alpha = 0.05\%$, the width of the confidence interval is set relative to the mean value, i.e., as relative precision of the estimate. We vary the relative precision depending on the trade-off specification dw. The higher the precision, the higher the prediction accuracy, but also the longer the prediction run. For $dw = w_{\top}$ (trade-off weight with highest accuracy), the relative precision is set to 5%. For $dw = w_{\perp}$ (trade-off weight with fastest prediction speed), the relative precision is set to 30%. In capacity planning, prediction errors of up to 30% are considered acceptable [31]. Accordingly, a trade-off specification $dw = w_i \in W$ then maps to a relative precision of $((K - i) * (30 - 5)/K) + 5$ percent.

The output of the model solving step is the prediction of the requested metrics *DM*. The prediction itself is thus tailored to the given performance query in order to provide the requested metrics in accordance with the given specification of how to trade-off between prediction accuracy and time-to-result.

5 Case Studies

In this section, we show the results of two industrial case studies [6] that represent the first end-to-end evaluation of the approach. In the first case study, we apply our online performance prediction technique to the CRM system of a leading Software-as-a-Service cloud provider. For the second case study, we implemented our model-based approach for self-aware performance and resource management for the BlueYonder system, a cloud service for preditictive analytics and big data analysis.

5.1 Software-as-a-Service CRM System

The Customer Relationship Management (CRM) application used in this case study is part of a component-based large-scale multi-tenant platform, i.e., the platform is shared among multiple customers (tenants). In this context, we evaluate the suitability of our modeling abstractions and the capabilities of our performance prediction techniques. We investigate scenarios of the core CRM application that consists of an application tier and a database tier. As part of the evaluation, we build an architecture-level performance model, conduct predictions varying the workload intensity and service input parameters, and compare the predictions with measurements of the real system.

5.1.1 System Setup

We conducted experiments in a resource environment as depicted in Figure 5. The two application server instances as well as the database instance and the Storage Area Network (SAN) were of the same type and configuration as the production system. The database server has 48 CPU cores and runs an Oracle Database, the application server instances each have 24 CPU cores where Jetty[7] is running as application server. The experimental environment can be understood as a *vertical slice* of the production environment, i.e., instead of $\approx$ 20 application servers the experimental environment provides 2 application servers, and instead of eight database servers the experimental environment provides one database server. Given that the application tier is *stateless* and the database tier is configured in a way that a tenant is served always by the same database node, the experiment setup can provide measurement results that are representative for production environments. The CRM system is deployed on the application servers and the database server as in production. The database contains the anonymized state from a production instance.

6. http://descartes.tools/dml/examples

7. http://www.eclipse.org/jetty/

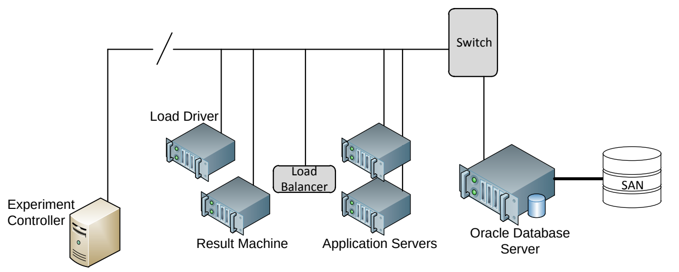


Fig. 5. Resource Environment

5.1.2 Architecture-Level Performance Model.

Figure 6 depicts a high-level overview of the structure of the architecture-level performance model describing the CRM application. The system model shows a load balancer that distributes incoming requests to one of the two CRM instances that themselves need a database instance. A CRM instance refers to a composite component that in turn consists of component instances of, e.g., the component providing the integration service. In this case study we model the component services with coarse-grained behavior descriptions. Given that the CRM services highly depend on meta-data that describes the tenants' customizations, a fine-grained service control flow is hard to model. For the model parameter estima-

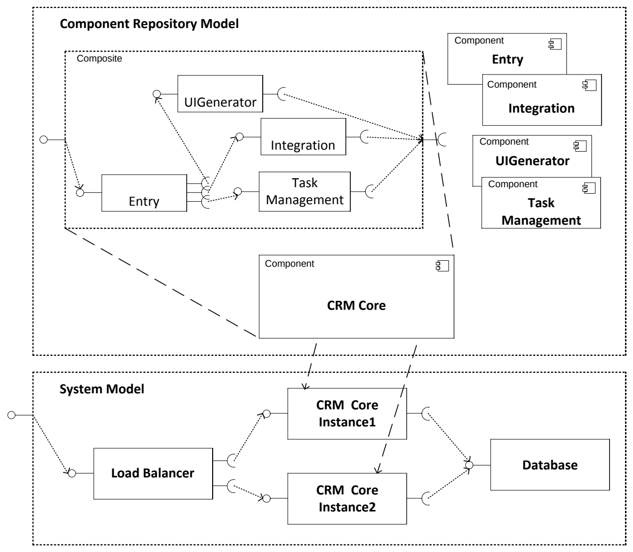


Fig. 6. Application Architecture Model

tion, we use monitoring statistics as they are captured in the production system. Thus, we do not inject additional monitoring overhead compared to the production system. The resource demands were estimated during performance tests executed with a steady state time of 900 seconds and a warm-up time of 600 seconds. CPU resource demands are approximated using response time measurements that are obtained in a low load scenario, i.e., in a scenario where both the application server CPU and the database server CPU load is below 20%. For each request, we measure the CPU time on the application

```
SELECT MEAN(app1.utilization, app2.utilization),
       dbs.utilization
CONSTRAINED AS ACCURATE
FOR RESOURCE 'ApplicationServer1' AS app1,
    RESOURCE 'ApplicationServer2' AS app2,
    RESOURCE 'DBServer' AS dbs
USING connector@location
```

Listing 2. Query1: Utilization Query

servers, the CPU time on the database server, and the I/O delay as observed at the database.

5.1.3 Results

In this case study, we investigate the prediction accuracy and time-to-result for different performance queries using the online performance prediction approach in Section 4. We issue various performance queries in the context of business scenario 'task management'. In an open workload, a sales agent logs-in and manages several activity lists. There are activity lists directly assigned to the sales agent (*MyActivities*), lists assigned to the sales agent's team (*MyTeamsOpenActivities* and *MyTeamsClosedActivities*), and a list of all activities (*AllActivities*). The load driver script of the task management workload ensures that the usage profile is executed for different sales agents.

We parameterize the architecture-level performance model under low load conditions ($\approx 20\%$ CPU utilization), and then conduct predictions for medium load conditions ($\approx 40\%$), high load conditions ($\approx 60\%$), and very high load conditions ($\approx 80\%$), comparing the results with steady-state measurements on the real system.

Query1. With the first performance query shown in Listing 2, we ask for the utilization of the application servers and database servers. Figure 7 shows the measured and predicted server CPU utilization for the different load levels. The utilization of the DB server varies from 20% to 80%, the application tier is only little utilized. The utilization predictions fit the measurements very well.

TABLE 1
Efficiency of Performance Predictions for Query1

Time to result for Query1	Load level low	medium	high	very high
Analysis Time BA [ms]	11	11	11	11
Sim. Time SimQPN [ms]	396	449	461	482

Table 1 shows the analysis and simulation times of the performance query for the different load scenarios. Each prediction has been repeated 30 times to obtain an average execution time for each prediction. The prediction process selects bounds analysis as solving method. For a comparison we also triggered the utilization predictions with SimQPN. We used Method of Welch [41] to determine the warm-up period and the steady-state run length for SimQPN. However, as shown in Table 1, the time-to-result would have been significantly longer, i.e., ≈400ms instead of 11ms. Note that we did not use LQNS, since it does not support multi-server queues with processor sharing as scheduling discipline [34].

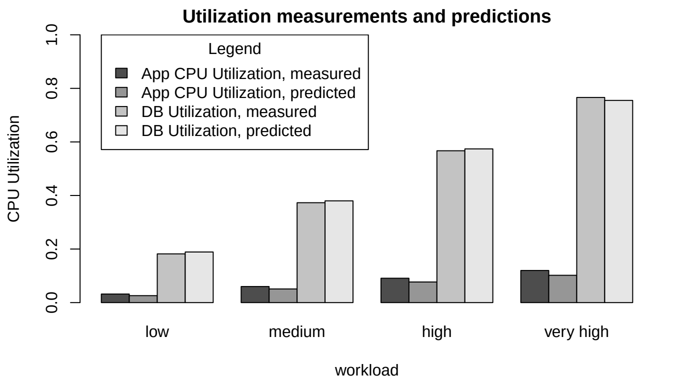


Fig. 7. Measurements, Prediction Results and Prediction Errors for Query1

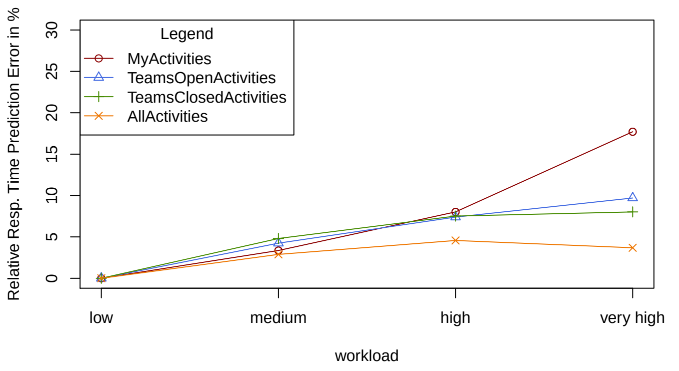


Fig. 8. Measurements and Prediction Results for Query2

Query2. The performance query shown in Listing 2 asks for the average response times of four task management services. The observed average response times vary between 200 and 2000 milliseconds, with service *AllActivities* having the slowest response time, and *MyActivities* the fastest response time. In Figure 8, the relative error of the average response time predictions is shown. It shows that the relative error is below 20% across all load levels. The error is computed relative to the measurements, e.g., a measurement of 100ms and a prediction of 90ms would result in a relative error of 10%. In the low load scenario, the prediction error is zero since the model is calibrated and adjusted with the measurements of this scenario.

TABLE 2
Efficiency of Performance Predictions for Query2

Time to result for Query2	Load level low	medium	high	very high
Sim. Time SimQPN [s]	5.9	6.7	7.6	8.4

Table 2 shows the simulation times of the performance query for the different load scenarios. The prediction process selects SimQPN as solving method. The simulation times are significantly longer than for the first performance query, and grows with the load level.

```
SELECT
    myac.avgResponseTime, toac.avgResponseTime,
    tcac.avgResponseTime, aac.avgResponseTime,
    MEAN(app1.utilization, app2.utilization),
    dbs.utilization
CONSTRAINED AS ACCURATE
FOR RESOURCE 'ApplicationServer1' AS app1,
    RESOURCE 'ApplicationServer2' AS app2,
    RESOURCE 'DBServer' AS dbs,
    SERVICE 'MyActivities' AS myac,
    SERVICE 'TeamsOpenActivities' AS toac,
    SERVICE 'TeamsClosedActivities' AS tcac,
    SERVICE 'AllActivities' AS aac
USING connector@location;
```

Listing 3. Query2: Average Response Time Query

```
SELECT
    PERCENTILE(myac.responseTime)[percentile="90"],
    PERCENTILE(toac.responseTime)[percentile="90"],
    PERCENTILE(tcac.responseTime)[percentile="90"],
    PERCENTILE(aac.responseTime)[percentile="90"],
    MEAN(app1.utilization, app2.utilization),
    dbs.utilization
CONSTRAINED AS ACCURATE
FOR RESOURCE 'ApplicationServer1' AS app1,
    RESOURCE 'ApplicationServer2' AS app2,
    RESOURCE 'DBServer' AS dbs,
    SERVICE 'MyActivities' AS myac,
    SERVICE 'TeamsOpenActivities' AS toac,
    SERVICE 'TeamsClosedActivities' AS tcac,
    SERVICE 'AllActivities' AS aac
USING connector@location;
```

Listing 4. Query3: Percentile Response Time Query

Query3. With the third performance query shown in Listing 2, we ask for the 90th percentile response times of the four task management services. The predictions of the percentiles fit the measurements with an error of below 20% as depicted in Figure 9.

TABLE 3
Efficiency of Performance Predictions for Query3

Time to result for Query3	Load level low	medium	high	very high
Sim. Time SimQPN [s]	54.3	65.3	89.8	114.2

Table 3 shows the simulation times of the performance query for the different load scenarios. The prediction process selects SimQPN as solving method. The simulation times are significantly longer than for the other two performance queries, since the logging data collected during the simulation includes samples of the response time distributions.

5.1.4 Discussion

The SaaS provider case study demonstrates the prediction capabilities of our approach in the context of a real-life enterprise software system. The achieved accuracy

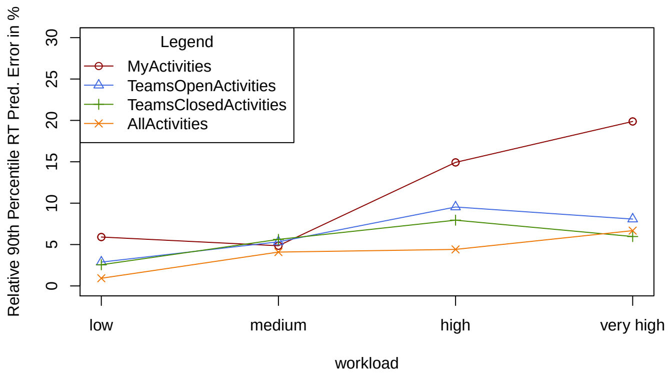


Fig. 9. Prediction Errors for Query3

for the database server and application server utilization predictions is within 5% error. For the service response times, the relative prediction error is within 20%. This applies both to average service response times as well as to the 90th percentile response times. Note that we considered further scenarios beyond ‘task management’ (see [30]), however, in this article, the focus is on prediction accuracy and time-to-result of performance predictions. We showed that the tailored performance prediction process successfully reduces the performance overhead. While the predictions for Query1 are faster than 1s, the time-to-result for Query2 is within 5-10s while for Query3 the prediction lasts up to two minutes.

For SaaS providers, a reliable performance and resource management requires the ability to answer questions concerning capacity planning, admission control, SLA management, and energy management. Performance predictions help to answer such questions. Note that in order to find a target system configuration at the model level, typically not only one performance query is issued but several performance queries assessing the degrees-of-freedom of the system configuration. In this context, differences in the time-to-result of performance predictions can add up to several minutes, and thus a tailoring of the performance prediction process is necessary.

5.2 Blue Yonder

In this section, we present the results of a case study conducted in cooperation with our industrial partner Blue Yonder GmbH & Co. KG. Blue Yonder is a leading service provider in the field of predictive analytics and big data. The company offers enterprise software services that are based on forecasts of, e.g., sales, costs, churn rates, etc. Blue Yonder employs machine learning techniques to obtain accurate forecasts based on historical data provided by their customers. Usually supervised machine learning can be applied, consisting of a training step that is used to infer a mathematical model from the available historical data. This model can then be used to calculate forecasts based on a given input data set. Training the model and calculating the forecasts requires a considerable amount of computational resources depending on the amount of customers, their input data, and their service-level agreements (SLAs).

Given the high costs of leasing IT resources, Blue Yonder is interested in saving costs by dynamically adjusting the amount of resources provided to its customers at run-time without violating the customer’s service level agreements (SLA).

5.2.1 Illustrating Example: DML Instance

Figure 10 depicts an example DML instance in a UML-like notation. In the following sections, we will refer to this figure to illustrate the most important concepts of DML. This example is a simplified version of the DML instance of the Blue Yonder software system that we use later in our case study presented in Section 5.2. Note that Figure 10 does not show the adaptation process, which is presented later as part of the case study.

The center of Figure 10 shows the resource landscape of the Blue Yonder system, consisting of a data center with four servers. Distributed over these servers, we see the software components, the services they provide, and how these services are connected. The example also shows three fine-grained behavior descriptions of the services, parameterized with the respective resource demands. Furthermore, Figure 10 also depicts the usage profile of the system as well as the parts of the system that can be adapted at run-time. A more detailed description of Blue Yonder’s system architecture and our experiment environment follows in Section 5.2.

5.2.2 System Architecture

A typical Blue Yonder system consists of three main software component types: the Gateway Server (`GW`), the Prediction Server (`PS`), and a third party component, the database (`DB`), see the example DML instance in Figure 10. The `GW` is the communication endpoint to the Blue Yonder system. Users can invoke a set of different services via HTTP. In the considered sample project, the available services are `train`, `predict`, and `results`. As their names suggest, the `train` service initiates the training step of the supervised learning algorithms, the `predict` service initiates the calculation of the forecasts based on the trained prediction model, and finally the `results` service provides the results to the customer. To train the prediction model, the `train` service accepts historical data. The `GW` receives this data, parses it, and generates a job, which is put into the `GW`’s queue and scheduled for processing. Then, an active `PS` takes the job from the queue, processes it (i.e., trains the prediction model) and stores the results in the database. After training, a user can invoke the `predict` service to calculate a forecast based on the trained prediction model. The user sends the data for which the forecast should be made to the `GW`. The `GW` reads the data and generates one or several jobs—depending on the size of the data—which are scheduled for processing. These jobs are again processed by one or several `PS` and the

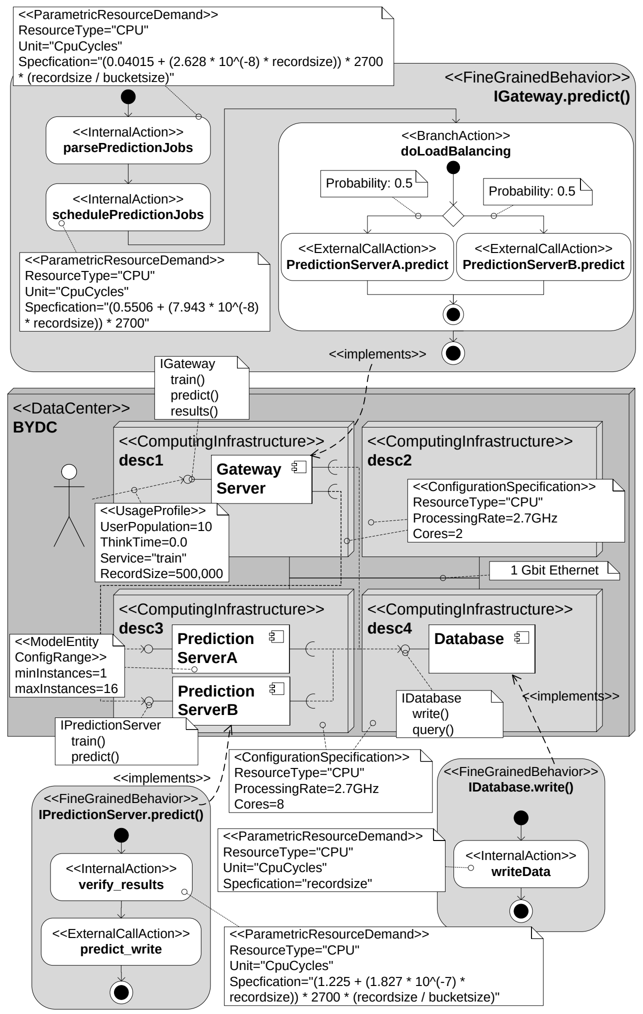


Fig. 10. DML instance describing an exemplary deployment of the Blue Yonder system.

results are stored in the database for retrieval by the user (`results` service). Technically, `GW` and `PS` are independent operating system processes that can be started and stopped on any machine in the resource landscape. The database is a standard MySQL database. Each customer has its own `GW`, `PS`, and `DB` instances, which are deployed in Blue Yonder's resource landscape. The number of component instances and their distribution in the system environment is called *topology*.

In our scenario, the resource landscape consists of heterogeneous hardware, two low-budget dual-core machines (`desc1` and `desc2`), and two high-end quad-core machines with hyper-threading, i.e., eight logical cores (`desc3` and `desc4`). The example topology is depicted in Figure 10.

The types of workload changes that occur in the system environment are characterized by the service type that is used (`train`, `predict`), the number of requests (request arrival rate), the type of request processing (sequential or parallel), and the size of the request (the number of records per request, typically varying between 10,000 and 500,000). To react on changes in the environment (changes in the workloads of existing customers or launching of new customer projects), additional `PS` instances can be started on other machines. Furthermore, `PS` instances can also be migrated between machines at run-time. The challenge of this case study, compared to the previous one, is that our approach is now faced with a heterogeneous hardware environment and with different performance requirements of multiple concurrent customers. For example, upon a workload change of a given customer, the adaptation process has to decide whether to start/stop a prediction server on a low-budget or a high-end machine, taking into account the performance requirements and topology of other customers.

TABLE 4
Measured and predicted average response times and their relative errors for nine parallel `predict` requests for varying mixed data record sizes with six `PS` instances allocated on `desc4`.

Record Sizes	Response Time [sec]		Error
[in 1,000 records]	measured	predicted	[in %]
50 & 100	194.50	175.26	-9.9
100 & 200	366.48	325.16	-11.3
150 & 300	545.05	485.65	-10.9
250 & 500	937.24	780.91	-16.7

5.2.3 *Model Implementation*

For this case study, we created a new DML instance for Blue Yonder's system. An overview of the resource landscape model, the application architecture model, the deployment model, and the usage profile model is depicted in Figure 10.

We evaluated the prediction accuracy for different workload mixes. Table 4 shows the absolute and relative prediction error for the average response times and Table 4 the absolute prediction errors for the CPU utilization on different hardware nodes. The results showed that the prediction error was below 20%.

The adaptation process we use in this case study consists of the following strategies: `FindDeployment`, `ReduceDeployment`, and `ConsolidateDeployment`. The `FindDeployment` strategy launches new `PS` instances until all customer SLAs are fulfilled. It contains two different tactics starting the new `PS` instances on low-budget machines and high-end machines, respectively. `ReduceDeployment` removes unnecessary `PS` instances from machines to save operating costs, e.g., if the workload of a customer has decreased. Finally, `ConsolidateDeployment` migrates `PS` instances between machines with the goal to improve efficiency. A schematic representation of the adaptation process model is depicted in Figure 11.

5.2.4 *Evaluation*

Currently, Blue Yonder uses dedicated resources for each customer to fulfill their respective SLAs. When acquiring

TABLE 5
Measured and predicted average CPU utilizations and their absolute errors for nine parallel `predict` requests for varying mixed data record sizes with six `PS` instances allocated on `desc4`.

Record Sizes	`desc2` [%]			`desc3` [%]			`desc4` [%]		
[in 1,000 records]	meas.	pred.	err.	meas.	pred.	err.	meas.	pred.	err.
50 & 150	9.42	27.6	18.2	17.11	6.0	11.1	51.56	38.9	12.7
100 & 200	10.35	19.4	9.1	16.54	4.0	12.5	49.19	40.8	8.4
150 & 300	10.51	16.2	5.7	16.33	3.3	13.0	47.79	41.4	6.4
250 & 500	10.20	13.7	3.5	16.65	2.8	13.9	44.67	41.9	2.8

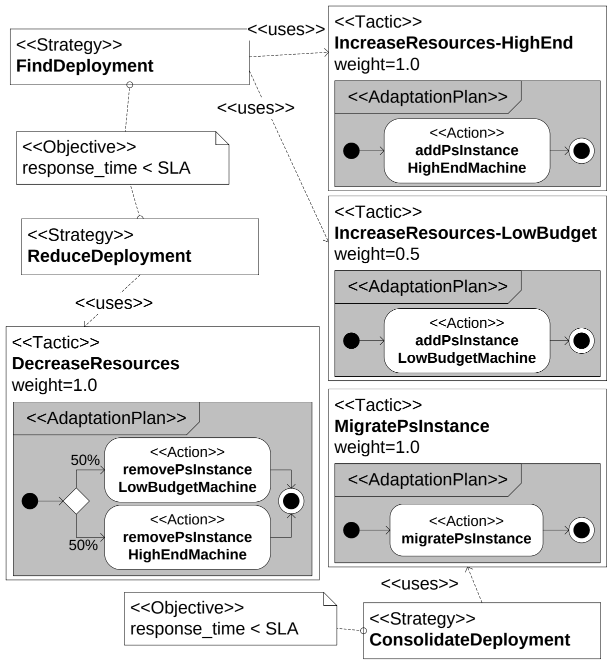


Fig. 11. Schematic representation of the adaptation process model used in Scenario 1.

new customer projects, Blue Yonder normally has to estimate how much resources are required to sustain the workloads of the new customers and ensure adequate performance. This estimation is based on the experience of Blue Yonder and can range from few low-budget desktop machines to hundreds of cores on high-end servers, depending on the amount of data to be analyzed and on the time available for the analysis. Generally a worst-case estimation is made, i.e., the system capacity is dimensioned to support the peak workload intensity. Consequently, Blue Yonder is interested in increasing their efficiency by dynamically sharing computing resources among different customers. As Blue Yonder has detailed information from their customers about when, how many, and which type of requests are expected to arrive, a self-adaptive approach that dynamically adapts the amount of resources according to the actual customer demand appears to be promising. In order to dynamically share the compute resources between customers while still ensuring certain request response times, we created an implementation of the model-based approach described in Section 3 for the Blue Yonder system.

5.2.4.1 Scenario 1: The goal of this scenario is to evaluate the effectiveness of our approach in adapting resource allocations to workload changes such that customer SLAs are fulfilled while considering the heterogeneous nature of the hardware environment.

Our scenario starts with the default topology (one `GW` on `desc2`, one `PS` on `desc4`, one `DB` on `desc3`) and a customer that issues one `predict` request with 500,000 records. The system needs to ensure a maximum request response time of 3,600 seconds (one hour). The default topology is sufficient to process this load without SLA violations. However, when the customer increases the number of requests to 50, the default topology needs to be adapted in order to still provide a response time within one hour for all requests. This event triggers the `FindDeployment` strategy with the objective to find a system configuration that can handle the load without SLA violations. Because the `IncreaseResources-HighEnd` tactic initially has the highest weight, the adaptation framework selects this tactic to add resources to the system. By applying the tactic, another `PS` instance is launched on the high-end machine `desc4`, which improves the response time of the system towards the strategy's objective, but SLAs are still violated. Thus, the adaptation framework continues applying this tactic. However, after the 16th iteration, the response time cannot be further improved. The reason is a constraint by Blue Yonder to avoid significant performance degradation. This constraint limits the number of `PS` instances per machine to two times the available (logical) cores. For example, for the high-end machines where we have four cores with hyper-threading (which is comparable to eight logical cores), 16 `PS` instances can be executed in parallel with negligible resource contention. However, if the algorithm tries to "*overcommit*" the resources of a machine, i.e., deploys more `PS` instances, the performance decreases significantly due to resource contention. Given that after 16 iterations, applying the tactic did not further improve the response time of the system, the adaptation framework revokes the application of this tactic and decreases its weight. However, since the objective has not been achieved yet, the adaptation process continues and applies the `IncreaseResources-LowBudget` tactic in the next iteration. This tactic adds another `PS` instance to the low budget machine `desc1`, which further improves the response time. The adaptation framework keeps applying this tactic for three additional iterations un-

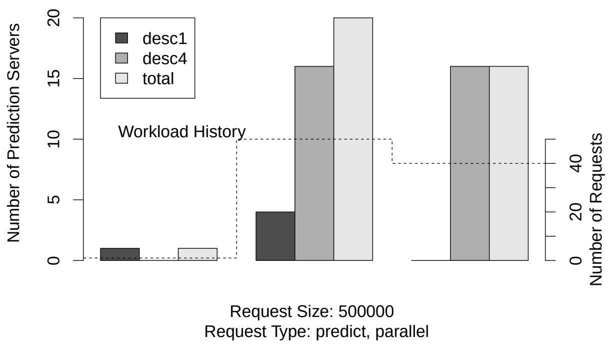


Fig. 12. Adaptation of the system environment to changes in the workload of a customer.

til all SLA violations are resolved. In summary, our modeled adaptation process suggests a deployment of 20 PS instances, four on desc1 and 16 on desc4. As Figure 12 shows, this deployment is sufficient to handle the increased load. This experiment shows that with our approach we are able to model an adaptation process that utilizes the available capacity, but does not exceed it by overcommitting resources.

In the next step, we reduce the workload from 50 to 40 parallel requests, i.e, less resources should be sufficient to maintain the SLA of one hour. Thus, the ReduceDeployment strategy is triggered. The adaptation framework applies the DecreaseResources tactic to reduce the amount of active PS instances. This tactic is repeated until SLAs start being violated, indicating that the process has removed too many PS instances. At that point, the application of the latest iteration of the tactic is revoked, and the adaptation process ends. However, the system configuration after applying this strategy may be further optimized in case PS instances are distributed over several machines that can be consolidated. To this end, the ConsolidateDeployment strategy applies the MigrateResources tactic, migrating the PS instances from the low-budget machine to the high-end server. As a result of our adaptation process, the four PS instances running on the low-budget machine are released. Figure 12 shows that the reduced number of PS instances fulfills the requirements.

5.2.4.2 Scenario 2: In this scenario, we show that our approach is applicable in scenarios where changes of the workload behavior of one customer affect the performance experienced by other customers. The goal is to show that our approach, compared to trigger-based approaches, can trade-off different performance requirements of customers with different priorities. Therefore, we have added two further strategies to our adaptation process model depicted in Figure 13.

The initial Blue Yonder topology in this scenario comprises four PS instances that are deployed on desc1 (see Figure 15). Two of these PS instances belong to customer A, which is a gold customer. The other two PS instances belong to customer B and C, respectively,

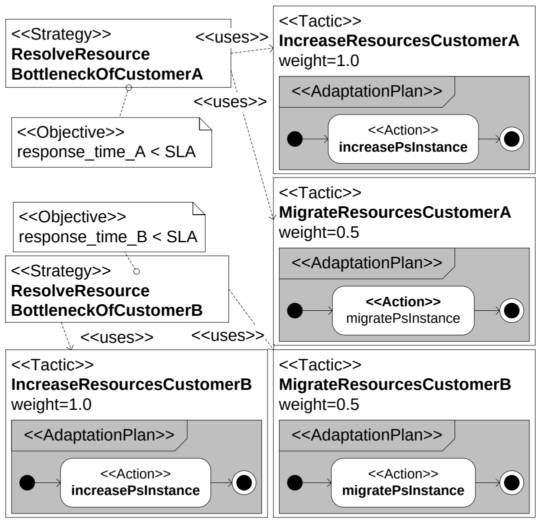


Fig. 13. Schematic representation of the adaptation process model used in Scenario 2.

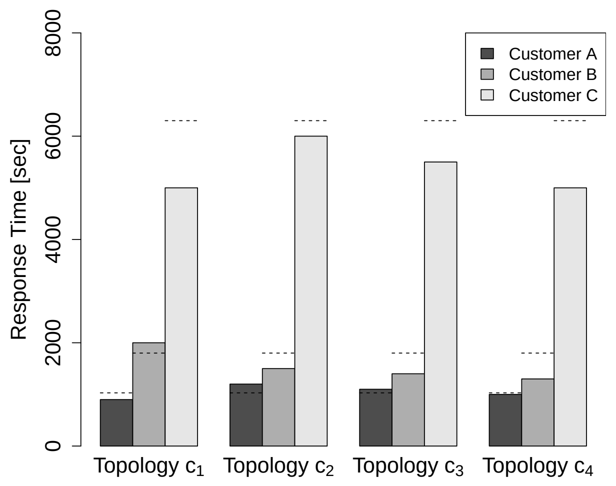


Fig. 14. Response times of the different customers during the adaptation process (SLAs are denoted by the dashed lines).

which are silver customers. A gold customer is a customer with higher priority, i.e., violating its SLAs causes higher penalties. Furthermore, to minimize penalties caused by major response time fluctuations, gold customers have the additional constraint that PS instances of such customers must not be executed on overcommitted machines, i.e., machines already executing two times more PS instances than there are logical cores (see previous scenario).

In this scenario, we assume that we observe an SLA violation for customer B due to a workload increase (see Figure 14), triggering our adaptation process. The process first applies the IncreaseResourcesCustomerB tactic of the ResolveResourceBottleneckOfCustomerB strategy, because this tactic has the highest

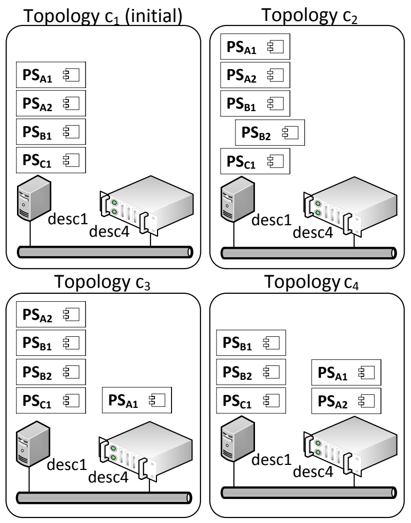


Fig. 15. Details of the different system configurations explored as part of the adaptation process triggered by a workload change of one customer affecting other customers.

weight. Applying this tactic starts another `PS` instance for customer B on `desc1` (topology c_2). However, in the new topology, the SLAs of the gold customer A are now violated due to mutual performance influences. This SLA violation triggers the `ResolveResourceBottleneckOfCustomerA` strategy and the adaptation framework executes the `IncreaseResourcesCustomerA` tactic. However, this tactic cannot be applied because of the constraint that `PS` instances of gold customers must not be executed on overcommitted machines. As a result, the tactic is revoked and its weight is reduced. In the next iteration, the adaptation process applies the `MigrateResourcesCustomerA` tactic to migrate a `PS` instance of customer A to `desc4`. The migration reduces response times, but still does not eliminate the SLA violation of customer A. Nevertheless, the tactic contributed towards the strategy's objective, and thus, the adaptation process continues by migrating the second `PS` instance of customer A to `desc4` (topology c_4). This resolves the problem and the adaptation process completes.

This scenario shows that using our model-based approach, we are able to explicitly consider the mutual performance influences between different customers (adding a new `PS` instance for customer B affected customer A) and model a process that is able to automatically find a way to resolve SLA violations for both customers. In the considered scenario, a conventional trigger-based approach would simply add a `PS` instance. The issue that adding this further instance leads to an SLA violation for customer A would only be detected after the system has been reconfigured. Of course, then a new trigger would start further adaptations to address this issue, however, penalty costs would arise due to the SLA violations that will already have occurred.

5.2.5 Scenario 3

In this scenario, we show that the allocation found by the modeled process provides a valid solution that uses resources efficiently. Like in the previous experiments, we use a workload of five parallel `predict` service requests with 500,000 data records. Moreover, the SLA with the customer states that the request has to be processed within 800 seconds. For such a scenario, our approach suggests to allocate five `PS` instances on `desc4` to maintain the SLA. To evaluate the quality of this solution, we compare measured and predicted average response time and CPU utilization for this scenario with a varying amount of `PS` instances deployed on a high-end machine (see Figure 16). As we can see, four `PS` instances are not enough to meet our deadline of maximum 800 seconds, since four instances need 974 seconds to process the given workload. This figure also shows that further `PS` instances would speed-up the processing of the requests, but would also lead to a higher resource usage.

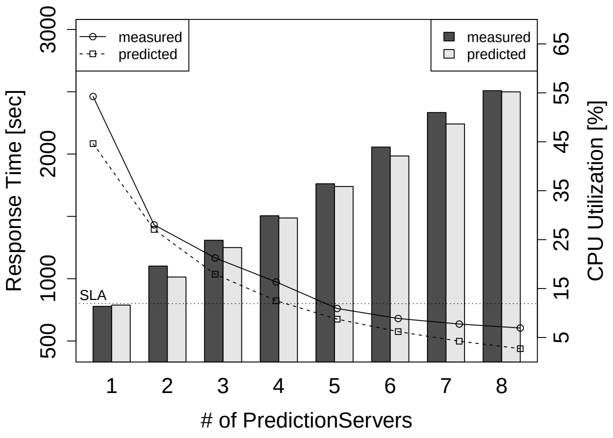


Fig. 16. Comparison of predicted and measured metrics for five parallel `predict` requests with a record size of 500,000. Prediction servers have been started on the high-end machine `desc4`.

5.3 Summary

The presented case studies are examples of how to use our approach to implement model-based performance and resource management at the system architecture level. The first case study based on a Software-as-a-Service CRM cloud application shows that our online performance prediction techniques exploiting the additional capabilities of DML models (e.g., multiple abstraction levels) is able to provide accurate predictions for the resource utilization (within 5% error) as well as mean and percentile response times (within 20% error). At the same time it provides the flexibility to automatically trade-off between prediction accuracy and speed depending on a performance query.

In the second case study, we showed that DML can be used in different scenarios to specify adaptation processes that leverage architectural information for automated adaptation decisions, trading-off different performance requirements of multiple users. In addition, the case study demonstrates the advantages of our approach compared to reactive, trigger-based adaptation approaches. The results showed that, for the considered scenarios, both the reactive and the proactive adaptation approach needed only approximately 45% of resources of the static allocation approach. However, the proactive approach could further reduce SLA violations up to 60% in our scenario. Our experiments revealed that the adaptation overhead is dominated by the number of iterations to find a suitable model configuration multiplied with the time for performance model analysis. To minimize the overhead, our online performance prediction technique automatically tailors the model analysis process to the requirements of the online prediction scenario while taking into account trade-offs between prediction accuracy and time-to-result (see Section 4).

6 Related Work

In recent years, many approaches for managing QoS properties and resource efficiency in the area of autonomic computing and software engineering have been proposed in the literature. In this section, we discuss two areas of related work: approaches for model-driven engineering of self-adaptive software systems based on different types of architectural models, and approaches employing different types of performance models to control and adapt systems such that performance requirements are fulfilled.

6.1 Architecture-Based Model-Driven Engineering of Self-Adaptive Software Systems

As surveyed in [42], many approaches exist for the engineering of self-adaptive software. In this section, we focus on model-driven approaches that employ different types of architectural models to systematically engineer self-adaptive systems.

From a high-level perspective, recent approaches supporting the systematic engineering of self-adaptive systems are EUREMA [43], a model-driven approach to build adaptation engines, FORMS [44], a formal reference model that can be used to describe the crucial aspects of self-adaptive systems relevant for reasoning about early design decisions, or DYNAMICO [45], a reference model that supports the designers of self-adaptive systems in deciding whether the monitoring infrastructure, the system, or the adaptation objectives must be adapted. With their reference models, these approaches describe the properties of adaptive systems at a higher level of abstraction. For example, we could consider using EUREMA or FORMS as part of our future work to specify our adaptation control loop or to formally describe our self-adaptive system.

More specific approaches that use architectural models to build self-adaptive software are, e.g., [13], [14], [15], [46], [47], [48], [49], [50]. Oreizy et al. present an infrastructure for system evolution and system adaptation [13]. In their approach, the software system is described as a dynamic architecture, characterized using graphs of components and connectors, and architectural changes are regarded as graph-rewrite operations. Garlan et al. present a framework called Rainbow that uses an architectural model together with existing architectural analysis techniques to realize self-adaptation [14]. In addition, Cheng and Garlan provide Stitch, a language to express repair strategies in a form that can be analyzed and automated by the Rainbow framework [51]. Another tool that provides a domain-specific modeling language for modeling adaptive systems is Genie [46]. It allows to capture the dynamic aspects of component frameworks like structural variability as well as environment and context variability. To describe system adaptation, Genie uses policies of the form of "on-event-do-actions" applying architectural changes with the goal to improve the current state of the system with respect to given QoS properties. DiVA is an approach for specifying dynamically adaptive software systems based on software product lines [15]. The approach is based on four meta-models providing means to model the system's architecture, environment, and variability, as well as a reasoning model that can be used to describe how different features impact the system QoS. To adapt and optimize component-based applications at run-time, DiVA uses several formalisms such as event-condition-action rules or goal-based optimization rules. Another model-driven engineering approach for the dynamic adaptation of component-based applications is MADAM [47]. It uses an application architecture model based on UML for adaptation reasoning. However, this approach is focused on applications in mobile computing scenarios. Its successor, MUSIC, claims to tackle ubiquitous computing environments [48]. Like our approach, MUSIC also implements a MAPE-K control loop, employing a coarse-grained QoS-aware architecture model as shared knowledge to realize the dynamic and automatic adaptation of applications and services. MUSIC uses utility functions to calculate at run-time the utility of different application configurations. Another model-centric approach is GRAF, a framework that uses graph-based models that are interpreted at run-time to manage system adaptation [49]. In this approach, the runtime model is represented by a graph that describes the adaptable software's state and behavior, captured using a subset of UML activity diagrams.

SASSY [50], [52], [53] is a model-driven framework for optimizing the QoS of orchestration processes in service-oriented applications at run-time. It supports self-adaptation through automatic service provider selection and architectural patterns (e.g., load-balancing, or fail-over). In contrast to our approach, SASSY is based on a coarse-grained application architecture model, expecting the response time distribution of each service as an input. This is a major assumption that is not required in our approach given that DML provides and end-to-end modeling framework supporting also the prediction of service response times. Furthermore, SASSY is focussed on adaptations in the application architecture; the resource landscape (OS, virtualization, middleware) is not modeled explicitly and therefore adaptations at these levels are not supported. Finally, the MAPE-K control loop of SASSY adapts the system to changes in the operating conditions in *a reactive manner*. The support for *proactive* system adaptation based on load forecasts is a major difference in our approach, which is also a major aspect of self-aware systems in general.

In summary, all above-mentioned approaches focus on improving one or more QoS properties of the adapted system by reconfiguring the system at the application level. However, for performance and particularly for resource management, it is important that the resource environment is considered explicitly as part of the adaptation process and the underlying system models. In our approach, adaptation decisions explicitly consider the predicted performance impact of the adaptation on the current state of the system, whereas related approaches do not provide sufficiently fine-grained models to enable such detailed impact analysis. In the following section, we discuss approaches trying to address this problem by explicitly using performance models.

6.2 Model-based Performance and Resource Management in Dynamic Environments

Over the past decade, with the adoption of virtualization and cloud computing, many approaches to online performance and resource management in dynamic environments have been developed in the research community. Such approaches are typically based on control theory feedback loops, machine learning techniques, or stochastic performance models, such as layered queueing networks or stochastic Petri nets. Approaches based on feedback loops and control theory (e.g., [54], [55]) can normally guarantee system stability by capturing the transient system behavior [55]. Machine learning techniques capture the system behavior based on observations at run-time without the need for an a priori analytical model of the system [56], [57]. Performance models are typically used in the context of utility-based optimization techniques. They are embedded within optimization frameworks aiming at optimizing multiple criteria such as different Quality-of-Service (QoS) metrics [4], [58], [59]. However, existing work mostly uses predictive performance models that do not capture the software architecture and configuration explicitly (e.g., [5], [6], [9], [31], [60], [61], [62], [63]). Such models include queueing networks (e.g., [31]), layered queueing networks (e.g., [6]), queueing Petri nets (e.g., [60]), stochastic process algebras [61], statistical regression (e.g., [62]), and Kriging models [9]. The models are typically solved analytically, e.g., based on mean-value analysis, as in [5], or by simulation where analytical solution is not a viable option, as in [63]. In summary, existing model-based techniques for online performance and resource management typically abstract the system as a "black-box" and do not explicitly model the software architecture and execution environment distinguishing performance-relevant behavior at the virtualization level vs. at the level of applications hosted inside the running VMs. However, explicitly considering dynamic changes at these levels and being able to predict their effect at run-time is indispensable for ensuring predictable performance and enforcing SLAs. A survey on model-based approaches to engineering of software systems confirms the importance of considering the dynamic aspects of modern IT systems and services as part of architectural models [64]. However, its results also show that currently, there are no performance modeling approaches that cover the explicit modeling of adaptation strategies or processes at the architecture level.

7 Conclusion

7.1 Summary

In this article, we presented a coherent motivation, design, implementation, and end-to-end evaluation of an approach for self-aware performance and resource management of modern dynamic IT systems and services. The proposed approach is based on model-driven algorithms and architectures and a generic adaptation control loop that leverages the novel features of DML to describe adaptive systems w.r.t. performance and resource efficiency. The benefit of our approach compared to the trigger-based and "black box" modeling approaches is that it considers the individual effects and complex interactions between application workload profiles and resources at the various levels of the execution environment. Furthermore, our model-based approach can be used to describe dynamic aspects like adaptation possibilities and adaptation processes at the model-level in a human-understandable, machine processable, and reusable manner. We evaluated our approach in the context of two different representative case studies. The first case study shows that our online performance prediction technique is able to predict the application performance accurately (within 20% error), and at the same time provides the flexibility to trade-off between prediction accuracy and speed depending on the constraints of a given prediction scenario. The second case study demonstrates that our model-based approach, which leverages the online performance prediction technique, can provide significant efficiency gains of up to 50% without sacrificing performance guarantees, and that it is able to trade-off performance requirements of different customers in heterogeneous hardware environments. Furthermore, we showed how our approach enables proactive system adaptation, reducing the amount of SLA violations by 60% compared to a trigger-based approach. The results of these case studies show that with our holistic model-based approach, one can apply model-driven techniques end-to-end to realize autonomic performance-aware resource management at run-time. Thereby, our approach serves as a proof-of-concept showing how techniques from the field of software engineering and autonomic computing can be combined to systematically design and implement self-aware systems.

7.2 Future Work

As part of our future work, we plan to integrate our meta-models for modeling storage and network infrastructure performance (currently under development) into DML and then conduct further case studies focusing on other resource types. Furthermore, we plan to conduct additional case studies with the model-based adaptation approach to investigate its applicability to a broader range of workloads (e.g., bursty workloads) and adaptations (e.g., short-term reconfigurations). In the long run, we plan to extend DML to support other QoS attributes (e.g., reliability, security) as part of adaptation decisions considering also trade-offs between different attributes.

References

[1] Amazon Web Services, "Amazon Auto Scaling," http://aws.amazon.com/documentation/autoscaling/, 2010.

[2] Microsoft Developer Network (MSDN), "Autoscaling and Windows Azure," http://msdn.microsoft.com/en-us/library/hh680945(v=pandp.50).aspx, 2012, last built: June 7, 2012.

[3] VMware, "Resource management with VMware DRS," http://www.vmware.com/pdf/vmware_drs_wp.pdf, 2006, latest revision: June 5, 2006.

[4] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu, "Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures," in *IEEE Int. Conf. on Distributed Computing Systems*, 2010, pp. 62 –73.

[5] Q. Zhang, L. Cherkasova, and E. Smirni, "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications," in *Proc. of the 4th Int. Conf. on Autonomic Computing*, 2007, p. 27.

[6] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai, "Performance model driven QoS guarantees and optimization in clouds," in *Proc. of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, ser. CLOUD '09, 2009, pp. 15–22.

[7] M. Bennani and D. Menasce, "Resource allocation for autonomic data centers using analytic performance models," in *Proc. of the 2nd Intl. Conf. on Autonomic Computing*, 2005, pp. 229–240.

[8] I. Cunha, J. Almeida, V. Almeida, and M. Santos, "Self-adaptive capacity management for multi-tier virtualized environments," in *IFIP/IEEE Int. Symp. on Integrated Network Management*, 2007, pp. 129–138.

[9] A. Gambi, G. Toffetti, C. Pautasso, and M. Pezze, "Kriging Controllers for Cloud Applications," *IEEE Internet Computing*, vol. 17, no. 4, pp. 40–47, 2013.

[10] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634 – 658, 2010.

[11] G. Blair, N. Bencomo, and R. B. France, "Models@Run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[12] B. H. C. Cheng *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin Heidelberg, 2009, vol. 5525, pp. 1–26.

[13] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, "An architecture-based approach to self-adaptive software," *Intelligent Systems and their Applications, IEEE*, vol. 14, no. 3, pp. 54–62, 1999.

[14] D. Garlan, B. Schmerl, and S.-W. Cheng, "Software architecture-based self-adaptation," in *Autonomic Computing and Networking*. Springer US, 2009, pp. 31–55.

[15] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg, "Models@run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[16] N. Huber, F. Brosig, and S. Kounev, "Modeling Dynamic Virtualized Resource Landscapes," in *ACM SIGSOFT Int. Conf. on the Quality of Software Architectures (QoSA 2012)*, 2012, pp. 81–90.

[17] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev, "Modeling Run-Time Adaptation at the System Architecture Level in Dynamic Service-Oriented Environments," *Service Oriented Computing and Applications Journal (SOCA)*, vol. 8, no. 1, pp. 73–89, 2014.

[18] F. Brosig, N. Huber, and S. Kounev, "Architecture-Level Software Performance Abstractions for Online Performance Prediction," *Elsevier Science of Computer Programming Journal (SciCo)*, vol. 90, Part B, pp. 71–92, 2014.

[19] OMG, "Meta Object Facility (MOF) Version 2.5," 2015. [Online]. Available: http://www.omg.org/spec/MOF/2.5/

[20] S. Kounev, F. Brosig, and N. Huber, "The Descartes Modeling Language," Institut für Informatik, Universität Würzburg, Tech. Rep., 2014. [Online]. Available: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:20-opus-104887

[21] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 2009.

[22] S. Kounev, N. Huber, F. Brosig, and X. Zhu, "A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures," *IEEE Computer*, vol. 49, no. 7, pp. 53–61, 2016.

[23] S. Kounev, X. Zhu, J. O. Kephart, and M. Kwiatkowska, Eds., *Model-driven Algorithms and Architectures for Self-Aware Computing Systems*, ser. Dagstuhl Reports, Dagstuhl, Germany, 2015. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/5038/

[24] S. Kounev, F. Brosig, N. Huber, and R. Reussner, "Towards self-aware performance and resource management in modern service-oriented systems," in *Proceedings of the 7th IEEE International Conference on Services Computing (SCC 2010)*. IEEE Computer Society, July 2010.

[25] S. Kounev, "Engineering of Self-Aware IT Systems and Services: State-of-the-Art and Research Challenges," in *Proceedings of the 8th European Performance Engineering Workshop (EPEW 2011)*, 2011.

[26] J. Kephart and D. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[27] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning," *Concurrency and Computation - Practice and Experience, John Wiley and Sons, Ltd.*, vol. 26, no. 12, pp. 2053–2078, 2014.

[28] P. Padala, A. Holler, L. Lu, X. Zhu, A. Parikh, and M. Yechuri, "Scaling of cloud applications using machine learning," *VMware Technical Journal*, May 2014.

[29] A. Martens, H. Koziolek, S. Becker, and R. Reussner, "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," in *Proc. of the First Joint WOSP/SIPEW Int. Conf. on Performance Engineering*, 2010, pp. 105–116.

[30] F. Brosig, "Architecture-level software performance models for online performance prediction," Ph.D. dissertation, Karlsruhe Institute of Technology (KIT), 2014.

[31] D. A. Menasce and A. F. A. Virgilio, *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[32] F. Gorsler, F. Brosig, and S. Kounev, "Performance Queries for Architecture-Level Performance Models," in *Proc. of the 5th ACM/SPEC Int. Conf. on Performance Engineering (ICPE 2014)*, 2014, pp. 99–110.

[33] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. New York, NY, USA: Wiley-Interscience, 1998.

[34] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz, "Layered Queueing Network Solver (LQNS) soft-

ware package," http://www.sce.carleton.ca/rads/lqns/, 2011, last visit: 2014-03-20.

[35] G. Franks, T. Omari, C. M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Trans. on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2009.

[36] S. Spinner, S. Kounev, and P. Meier, "Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0," in *Int. Conf. on Application and Theory of Petri Nets and Concurrency*, vol. 7347, 2012, pp. 388–397.

[37] C. Woodside, J. Neilson, D. Petriu, and S. Majumdar, "The stochastic rendezvous network model for performance of synchronous client-server-like distributed software," *IEEE Trans. on Computers*, vol. 44, no. 1, pp. 20–34, Jan 1995.

[38] J. Rolia and K. Sevcik, "The method of layers," *IEEE Trans. on Software Engineering*, vol. 21, no. 8, pp. 689–700, Aug. 1995.

[39] H. Koziolek, "Parameter dependencies for reusable performance specifications of software components," Ph.D. dissertation, University of Oldenburg, March 2008.

[40] P. Meier, S. Kounev, and H. Koziolek, "Automated Transformation of Palladio Component Models to Queueing Petri Nets," in *19th IEEE/ACM Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011)*, 2011, pp. 339–348.

[41] A. Law and W. Kelton, *Simulation modeling and analysis*, ser. McGraw-Hill series in industrial engineering and management science. McGraw-Hill, 2000.

[42] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions Autonomous and Adaptive Systems*, vol. 4, no. 2, 2009.

[43] T. Vogel and H. Giese, "Model-Driven Engineering of Self-Adaptive Software with EUREMA," *ACM Transactions Autonomous and Adaptive Systems*, vol. 8, no. 4, pp. 18:1–18:33, 1 2014.

[44] D. Weyns, S. Malek, and J. Andersson, "FORMS: Unifying reference model for formal specification of distributed self-adaptive systems," *ACM Transactions Autonomous and Adaptive Systems*, vol. 7, no. 1, p. 8, 2012.

[45] N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien, and R. Casallas, "DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS, 2013, vol. 7475, pp. 265–293.

[46] N. Bencomo and G. Blair, "Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems," in *Software Engineering for Self-Adaptive Systems*, ser. LNCS. Springer Berlin Heidelberg, 2009, vol. 5525, pp. 183–200.

[47] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav, "A comprehensive solution for application-level adaptation," *Software Practice & Experience*, vol. 39, pp. 385–422, 2009.

[48] S. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G. Papadopoulos, "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *Journal of Systems and Software*, vol. 85, pp. 2840–2859, 2012.

[49] M. Amoui, M. Derakhshanmanesh, J. Ebert, and L. Tahvildari, "Achieving dynamic adaptation via management and interpretation of runtime models," *Journal of Systems and Software*, vol. 85, pp. 2720–2737, 2012.

[50] D. Menasce, H. Gomaa, s. Malek, and J. Sousa, "Sassy: A framework for self-architecting service-oriented systems," *IEEE Software*, vol. 28, no. 6, pp. 78–85, Nov 2011.

[51] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860 – 2875, 2012.

[52] D. A. Menascé, J. P. Sousa, S. Malek, and H. Gomaa, "Qos architectural patterns for self-architecting software systems," in *Proceedings of the 7th International Conference on Autonomic Computing, ICAC*, 2010, pp. 195–204.

[53] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa, "A framework for utility-based service oriented design in SASSY," in *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering*, 2010, pp. 27–36.

[54] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: A control-theoretical approach," *IEEE Trans. on Par. and Distr. Syst.*, vol. 13, no. 1, pp. 80–96, 2002.

[55] J. Almeida, V. Almeida, D. Ardagna, I. Cunha, C. Francalanci, and M. Trubian, "Joint admission control and resource allocation in virtualized servers," *Jour. of Par. and Distr. Comp.*, vol. 70, no. 4, pp. 344 – 362, 2010.

[56] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," in *IEEE Intl. Conf. on Autonomic Computing*, 2006, pp. 65–73.

[57] J. Kephart, H. Chan, R. Das, D. Levine, G. Tesauro, F. Rawson, and C. Lefurgy, "Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Tradeoffs," in *Proc. of the 4th Intl. Conf. on Autonomic Computing* , 2007, p. 24.

[58] A. Verma, P. Ahuja, and A. Neogi, "pMapper: power and migration cost aware application placement in virtualized systems," in *Proceedings of the 9th ACM/IFIP/USENIX Int. Conf. on Middleware*, 2008, pp. 243–264.

[59] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan, "Online Self-Reconfiguration with Performance Guarantee for Energy-Efficient Large-Scale Cloud Computing Data Centers," in *IEEE Intl. Conf. on Services Computing*, 2010, pp. 514–521.

[60] S. Kounev, "Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 7, pp. 486–502, 2006.

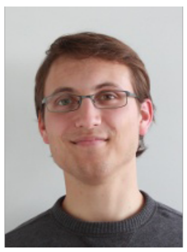
[61] S. Gilmore, V. Haenel, L. Kloul, and M. Maidl, "Choreographing Security and Performance Analysis for Web Services," in *Formal Techniques for Computer Systems and Business Processes*, 2005, pp. 200–214.

[62] E. Eskenazi, A. Fioukov, and D. Hammer, "Performance Prediction for Component Compositions," in *Proc. of the Intl. Symp. on Component-Based Software Engineering*, 2004, pp. 280–293.

[63] G. Jung, K. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "Generating Adaptation Policies for Multi-tier Applications in Consolidated Server Environments," in *Proc. of the 2008 Intl. Conf. on Autonomic Computing*, 2008, pp. 23–32.

[64] M. Becker, M. Luckey, and S. Becker, "Model-driven performance engineering of self-adaptive systems: A survey," in *ACM SIGSOFT Int. Conf. on Quality of Software Architectures*, 2012, pp. 117–122.

Nikolaus Huber is a software architect at Car-Garantie Versicherungs AG and an associate researcher in the Chair of Software Engineering at the University of Würzburg. He received a PhD in computer science from the Karlsruhe Institute of Technology (KIT). His research interests include autonomic and adaptive systems, software architectures, and performance engineering of software systems.



Fabian Brosig is a lead developer at Minodes GmbH and an associate researcher in the Chair of Software Engineering at the University of Würzburg. He received a PhD in computer science from the Karlsruhe Institute of Technology (KIT). His research interests include run-time performance and resource management, performance modeling and analysis, software performance engineering, virtualization and Cloud Computing.

Simon Spinner is a software engineer at IBM and a PhD candidate at the Chair of Software Engineering at the University of Würzburg. He received a master's degree in computer science from the Karlsruhe Institute of Technology (KIT). His research interests include run-time performance and resource management, performance model extraction, performance modeling and analysis, virtualization and Cloud Computing.

Samuel Kounev is a professor and chair of software engineering at the University of Würzburg. His research is focussed on the engineering of dependable and efficient software systems, including software design, modeling and architecture-based analysis; systems benchmarking and experimental analysis; and autonomic and self-aware computing. He received a PhD in computer science from TU Darmstadt. He is a member of ACM, IEEE, and the German Computer Science Society.

Manuel Bähr Manuel Bähr is the Head of Platform Development at Blue Yonder GmbH in Germany. He received his PhD in theoretical physics from the University of Karlsruhe in 2008. His research interests include high performance scalable architectures and self-adaptive systems in the area of predictive analytics.