Kriging-based Self-Adaptive Controllers for the Cloud

Doctoral Dissertation submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Alessio Gambi

under the supervision of

Prof. Mauro Pezzè

July 2013

Dissertation Committee

Prof. Cesare Pautasso	Università della Svizzera Italiana, Switzerland
Prof. Antonio Carzaniga	Università della Svizzera Italiana, Switzerland
Prof. Marin Litoiu	York University, Canada
Prof. Nenad Medvidovic	University of Southern California, USA

Dissertation accepted on 15 July 2013

Prof. Mauro Pezzè
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Antonio Carzaniga
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Alessio Gambi
Lugano, 15 July 2013

To my wife Daniela
who supported me all long the way with passion and patience.

Abstract

We propose Kriging-based self-adaptive controllers to manage the allocation of resources to computing systems that need to provide guarantees on their quality of service at runtime while minimizing running costs.

Service providers need to adjust the configurations of their systems and the resources allocated to them to maintain an acceptable level of service at runtime despite fluctuations in the workload, dynamisms of the environment, and other unexpected events. If unsuitably configured, systems may misbehave, saturate, and violate the service level agreement that are stipulated with end-users, leading to penalties, financial losses and damage to service providers' reputation.

Static system configurations are limited by the strong assumptions on the runtime system behavior and working conditions, and in general lead to either system over-provisioning (i.e., too expensive), or under-provisioning (i.e., too many violations). Fixed adaptation strategies that are commonly based on thresholds and rules, can deal well with simple systems and expected workload fluctuations, but are limited because they require experts for their setup, do not generally adapt, and do not scale well with system complexity. Self-adaptive controllers based on models can deal with predicted and unpredicted working conditions and can adapt. Among them, controllers based on white-box models require the knowledge of systems internal to work properly, but that may be too difficult or even impossible to gather, thus resulting in a limited applicability or in poor accuracy of the models. On the contrary, controllers based on black-box models that are built from monitoring data of running systems are applicable to a wider set of systems.

Among the alternative black-box models, we choose to adopt Kriging models as core elements for model-based self-adaptive controllers. Our choice is motivated by several reasons: (i) Kriging models are accurate in capturing the behavior of running systems; (ii) they are robust against noise and inaccuracy of monitoring data, make predictions in a timely fashion, and can be retrained on-line with negligible overhead; (iii) they are based on a solid theory that extends traditional regression with statistical scaffoldings and that enables them to pair confidence measures with predictions; (iv) they can be used to design effective and efficient proactive controllers that account for uncertainty while planning their control actions.

We apply Kriging-based controllers in the domain of Clouds, in particular, at the infrastructure level. Clouds offer the technical means to dynamically allocate and de-allocate virtual machines thus enabling system *elasticity* that controllers leverage to maintain acceptable system performances while minimizing costs (i.e., the amount of running virtual machines).

Through an experimental validation, we show that Kriging models are accurate, fast and flexible enough to be used inside model-based self-adaptive controllers for the Cloud. The accuracy of Kriging models is comparable to the one of other complex models, but Kriging models are faster to train, easier to manage, and more scalable with system complexity. We compare our proactive controllers based on Kriging models against state-of-the-art solutions and we conclude that our controllers are efficient, effective and more generally applicable than the other considered solutions.

Contents

Figures

Tables

Chapter 1

Introduction

In this thesis, we explore the use of model-based self-adaptive controllers for dynamic resources allocation: Controllers change the allocation of resources to running systems to avoid violations of service level agreements and to minimize the amount of resources used by the controlled systems. In particular, we propose to use Kriging models as core components for engineering such self-adaptive controllers.

Business organizations and enterprises may provide functionalities to customers by means of Web services, playing in this way the role of *service providers*. Customers, also referred as *final users*, are willing to pay for using these Web services but require guarantees on services behavior. For this reason, service providers and final users stipulate contracts, called Service Level Agreements (SLAs) that define what are the expected service behavior and expected service usage. SLAs are commonly expressed in terms of Quality of Service (QoS), i.e., measurable levels of quality attributes, such as availability, reliability, and performance that may be specified over a finite temporal horizon. For example, an SLA may require that a service is always available during working days, that no more than five consecutive service requests fail within two minutes, or that the service will respond to client requests at least 99.5% of the times in each calendar month with an average response time of five seconds [91]. If the service provider fails to adhere to such contract (i.e., violates the SLA) then clients can claim penalties that result in missed revenues, additional costs, and decreased reputation for the service provider.

Commonly, designers configure enterprise systems so that they can provide the required levels of service. For example, with *capacity planning* designers decide upon the allocation of resources to systems to provide predictable levels of performances [2]. Capacity planning relies on principles such as *resource partitioning* and *static resources allocation* that aim to ensure performance isolation and predictable behavior of applications. It consists of identifying cost-effective system configurations to cope with predefined workload scenarios that are expected to occur at runtime. Designers choose the optimal system configuration according to some high level organization policy. For example, a company may want to completely avoid SLA violations, while another one may tolerate some amount of violations. In the former case, designers provision the system to cope with the worst scenario that they expect, otherwise, they provision the system to cope only with a fraction of it.

Following these principles, designers tend to provision applications with new resources to attain acceptable guarantees on the QoS of services, that is, designers *over-provision* the systems. In this configuration, the costs of ownership for the organization, which include for example costs for cooling, for electric supply, and for maintenance, increase. At the same time, resources are *under-utilized* for most of the time. As reported by several surveys, the average load of servers that are managed according to this model is between 5% and 15% [81; 87].

Besides cost-inefficiency, if design-time assumptions, like the workload distribution, the magnitude of its peaks, or its time-of-the-day dynamics, do not hold, static system configurations may become inefficient, systems may saturate, and services may stop behaving as prescribed by their SLAs. To reduce at the same time the risk of violating SLAs and the average costs of running the enterprise systems, providers can resort to *dynamic resource provisioning*.

Nowadays, with the advent of virtualization service providers can outsource the management of basic infrastructure resources to Cloud providers, and can cut infrastructure costs by allocating and deallocating resources to their applications on the fly. More interestingly, service providers can implement dynamic adaptation strategies that automatically react to important changes in the working conditions and modify the system accordingly to avoid violations of SLA. Clouds offer the technical means to dynamically scale applications in order to support a variable number of clients during runtime. They can provision new servers within minutes and remove underutilized servers in seconds.

Problem statement

Unfortunately, Cloud providers do not yet solve the problem of finding the most suitable configuration that systems should implement to avoid SLA violations. This problem presents some non-trivial challenges: Uncontrolled changes inside the Cloud infrastructure, unexpected fluctuations and sudden spikes in the service demands, variety and complexity of the virtualized applications, and heterogeneity of resources inside the Cloud.

Uncontrolled changes in working conditions of the Cloud may affect the availability of physical resources and the end-to-end QoS of applications that they run. For example, events like the deployment of new VMs or the failure of some physical machines may have an impact on the virtualized applications. These difficulties depend on the impossibility of directly monitoring these events from the outside of the Cloud.

Clouds are mainly used to run applications that are usually exposed to a large audience and have variable workload. Workload can change in terms of both intensity and request mix, and this may require different amounts and distribution of resources to application components. For example, the workload may naturally fluctuate considerably as its natural request distribution, but it may suddenly increase over a very short time if the service is under a flash-crowd, for example due to a *slash-dot* effect. These fluctuations may exhaust the available resources and result in service misbehavior and, possibly, in SLA violations.

The kind of systems deployed in Clouds ranges from multi-tier applications, to Web service compositions, content delivery networks, data processing systems, grid middleware and other customized systems [7]. Applications may have complex software architectures and it is not always clear what are the relationships between system configurations, parameter settings, resource allocations, incoming workload, and the end-to-end application behavior. In the same way, it is not clear what are the effects of acting on any of these concerns at rutime.

Cloud middlewares abstract heterogenous physical resources into virtual machines that from the final user point of view seem to be homogeneous. However, because of this inner heterogeneity different instances of the same virtual machine may show different behavior when run in the same Cloud. This implies that the applications running different virtual machines under a predefined load may provide different end-to-end QoS depending on the actual mapping of virtual machines to physical servers that is not under control of the service provider.

Controllers for dynamic resource provisioning

Service providers resort to controllers that continuously monitor the behavior of controlled systems and adapt the configuration of controlled systems to provide a consistent level of service.

To be effective, these controllers should fulfill the strong requirements that derive from the non trivial challenges that we identified. For example, controllers should perceive unexpected and important changes in the application behavior, and plan actions to reduce the effects of these changes on the application QoS. They should predict expected fluctuations of requests to proactively allocate the right amount of resources, and they should understand if sudden peaks that demand for a quick increase of allocated resources are harming the system. Controllers should also capture the global behavior of applications no matter their internal complexity or number of software tiers. They should learn from the actual perceived situation, update their control strategy, and consequently adapt to the working context.

Different solutions are currently available. Controllers that adopt fixed strategies, such as rule-based controllers, may be good if there are no unexpected changes, emergent behaviors and the virtual machines behave consistently during the system lifetime. They become less effective when the actual working conditions or system behavior differ too much from design time expectations.

Controllers that are derived from control theory are fast, robust, have tunable parameters and are grounded on a solid mathematical background [66]. They are appealing but difficult to apply because the system identification process at their core is generally too expensive. Furthermore, they require strong assumptions about the linearity of the system behavior to prove properties like system stability and absence of oscillations.

Controllers that exploit machine learning techniques, such as reinforcement learning, are general and do not require any *a-priori* knowledge. They are proven to produce optimal control policies that are able to deal with unexpected cases and emergent behaviors, but unfortunately they come at the very high costs of exploration.

Model-based controllers exploit models of the system to support their analysis and planning activities. Some controllers use white-box models, like queuing network, while other use black-box and surrogate models.

Controllers based on white-box models are limited by the availability of knowledge about the system internals needed to build accurate models, and by the time required to obtain reliable prediction out of them. In the domain of Clouds for example, there is no visibility inside the system and the parameters of models are difficult to estimate correctly, therefore models can provide only approximate descriptions.

Controllers based on black-box and surrogate models build models of the system behavior by combining input/output data using regression techniques. They are generally applicable because require no knowledge about system internals, and usually can work with data passively collected from the running systems. Furthermore, black-box models can be adapted or re-trained, thus enabling the implementation of model-based controllers that self-adapt.

In this thesis, we target self-adaptive model-based controllers that exploit black-box and surrogate models to dynamically control the allocation of resources to applications running in Clouds. Despite its application to this specific domain, our solution does not rely on any peculiarity of Clouds, has a general nature, and therefore it can be applied easily in other contexts.

There are a lot of possible models that can be used, but a trade off between accuracy of representation, costs to manage the model, and time to obtain results out of it is evident: Very accurate models need a lot of data and a lot of time to process them, while simplistic linear models are very fast to train but provide poor predictions. For example, neural networks have the potential to be very effective, but are difficult to setup, have high training costs, and are difficult to interpret once ready. Bayesian trees instead are simpler to build and to interpret but require a lengthy training procedure. Simpler models, like traditional linear regression models, work in less time but at the cost of very rough approximations. Ideally, a model should provide good levels of accuracy, with acceptable speed, and at reasonable costs in a way to support controller's activities without harming its reactiveness.

In this thesis, we propose to use Kriging models as core element of self-adaptive model-based controllers to implement an effective solution that can suitably balance the trade off between accuracy, cost and time of on-line control of complex systems. Kriging models, also called Gaussian process regressors, approximate target functions by means of a *spatial* correlation between training samples. They extend traditional linear regression with a statistical framework that allows them to predict the value of the target function in un-sampled locations together with a confidence measure.

We choose Kriging models because they are easy to set up, do not require any prior knowledge on system internals, but can accommodate it when available, and because we can use their statistical scaffoldings to implement robust controllers. Compared to other black-box models, they are either more accurate, faster to train and to query, and less expensive to manage.

Kriging-based controllers

We design self-adaptive Kriging-based controllers that exploit Kriging models in different ways: They analyze monitoring data to identify risky situations that may lead to SLA violations or inefficient use of resources, and they predict the system behavior in different workload scenarios and alternative system configurations. For each service level objective, controllers build a Kriging model that represents the objective metric, for instance the response time or the system throughput, as a function of system configuration, for instance the number and types of virtual machines, and a representation of the workload intensity and mix [103].

Kriging models pair predictions with confidence measures, and we leverage this feature to implement robust control policies that weigh the importance of model predictions using their confidence measure, and take informed decisions about the best control strategy to implement. When the confidence of predictions is too low, controllers discard them and may resort to alternative strategies or heuristics. For example, controllers can query other models or take the system to the closest, yet known, system configuration.

Initially, we train Kriging models with data collected by measuring the system behavior at staging time, and then in production we let controllers continuously update them. By means of model adaptation, controllers adapt themselves to the changing system behavior. As controllers collect more samples, the accuracy of models improve, while their uncertainty decreases, and the time to build the model increases. To avoid the collection of unmanageable sets of samples, many of which do not provide additional information to the model, the controller filters out old samples belonging to the same configuration.

In this thesis, we make the following major contributions:

1) The identification of Kriging models as suitable core components of self-adaptive model-based controllers for dynamic resource management.
2) The design of controllers that exploit Kriging models and implement efficient, effective, and robust model predictive control. The idea behind these controllers is to not only to use Kriging models to predict the immediate system behavior given the actual monitoring data and trigger system re-adaptation, but also to simulate the effects of control actions potentially implementable on the system state while planning the most suitable control strategy.

The proposed approach aims to be general. However, in this thesis, we refer to the domain of applications running in Cloud infrastructure under the Infrastructure as a Service (IaaS) paradigm [109]. We choose this domain because it is significative in the context of software services, is representative of the problem of SLA protection, and encompasses all the complexities that stem from self-adaptive model-based control of complex systems. Cloud IaaS is also increasingly attracting interest from both industry and academia [34; 25; 41].

During the evaluation, we consider only *feasible* SLAs that pertain to performance and costs. This choice is motivated by the fact that Clouds come with the mirage that systems can scale endlessly, but in practice, either service providers have limited budget or Cloud infrastructures have finite resources. This results in finite size system configurations that can sustain only limited workloads under specific SLA conditions. In other words, during the evaluation, we consider only workloads that can be sustained by at least one of the possible resource allocations. In the SLA, we consider performance and costs qualities because they are perceived by service providers and final users as the most important: Costs are primary concerns of commercial companies, in a sense they are also the main driver of the entire Cloud movement. Performance are what greatly impacts on the quality of service as it is experienced by the final users [63].

In the evaluation, we made two more working assumptions. First, we assume that controlled applications can correctly scale without any direct interaction with the controllers. The thesis focuses on the control of elastic applications, but not on their design. In practice, controllers interact with the Cloud platforms to start or stop the virtual machines, therefore we assume that virtual machines are able to automatically join (or leave) the running applications without causing any hard failure. Second, we assume that virtual machines start and stop in *near*-real time, and consistently across the deployments: It may take up to minutes to have a new instance ready, but there are no strong deviations from this behavior between different deployments. With this assumption, we clarify that controllers can act on the applications as fast as their actuators: If the actuator takes too much time to perform the control actions compared to the application dynamics then the effects of control appear too late, the controller account for long-term predictors, or the SLA is quite tolerant.

1.1 Research Questions

In this thesis, we tackle the problem of design efficient and effective self-adaptive controllers that guarantee consistent QoS while efficiently use the resources leased from Cloud infrastructures. We formalize our main research hypothesis as follows:

> **Research Hypothesis**: *Kriging models can be used as the core element of efficient and effective self-adaptive model-based controllers.*

In other terms, the thesis proposes the use of Kriging models to build efficient controllers for the Cloud, and investigates the effectiveness of Kriging models in the design and engineering of model-based controllers for the Cloud.

For validating our main research hypothesis, we divide it in two parts: First, we investigate how accurately Kriging models can capture the behavior of elastic applications running in the Cloud. Second, we determine to what extend Kriging models can be used to build efficient and effective autonomic controllers. Following these observations, we refine the main research hypothesis in four research questions.

In the first part, we investigate the accuracy of Kriging models to capture the behavior of elastic applications in relation to the complexity of elastic applications and the availability of data describing their behavior. We address this issue by means of the first research question:

RQ-1: How accurately do Kriging models capture the behavior of elastic applications run by Cloud infrastructure?

Proving that Kriging models accurately describe elastic applications is a necessary but not sufficient condition to prove that they can be used inside self-adaptive controllers. In fact, the model accuracy is fundamental for model-based control, as controllers leverage models to predict the potential effects of their control strategies during planning, and inaccurate predictions may lead to ineffective, or even damaging, control strategies. However, beside accuracy controllers require that models have additional properties: They should be easy to set up, to build, and to maintain; they should be robust when data are noisy; and, they should be flexible and adapt to changes of application behavior.

In the second part, we address the issue of the suitability of Kriging models to build autonomic controllers, by investigating the next research questions:

RQ-2: Are Kriging models robust with respect to monitoring data noise and precision?
RQ-3: Are Kriging models fast enough to fit the self-adaptive control loop?
RQ-4: Can Kriging models adapt to the behavior of running applications by using the data passively monitored from them?

Research question RQ-2 aims to assure that the accuracy of Kriging models remains sufficiently high to support effective analysis and provide good predictors of applications behavior even when used in real, noisy environments. Research question RQ-3 aims to verify that the predictions are readily available to controllers just when they need them, and that models can be retrained on-line without causing any delay in the control loop. Research question RQ-4 aims to prove that Kriging models have the necessary flexibility to support controllers self-adaptation.

By answering all these research questions, we can prove that Kriging models are indeed suitable core elements to build efficient model-based self-adaptive controllers.

1.2 Scientific Contributions

This thesis advances the state of the art of self-adaptive model-based controllers and controllers for the Cloud by making the following major scientific contributions:

Suitability of Kriging models for control The first contribution is the suitability of Kriging models in the context of model-based self-adaptive controllers. We show that Kriging models are accurate, robust in real situations, timely in predicting the behavior of applications, and can be trained on-line to implement controllers self-adaptation.

Kriging-based self-adaptive controllers The second contribution is the design of efficient and effective controllers that we apply in the context of Cloud computing. Our controllers leverage Kriging models to implement robust control policies that account for model inaccuracies by identifying them upfront and by remedying with alternative modeling choices. Controllers use Kriging models also to simulate the evolution of system state in the future and provide even more accurate predictions.

1.3 Structure of the Dissertation

The rest of this dissertation is organized as follows:

Chapter 2 presents the background concepts of the work. It describes main concepts belonging to Cloud computing, Autonomic computing and self-adaptive systems applied in Clouds, and Kriging modeling.

Chapter 3 discusses state-of-the-art solutions that employ controllers for the Cloud, use surrogate models for performance predictions, and other control solutions that exploits Kriging models.

Chapter 4 describes the solution that we propose in this thesis, i.e., Kriging-based self-adaptive controllers, and its application in the context of Cloud computing.

It motivates the use of self-adaptive model-based controllers, discusses the rationales behind the choice of Kriging models, and presents details on the design and implementation of these controllers.

Chapter 5 describes the evaluation process to validate our solution. The chapter shows the suitability of Kriging models for modeling applications in the Cloud and the performance of Kriging-based controllers. It critically discusses these results and compares them against related approaches.

Chapter 6 concludes the work. It summarizes our methodology, lists the achieved scientific contributions, discusses the limitations of our solution, and illustrates the future research that can spring off our work.

Appendices contain additional material and further details about the solution and the evaluation.

Chapter 2

Background

This chapter introduces the basic concepts over which the thesis develops: Cloud computing, Autonomic computing, self-adaptive controllers for the Cloud, and Kriging models.

2.1 Cloud Computing

Clouds provide resources, like computing power, storage space, networking, applications and data as part of large data centers, and offer them on demand as services [8]. Customers access them remotely through the Internet under a *pay-as-you-go* billing model, and they are charged proportionally to the amount of resources that they consume.

From an historical viewpoint, Cloud computing represents the last step of a trend started several years ago with distributed systems, Grids, and Utility computing [7]. This trend moved towards the delivery of services, mainly through the Internet, in a way that resembles the delivery of traditional services by utility companies, such as the one for water and electric power supplies. In this case customers are charged on the amount of resources used and do not participate to infrastructure setup or management.

Taking a different viewpoint, one can see Clouds also as an evolution of traditional Web hosting services. Nowadays, business organizations and enterprises outsource their infrastructures, platforms and applications to Cloud providers to reduce the costs of owning, managing and running IT infrastructures. Furthermore with Clouds, organizations can exploit a set of resources that is unconstrained from their specific computing infrastructures.

Clouds are characterized by several aspects: i) The ubiquitous access to resources and electronic services: Everything is on-line and available from everywhere. ii) The opaque management of data and physical resources: Consumers are not able, by design, to locate and directly access the *physical* systems running their applications.

iii) The on-demand, self-service nature of service provisioning: Consumers can request more or less services, whenever they need, and receive them almost with no waiting time. iv) The usage-based economic model, commonly implemented as pay-as-you-go: Customers pay only for what they actually consume.

Clouds are organized with a strong separation of concerns. Subjects that provide the applications logic, and subjects that are in charge of carrying out the IT management are agnostic of one other. In this way, service providers can focus on its primary businesses, while Cloud providers takes care of the other duties. For example, if providers want to focus on application development, Clouds can take care of system infrastructure and runtime environments. If providers want to focus on both applications and platform, then Clouds can take care of just resource provisioning. Moreover, Cloud infrastructures allow customers to adjust at runtime their service demand: For example, customers can change the amount of resources allocated to their systems, can deploy new systems and can un-deploy them later on.

Many of distinguishing features offered by Cloud providers are implemented with state-of-the-art technologies and methods that derive from Service Oriented computing [80], Grid computing [12], and system level virtualization [9]. Service Oriented computing has the biggest impact on the management processes relates to the Cloud, while virtualization has the biggest impact resource-wise [58]. As a matter of fact, virtualization is the game changing technology that provides isolated execution environments, generally in the form of virtual machines (VM), and the tools for their dynamic allocation, deallocation and management, e.g., checkpoint, replication, cloning, and migration.

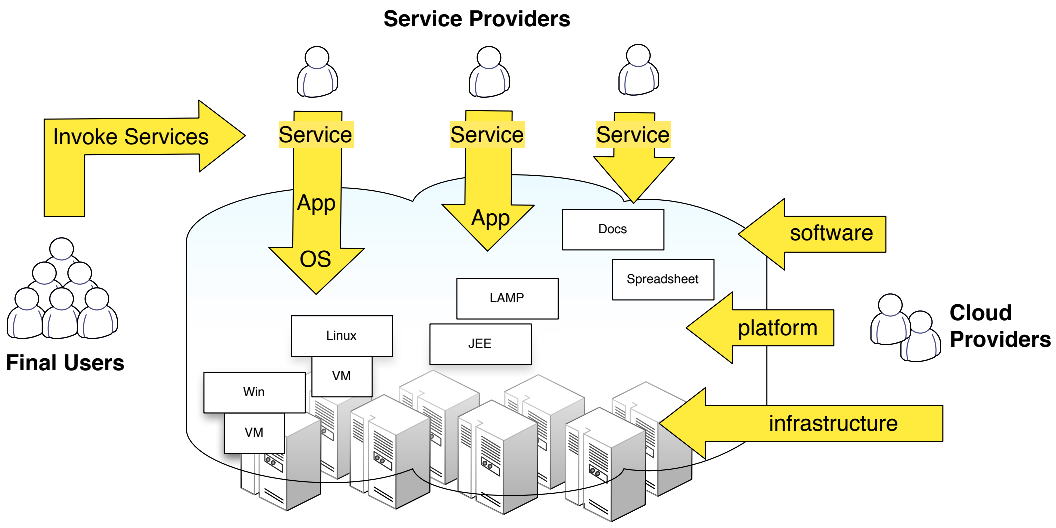

Figure 2.1. Overview of Cloud computing

Generally, a Cloud is composed of three layers: The *infrastructure* is the lowest layer and delivers basic storage, networking and computational capabilities. The *platform* is the middle layer. It builds on top of the infrastructure and provides services in an integrated development or runtime environments. Finally, the *application* is the top layer and offers to final users complete, desktop-like applications as services [64].

Figure 2.1 exemplifies the internal organization of a generic Cloud: Final users access the electronic services as before. Cloud providers manages the “internals” of the Cloud at the different levels. Service providers interact with the Cloud by either managing virtual machines (left), deploying entire applications in predefined environments (center), or composing third party electronic services (right). In these settings, service providers act as final users of Clouds. They buy resources and other services from the Cloud, and use them to provide added value services to their own final users.

Cloud computing defined the three service delivery paradigms:

Software as a Service (SaaS) where clients access complete software systems remotely and use them as they were on local machines. In this paradigm clients have no control over the application nor the infrastructure. Everything is controlled by Cloud providers.

Platform as a Service (PaaS) where clients provide the application logic and have the control over its configuration. In this paradigm, Clouds retain the control over the system infrastructure and the middleware components upon which client applications are built. Usually, Clouds provide “sand boxed” environments and clients logic cannot implement critical system level calls.

Infrastructure as a Service (IaaS) where clients provide the whole software that implements the services and Clouds control the system level resources. In this paradigm clients temporarily acquire remote and isolated virtual servers over which they have full control.

This thesis develops its contributions in the context of Cloud IaaS. We target platforms that offer infrastructure level resources as services, and run virtual machines on behalf of service providers. Service providers control the lifecycle of virtual machines, while Clouds implement management actions like virtual machines placement and physical resources allocation. In essence, IaaS enables service providers to add or remove virtual servers on the fly, and dynamically scale their systems. Service provides use this feature to implement *elastic* applications that can be adapted to dynamically changing working conditions.

Commercial IaaS platforms, like Amazon Ec2[1], RightScale[2], Scalr[3], and many others, come with basic tools to automate the scaling of customer applications. These tools share the same threshold-based basic pattern: Service providers select a subset of the

[1]`http://aws.amazon.com/ec2/`
[2]`http://www.rightscale.com/`
[3]`http://scalr.net/`

application variables monitored by the Cloud platform, specify a set of conditions over their values, and specify how to react when those conditions are met. For example, service provider can monitor the average CPU usage of virtual machines, set a threshold on its value, add a new virtual machine whenever the average CPU usage passes the threshold. In this systems, Cloud providers take care of monitoring the system variables, checking all the conditions, and triggering the specified control actions.

Service providers manually specify these rules and have no tools other than their personal experience to check the suitability of their choice. Also commercial platforms have no tools to support the discovery, update and management of the rules: If the system behave differently from the service providers expectations service providers have to disable, modify and re-enable all the rules.

Static threshold- and rule- based strategies are intuitive and easy to understand, and they may be the right choice for cases when service providers can make strong assumptions on their systems or applications have simple software architectures. However, as the situation becomes more complex, for example, in case of highly dynamic workload, more complex software architectures and unexpected events, fixed control strategies may not be cost-effective.

These situations require solutions that react to known, predictable and recurrent events, that predict the behavior of complex applications, and that *proactively* react to unexpected or potential risky situations. An ideal solution should be able to learn from its past experience and update its control strategy in spite of that, possibly with little or non human intervention, as fostered by the autonomic computing community.

2.2 Autonomic computing and self-adaptive controllers

Autonomic computing is about the idea of systems that can automatically and autonomously manage themselves according to high level objectives defined by system administrators [38].

The central element in the implementation of autonomic computing is the *autonomic element*, also called autonomic controller. Autonomic elements work in a closed loop with the systems they control: They sense the environment and the main variables of the controlled system, and then apply control actions to modify it. In between sensing and acting, controllers implement activities that logically form a loop[4], usually called the MAPE-K loop [40]. In the MAPE-K loop, controllers monitor data (M), analyze (A) them to plan (P) a suitable control strategy, and execute (E) the strategy by implementing the control actions on the system. In all of these activities, autonomic controllers rely on some knowledge (K) about the controlled systems and environment [53].

[4]The activities of autonomic controllers are not necessary executed in sequence, even if this is the most common pattern.

This thesis focuses mainly on the two central activities of the loop, i.e., analysis and planning, and the knowledge component.

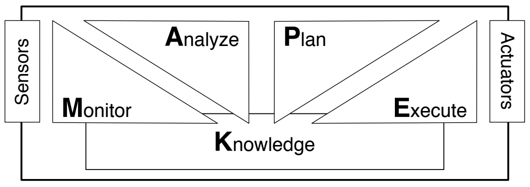

Figure 2.2. Autonomic element

The high level objectives that controllers try to achieve may concern several aspects, called *self-** properties, that include: optimization, configuration, fault tolerance and security. Self-optimizing systems aim to maximize some utility function of the system; self-configuring systems adapt the software architecture to dynamic environments; self-healing systems recover from failures; and, self-protecting systems deal with security attacks.

Simple self-* systems usually deal with only one property, while more advanced solutions deal with many properties at the same time. This thesis is centered around the design of controllers that provide the system mainly with self-optimizing and self-configuring abilities.

Common controllers architectures

Controllers come in different flavors and with several architectures as discussed by Brun et al. [15] and Patikirokorala et al. [83]. Authors group controllers in centralized and distributed depending on their architectures.

Centralized controllers are the most common solutions and refer to centralized single-controllers and centralized multi-controllers.

Figure 2.3 exemplifies some centralized controllers as block diagrams: I) Feedback controllers. They use output of the system to compute an error metric and plan a control strategy that reduces it. II) Feed forward controllers. They receive the same input signals of the controlled system and compute the control strategy on them. III) Feedback and feedforward controllers. They combine the former two controllers in a single solution. IV) Model predictive controllers. They employ a model of the system in conjunction with an optimization procedure to define the control strategy: At each control interval, controllers simulate the system using the model and plan the best strategy over a receding horizon. V) Cascading controllers. They chain two multiple controllers such that outer controller controls the inner controller, and the inner controllers act on the system. VI) Multi controllers with switching. They run several controllers in parallel and use them in mutual exclusion to control the system.

Distributed controllers are the alternative to centralized controllers and their control logic is deployed over multiple controllers. We report the most common archi-

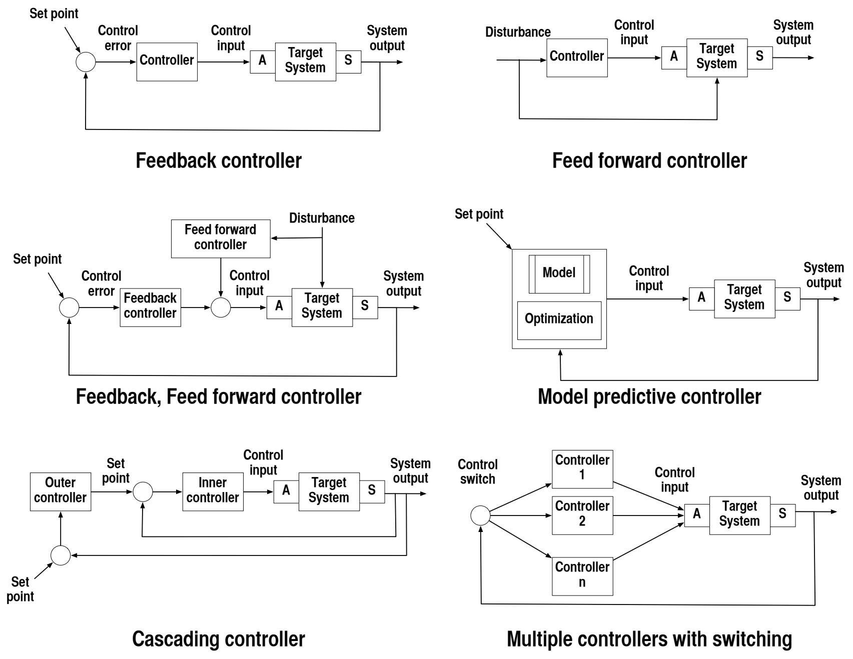

Figure 2.3. Block diagrams of centralized controller architectures [83]

tectures of distributed controllers in Figure 2.4: I) Hierarchical controllers. They are organized hierarchically, and controllers in upper-level act on controllers in the lower-levels, and eventually on the target system. II) Fully distributed controllers. They are composed of independent local controllers that communicate over a shared communication substrate. Controllers collaborate to fulfill the control goals.

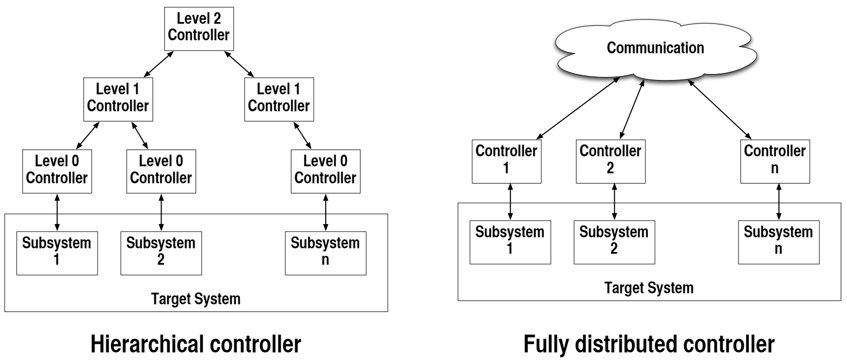

Figure 2.4. Block diagrams of distributed controller architectures [83]

Controllers presented so far assume that the system behavior is stable and that all the system identification activities are done before going live. This is not always possible because the controlled systems and the environment may show transient and emerging behaviors, or because an extensive model identification is too expensive.

Self-adaptive controllers instead are able to adapt as the system behavior changes.

Controllers adaptivity is a concept taken from advanced control theory where it is usually implemented in the form of automatic tuning of controller parameters or tracking of *hidden* variables [116]. In the context of autonomic computing, self-adaptivity has a broader meaning than parameter estimation and variable tracking, and encompasses also updates on control strategies and structural changes of controllers internal architecture [15].

In Figure 2.5, we illustrate the block diagrams of generic self-adaptive and self-tuning controllers. Self-tuning controllers employ an estimator component that tracks the hidden variables and provides the controller with their values. Self-adaptive controllers employ a reconfiguration layer that acts on the control layer to adapt it. For example, Magee and Kramer propose a three-layered control approach where an high level goal management layer controls the underlying levels for change management and software components [53].

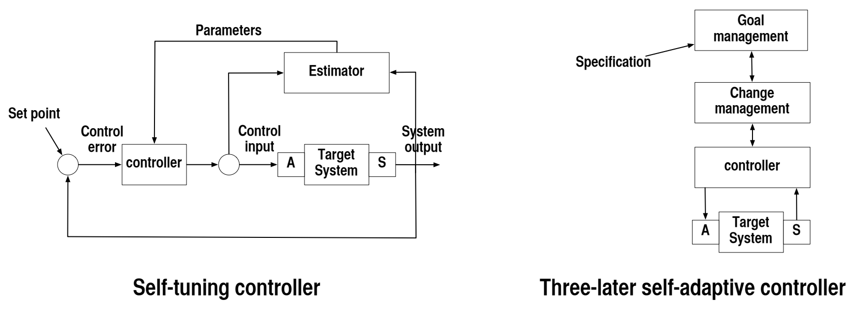

Figure 2.5. Block diagrams of common self-adaptive controllers

In this thesis we focus on centralized controllers, in particular on model based self-adaptive centralized controllers, to tackle the problem of dynamically control elastic applications in the Cloud: The control strategy is derived from models that describe the system behaviors that are adapted at runtime. We choose the centralized approach over the distribute one, because it is conceptually simpler and easier to manage, and we choose a model based controller because is more flexible and generally applicable than standard feedback and feedforward solutions.

2.3 Autonomic and self-adaptive controllers for the Cloud

Autonomic and self-adaptive controllers are designed to deal with complex systems, highly dynamic environments, uncertainty and emerging behaviors. Additionally, they can learn from past experience [49]. These characteristics make them a sensible choice for managing elastic applications running in Clouds. In fact, Clouds are very dynamic environments that provide high degree of automation to carry out all the management tasks, and host complex applications that may be composed of several interdependent software components, and may face highly variable workloads.

In Cloud settings, state-of-the-art approaches that are based on autonomic computing principles can be classified as: (i) approaches used *inside* the Cloud to perform activities on its internals, such as controlling resources allocation between the hardware and virtualized layers, (ii) approaches used *outside* the Cloud to perform activities that target user applications and leverage Cloud features , and (iii) approaches that combine the previous two.

Inside the Cloud, controllers are applied mainly for optimization goals: For example, they are used to optimize the placement of virtual machine to minimize the number of active physical servers (server consolidation)[16]. Or they are used to adjust the CPU frequency of physical servers to improve their power efficiency while maintain VM-related metrics to specified levels[48; 92].

Outside the Cloud, controllers are commonly used to manage final users applications: For example, they are used to control the horizontal and vertical scale of elastic applications, or to dynamically migrate VMs between physical machine, data centers, or federated Clouds to maintain acceptable levels of dependability.

The focus of the thesis is on the use of autonomic approaches outside the Cloud, that is, *exogenous* controllers [115]. We make this decision bearing in mind generality and separation of concerns: By developing controllers that run outside the Clouds we can reuse the same controller across different Cloud infrastructures as long as the controlled application remains the same. Furthermore, by running controllers on the side of service providers, we leave full control to service providers on the controllers if they want to remove, change, or update them. This is different for example from the approach offered by Amazon and other Cloud providers that allow service providers to specify rules, but fully control their execution.

We design controllers that operate on behalf of service providers and manage their elastic applications at runtime: Controllers bind public interfaces exposed by Clouds, monitor applications behavior and billing status, and take informed decisions that trade off running costs and quality of service before acting upon the virtualized infrastructure. In particular, controllers take decisions about what type and how many instances of virtual machines have to be replicated or removed, and when to implement the control actions.

In this thesis, we follow the reference architecture for self-adaptive controllers and we adapt it to fit the Cloud domain. Figure 2.6 shows the resulting situation: The controlled system is a virtualized application that runs inside the Cloud. The Cloud is operated by an IaaS Cloud provider. The environment embeds both the controlled system and the Cloud. Final user perturb the application by means of the service invocations. The controller lies outside the Cloud, monitors the invocations by the final users and the service behavior, and invokes operations at the infrastructure level on the Cloud.

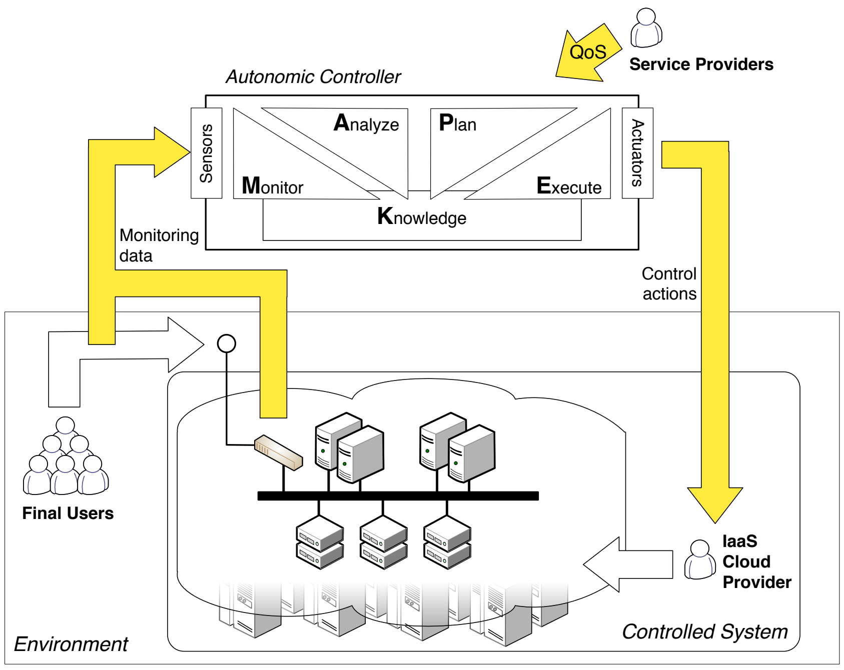

Figure 2.6. Reference architecture for autonomic controllers in the context of elastic applications and Cloud IaaS

2.4 Kriging models

This thesis proposes Kriging models as the basis to build self-adaptive model based controllers. This section introduces Kriging modeling, focusing on the aspects of interest in this thesis. The interested reader can find a comprehensive presentation of Kriging models in [90; 107].

Kriging models were defined in the context of mathematical geology by the French mathematician Georges Matheron [72]. They are named after Daniel G. Krige, the South-African mining engineer who firstly proposed them in the 1950s [54]. Originally, Kriging models were proposed to approximate the underground concentration of valuable minerals by interpolating samples taken at different locations, and gradually they have been used in other contexts. Starting from the seventies, Kriging models have been used to develop approximations[5] of deterministic and expensive simulation for designing optimization functions in domains such as geo-statistics, meteorology, avionics and circuit design [94]. Recently, Kriging models have been used in Machine Learning, where they are known as Gaussian processes [90].

Kriging models are a family of black box models that approximate non linear multimodal functions. They are build starting from a training data set and capture the rela-

[5]In relation of this application as approximations of simulations Kriging models are also referred to as *surrogate* models or meta-models.

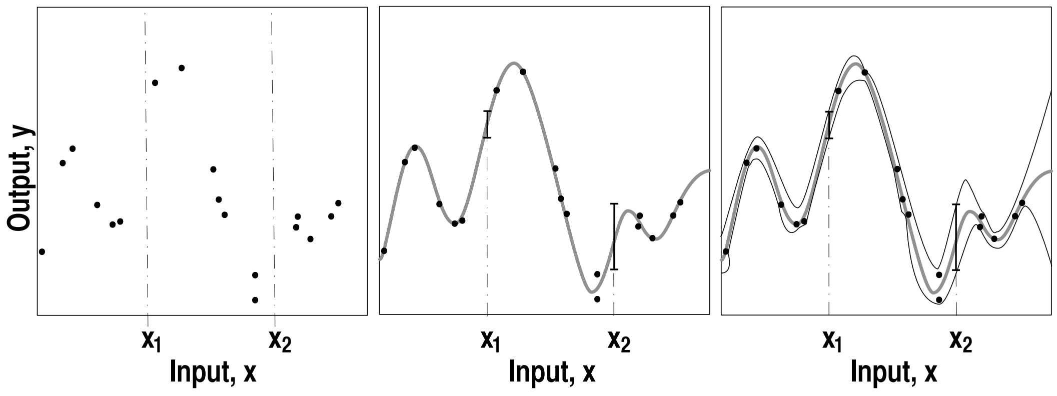

Figure 2.7. Example of regression with Gaussian Processes [88]

tions among input and output data, a process commonly known as *supervised learning*.

Once trained, they predict output values at input locations that were not present in the training set and that we denote by $\mathbf{x}_*$. Kriging models are suitable for regression problems when inputs and outputs refer to continuous dimensions, but can also be used for classification problems that involve non-continuous features [23]. Figure 2.7 shows how Kriging models works over a simple example taken from [88]: The left panel reports the training points (as black dots) that we use to train a Kriging model to predict the output function (y). The panel highlights two unknown points (x_1 and x_2) that we want to predict with the model. The central panel shows the Kriging model that results from the training data (as solid gray line). For x_1 and x_2, the panel shows also the confidence interval that is associate with the predictions as error bars centered in x_1 and x_2. The right panel shows the complete Kriging model. In this panel, we plot the model predictions and the corresponding confidence area for the whole input space.

Kriging models have many interesting properties: (i) They are *global*, i.e., they span over the whole input space, differently form classic regression models that are commonly preferred for local predictions. (ii) Whenever it is assumed that data are noise free, like in the context of *deterministic* computer simulations, Kriging models produce the *exact* interpolation of data. Otherwise, they implement a form of data *smoothing*. (iii) Kriging models can provide a measure of the uncertainty of their own predictions; this measure accounts for both the availability of training samples, their distribution over the input space, and their level of noise. (iv) Kriging models in general do not need a very large amount of training samples to provide acceptable predictions, thus are relatively fast to train.

Kriging are a valuable alternative to other black-box models, either classic regression methods that are based on first- or second-order polynomials, and non-parametric models, such as smoothing splines and radial basis functions [107]. Differently from classic models like artificial neural networks (ANNs) or traditional linear regressors that require extensive configurations in terms of layers, neurons and model degrees to match systems complexity, Kriging models are easier to use because they automatically search for the relationships among measured data in the scope of a fixed, but very expressive, internal structure.

Both ANNs and Kriging models can be used as black-box models to predict not yet experienced values, but only Kriging models can be inspected and interpreted to better understand the properties of the predicted values: By inspecting the model after the training, we can derive conclusions about the relative influence of each input dimension on the modeled output, and this can be used, for example, to filter out not-informative input features, thus reducing the complexity of the model itself [90].

As any other approach based on data, Kriging models depend on the availability and the quality of training data to make accurate prediction. However, Kriging models can leverage an uncertainty measure to identify regions in the input space where training data are scarce compared to the complexity of the unknown function. We can use this measure to account for the reliability and risks of using the values predicted by the models.

The remaining sections give an intuition on how Kriging models work along with the two main point of view: Kriging models as extensions of traditional linear regression, and Kriging models according to the Bayesian theory.

Internals of Kriging models

Given a training set consisting of input vectors ($\mathbf{x}^i$) and corresponding observations (y^i), the aim of supervised learning is to infer a model of the (unknown) function (f) that has generated the data. The model of the unknown function is generally given by a linear combination of parameters and functions to be estimated.

$$\hat{y} = f(\mathbf{x}) + \epsilon \tag{2.1}$$

where ϵ is the error term. Observations are assumed independent and identically distributed (i.i.d) and the error is assumed Gaussian with zero mean and given standard deviation σ_ϵ^2.

Two main viewpoints describe how Kriging models work: According to the first, Kriging models extend traditional interpolation methods with a statistical framework based on stochastic processes [6], typically Gaussian processes [23]. According to the second viewpoint, Kriging models can be framed in the Bayesian theory in terms of prior and posterior distributions, likelihoods and observations.

[6]Stochastic processes generalize stochastic distributions to functions: A distribution describes a random variable, while a process describes a random function.

Kriging model as extensions of linear regression

As summarized by Jones et al. [45], Kriging modeling can be interpreted as an extension of classic regression with stochastic processes. In fact, they share a common mathematical framework that includes regressors and errors, but they put different emphasis on the two aspects: Classic regression focuses on the regressors and their coefficients, and makes simplistic assumptions about the errors and their independence. Kriging modeling instead focuses on the correlation structure of the errors, and makes simplistic assumptions about the regressors. In other words, classic regression is about estimating regression coefficients that together with the assumed functional form fully describe *what the function is about*, while Kriging modeling is about estimating correlation parameters that describe *how the function typically behaves*, thus describing how much the function tends to change while moving along different input dimensions.

Kriging models correlate training samples by assuming that the target function is continuous and that there is a *spatial* correlation between observations: Samples that are close to each other in the input space have similar output values. Kriging models capture this correlation by exploiting the error terms: ϵ is modeled as a stochastic process and it depends on the observation, therefore it can be re-written as $\epsilon(\mathbf{x}^i)$. From the assumption of continuity of the unknown function and the general trend, it follows that $\epsilon(\mathbf{x})$, which is computed as their difference, must be continuous too. And for any two points close together ($\mathbf{x}^i$ and $\mathbf{x}^j$) the corresponding errors terms ($\epsilon(\mathbf{x}^i)$ and $\epsilon(\mathbf{x}^j)$) should be also close.

Figure 2.8 captures this intuition over a simplified example that we take from [45]: The thick line represents the unknown function, and the black dots represent the collected observations. Let's assume that there is no additive noise in the data and that the empirical mean ($\hat{\mu}$) is the global trend, which is represented as a thin line in the diagram. According to the theory, given an x_i, the error term $\epsilon(x_i)$ is computed as the distance between the model, which is $\hat{\mu}$, and the actual value of the function y_i. From the figure, we can intuitively see that error terms $\epsilon(\mathbf{x}^*)$ and $\epsilon(\mathbf{x}^i)$ that corresponds to the unknown location $\mathbf{x}^*$ and the nearby location $\mathbf{x}^i$ have similar values.

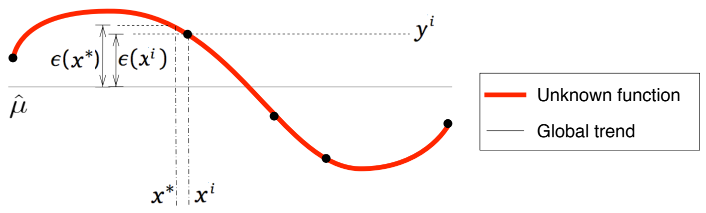

Figure 2.8. Error correlation in Kriging models [45]

Kriging models capture the correlation of samples in a correlation matrix that we denote by $\mathbf{K}$. Given $\mathbf{K}$, Kriging models predict the value of the target function in the unknown coordinates $\mathbf{x}^*$ as the *most consistent* value with respect to the estimated typical behavior of the function that is summarized by best linear unbiased predictor:

$$\hat{y}(\mathbf{x}^*) = \mu(\mathbf{x}^*) + \mathbf{k}^T\ \mathbf{K}^{-1}\ (\mathbf{y} - \mu(\mathbf{x}^*)\mathbf{1}) \tag{2.2}$$

The first term of the predictor is the evaluation of the mean at the untrained input and the second term is the adjustment that is based on the correlation among the errors. It must be noted that, if there is no correlation ($\mathbf{k} = 0$) the predictor reduces to the mean ($\hat{\mu}$), while at a sampled points ($\mathbf{k}$=$\mathbf{K}_i$) the predictor interpolate the data. In between these extremes, $\hat{y}$ is based on a smooth interpolation of the data

The estimation of accuracy of the prediction is affected by the correlation of the error terms and it can be measured as the mean squared error (MSE) with the following formula:

$$s^2(\mathbf{x}^*) = \sigma_\epsilon^2\ [1\ -\ \mathbf{k}^T\mathbf{K}^{-1}\mathbf{k}\ +\ \frac{(1\ -\ \mathbf{1}^T\mathbf{K}^{-1}\mathbf{k})^2}{\mathbf{1}^T\mathbf{K}^{-1}\mathbf{1}}] \tag{2.3}$$

The term $\mathbf{k}^T\mathbf{K}^{-1}\mathbf{k}$ represents the reduction in prediction error due to the fact that $\mathbf{x}^*$ is correlated with the training points, i.e., no correlation implies no adjustment. The term $(1 - \mathbf{1}^T\mathbf{K}^{-1}\mathbf{k})^2/\mathbf{1}^T\mathbf{K}^{-1}\mathbf{1}$ reflects the uncertainty that stems from not knowing the function exactly.

Kriging model according to the Bayesian view

Kriging models can be interpreted also according to the Bayesian theory. This is the viewpoint of the Machine Learning community, and is summarized by Rasmussen et al. [90].

The inference process starts with a *prior* distribution over a set of possible functions that may represent the unknown function. In this particular case, the prior is a Gaussian process and samples drawn from the unknown function are assumed to be normally distributed. Their covariance is assumed to be depend on samples spatial location, i.e., the distance between them. The prior is combined with a set of *observations*, i.e., the training points, to obtain the *posterior* distribution that is used to make the predictions. This combination is governed by the Bayes rule:

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{marginal likelihood}}$$

The prior encapsulates all the knowledge that is available at the beginning, i.e. before collecting any data. In our context, the prior encapsulates the expectations over the kind of functions to be observed: Smooth functions with *stationary* variance. The information from the training set are used to derive the likelihood and the marginal likelihood (or evidence). Finally, the resulting posterior, which is also a Gaussian process, describes the expected behavior of the function as was derived from the observations.

Figure 2.9 gives the intuition of the inference process with the Bayesian model in a simple example. We take the example from [90]. The left panel shows some of the distributions (as solid lines) that are drawn from the Gaussian process used as prior. The same panel highlights also the confidence interval of the Gaussian process as gray area. The right panel instead shows how this evolves after the observation of two samples. The panel shows some distributions (as dotted lines) that are drawn from the resulting posterior and the confidence interval. In particular, and as the intuition suggests, the confidence interval adapts to the data: It is reduced close to the observations while is not changed away form them.

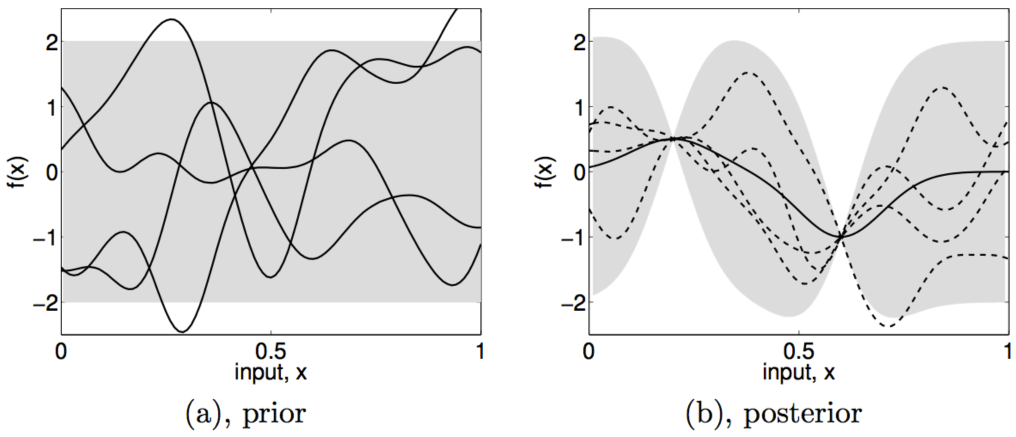

Figure 2.9. Bayesian inference with Gaussian processes [90].

Informally, the net result of this inference process is that some of the potential distributions that could be drawn from the prior are filtered out with the observations: Distributions that behave according to the original prior **and** the data are retained while the others, which do not interpolate the observations, are removed. Once the posterior is available, its mean function computes the predictions and its variance (at the prediction locations) computes their uncertainty. The figure shows the mean function of the posterior as solid bold line.

This theory involves the computation of several integrals and other complex mathematical operations, but by focusing on stochastic processes that are Gaussian, it turns out that these computations become relatively easy as the model can be treated analytically. We can write the general model (see 2.1) in terms of the parameters $\mathbf{w}$ and make inference on them[7]:

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{w} \tag{2.4}$$

In this formulation, the problem is to find the value for the parameters using the Bayes rule, i.e., find the posterior distribution of the parameters given the prior and the observations

$$p(\mathbf{w}|\mathbf{X},\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{X},\mathbf{w})p(\mathbf{w})}{p(\mathbf{y}|\mathbf{X})} \tag{2.5}$$

[7]An equivalent derivation in the function space can be performed as well.

In absence of evidence, the common prior ($p(\mathbf{w})$) is the zero mean Gaussian with covariance matrix Σ_p. This choice is not a huge limitation because the resulting posterior will have a mean that is different from zero because that depends on the training data. Of course, whenever some *a-priori* knowledge is available, it can be *plugged* in the model by defining a suitable prior.

Thanks to the independence assumption of the observations, we can factor the likelihood over the training points to obtain

$$p(\mathbf{y}|\mathbf{X},\mathbf{w}) = \prod_{i=1}^{n} p(y_i|\mathbf{x}_i,\mathbf{w}) = \ldots = \mathcal{N}(\mathbf{X}^T\mathbf{w}, \sigma_\epsilon^2\mathbb{I}) \tag{2.6}$$

which is Gaussian. Finally, the marginal likelihood is independent of the parameters and is given by

$$p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|\mathbf{X},\mathbf{w})p(\mathbf{w})d\mathbf{w} \tag{2.7}$$

It can be shown that the posterior is Gaussian and is given by

$$\begin{aligned} p(\mathbf{w}|\mathbf{X},\mathbf{y}) &\sim \mathcal{N}(\overline{\mathbf{w}}, A^{-1}), \quad \text{where} \\ A &= \sigma_\epsilon^{-2}\mathbf{X}\mathbf{X}^T + \Sigma_p^{-1}, \quad \text{and} \\ \overline{\mathbf{w}} &= \sigma_\epsilon^{-2}A^{-1}\mathbf{X}\mathbf{y} \end{aligned} \tag{2.8}$$

The model computes the prediction for a given input by averaging over all the possible parameter values weighted by their posterior probability. The prediction and its uncertainty are given again by the mean and variance of a Gaussian distribution that correlates test input with the observed values in a well defined form.

$$\begin{aligned} p(y_*|\mathbf{x}_*,\mathbf{X},\mathbf{y}) &= \int p(y_*|\mathbf{x}_*,\mathbf{w})p(\mathbf{w}|\mathbf{X},\mathbf{y})d\mathbf{w} \\ &= \ldots \\ &= \mathcal{N}(\frac{1}{\sigma_\epsilon^2}\mathbf{x}_*{}^T A^{-1}\mathbf{X}\mathbf{y}, \mathbf{x}_*{}^T A^{-1}\mathbf{x}_*) \end{aligned} \tag{2.9}$$

Sample correlation is the central point of Kriging modeling

In their more general formulation, Kriging models contains two terms: The first is called *global trend* and is meant to capture model-wise variations; the second instead accounts for *local* modifications and it is implemented by means of Gaussian processes.

Universal Kriging models put no constraints on the global trend, Ordinary Kriging models assume a constant but unknown global trend, while Simple Kriging models assume a known constant trend that, by convention, is equal to zero. These models allow for different levels of expressivity, but in any case, the most important part remains the correlation function.

The correlation depends on a point-wise distance metric, it must be maximal when the distance between the corresponding points is small and almost zero when that distance is great. Its common form is given by

$$\mathrm{Corr}[\mathbf{x}_i, \mathbf{x}_j] = \exp[-\mathrm{dist}(\mathbf{x}_i, \mathbf{x}_j)] \tag{2.10}$$

In particular, Kriging modeling defines such a distance metrics as:

$$\mathrm{dist}(\mathbf{x}_i, \mathbf{x}_j) = \sum_{h=1}^{d} \theta_h |\mathbf{x}_h^i - \mathbf{x}_h^j|^{p_h} \tag{2.11}$$

but alternative formulations are also possible [90].

The behavior of this distance metric is governed by two *hyper*-parameters for each of the H input dimensions: p_h determines the *smoothness* of the function, while θ_h determines its *activity*, i.e., the amount of variation of the function per unit variation in the input. Thanks to the hyper-parameters, different weights can be associated with different input dimensions, thus modeling their relative importance. This is very different from standard Euclidean distance metrics that treats all the input dimensions at the same way.

The value of the hyper-parameters must be set before we can actually compute the predictive mean and variance, and in practice, this is done by elaborating the training data. As Kriging models aim to capture the *typical* behavior of the unknown function as expressed by the training data, the value of hyper-parameters is obtained via maximizing the likelihood (MLE) of the sample within the average value of the function and the variance of the error[8].

To carry out the optimization, a total of $2|H| + 2$ parameters must be estimated: two hyper-parameters for each input dimension, plus the mean and variance of the underlying stochastic process. For example, compared to a quadratic model that needs a total of $(|H| + 2)(|H| + 1)/2$ parameters, the set of parameters for Kriging models is smaller and the optimization may converge faster.

For the sake of presentation, in this section we assumed noise-free data. In practice, as we use Kriging models to describe real systems, we cannot make such an assumption. In the remaining of the thesis, we follow the approach presented by Forrester [27], and we add a "noise term" (thus more unknown parameters to be estimated) to the diagonal of the correlation matrix. Assuming a Gaussian noise, all the previous conclusions derived from Kriging models remain valid. The net effect of adding the noise term is that the final model implements a form of data smoothing and it may not perfectly interpolate the data. This modification is sensible as more noise implies more smoothing of the data and more uncertainty of the predictions.

[8]Even if the goal of the optimization is to maximize the likelihood, usually the actual optimization problem (for numerical reasons) is formulated as minimization of the negative log-likelihood.

Chapter 3

Related Work

This chapter presents the most relevant work in the following areas: Control of Cloud based applications, performance modeling by means of black box and surrogate models and control based on Kriging models and Gaussian processes. It is organized as follows: Section 3.1 presents work about automatic scaling of elastic application in Clouds IaaS; Section 3.2 presents work about the use of surrogate models for performance prediction in the context of performance engineering; Section 3.3 presents work on Kriging models and Gaussian processes used in the context of automatic control.

3.1 Controllers for elastic applications in Clouds

Controllers for elastic application in the Clouds belong to a research area known as dynamic system management (DSM) that deals in general with the study and design of controllers. A broad view on DSM is out of the scope of the thesis, so we give here only a very short summary of the various proposals that DSM accounts for: Hardware resources allocation, adjustment and scheduling for performance guarantees [52; 73; 79; 65; 119]; green computing, power management and dynamic CPU frequency scaling for reducing resource consumption and impacts on the environment [55; 56]; system scaling and component replication to align system capacity and workload demand [76; 26; 33; 84]; components and VM migration, redeployment and rescheduling to better consolidate servers [50; 105; 106]; dynamic access control, load balancing and other dynamic dispatching techniques, service composition adaptation and rebinding, to provide consistent, stable and suitable QoS to clients [10; 17; 3; 99; 5; 75; 74].

In our context, controllers aim to identify suitable allocations of resources at runtime, based on some knowledge encapsulated in the controllers in the form of rules or models. The problem of building efficient self-adaptive controllers has been the target of several research projects, which resulted in many approaches that differ for the models used to capture the knowledge of the system and for the strategies to identify suitable configurations while reacting to changes of workload conditions. The models of system behavior that are used in those controllers span from simple rules to complex analytic or surrogate models.

Some approaches require models to be defined and tuned at design time while others define and tune models at runtime. This choice depends on the nature of the models, the complexity of the system and the reliability requirements of the controllers: Rules and analytic models, like queuing networks, must be defined and tuned at design time and hardly adapt to unpredictable configurations; surrogate models, like Kriging models, may be defined and tuned at runtime and can adapt to emerging scenarios. In practice, defining proper analytic models for complex systems may require a considerable effort, but their reliability can be assured analytically or experimentally before deployment. Surrogate models, instead, can be tuned dynamically according to the system behavior either during testing or at runtime, sometimes even independently from the complexity of the modeled system. Their reliability however can be assured only on the basis of some hypothesis on the behavior of the system that may not be statically verifiable.

We classify related work about automatic controllers for the Cloud according to two main dimensions: *Assurance*, which refers to degree of predictability of the runtime behavior, and thus the level of reliability of the system, and *flexibility*, which refers to the capability of controllers to cope with changing and evolving configurations, and thus to the level of adaptability of the system.

We estimated the assurance level of the different approaches considering the formality of the control model, the hypothesis on the system behavior and the correspondence between the system and the model. Formality of the control model accounts for the amount of formal guarantees that each approach gives on the model or control. As an example, in traditional control theory, under proper assumptions, the controlled system can be guaranteed to converge to an equilibrium point within a given range of oscillations. Hypotheses on the system behavior accounts for the set of assumptions that need to be valid for the approach to correctly model the system, and thus to work suitably. For example, control theory assumes system linearity, while Kriging models assume a certain degree of smoothness of the modeled system. Finally, correspondence between the system and the model accounts for the (timely) coherence between the current state of the system and its runtime models, similarly to model accuracy.

We estimated the flexibility of the different controllers by considering the amount of configurations that they can be encompassed and the possibility to extend the set of considered system configurations at runtime. For example, some approaches assume that the system never changes because they cannot be adapted, others instead have limited adaptations capabilities, such as the tuning of some parameters, while the remaining can be updated to greater extent.

Similarities in these approaches suggested the creation of three groups that we discuss separately thereafter: Static approaches, dynamic approaches and hybrid approaches.

Static approaches

Static approaches are based on models of the controlled system that are defined and verified at design time and not modified, updated or tuned in production after system deployment. We refer to these models as static models to stress the design nature of their construction and analysis. These models privilege design time assurance over runtime flexibility: The models of the system identified at design time are not changed during runtime activities to preserve the validity of the proofs used to identify system *viability zones* that are defined as states in which the system operation is not compromised [100], along with the main properties of the system behavior. Being defined at design time and not updated at runtime, static approaches rely heavily on the correctness and completeness of the knowledge encoded in their models. This implies an extensive degree of experience and understanding of the system behavior, and generally high costs in terms of model identification and/or synthesis. Moreover, formal approaches, in particular classic control theory, are generally applicable under strong assumptions in terms of linearity, monotonicity, and reliability of the controlled system, thus significantly reducing the system viability zones. Under these assumptions, feedback-loop approaches can be proven robust to a certain degree of error in the estimation of the model parameters, hence they can provide effective control actions (for example, maintaining a performance metric within a certain range) even without self-tuning at runtime. However, both errors in parameter estimation and wrong assumptions on system behavior properties can reduce the efficiency of the control in terms of possible SLO violations or resource assignment.

The most popular static approaches are based on either analytical models or rules and thresholds, while only some of them are based based on black-box and surrogate models defined and tuned before deployment.

Approaches based on Control Theory Cloud controllers based on control theory adapt classic control theory techniques in the context of computing systems aiming to design adaptive, robust, and stable systems [66]. Here we provide only basic information about classic control theory, and we suggest Hellerstein et al. for additional details on control theory approaches to manage computing systems [37].

Some approaches adapt standard control techniques, such as proportional, integral and derivative techniques, to synthesize self-adaptive controllers. These controllers use simple models and provide formal guarantees about the behavior of the system, but rely on very strong assumptions that are seldom verified in highly varying environments. Other approaches try to extend the scope of applicability of controllers based on classic control theory, by proposing complex parametric models where part of the parameters are unknown at design time. These approaches can still provide a limited set of formal guarantees depending on the assumption of proper on-line estimation methods.

An interesting example is the control theoretic solution that Maggio et al. propose to deal with self-optimization problems [67]. This solution is developed in the context of a common framework for monitoring system performances and adjust the allocation of resources to applications in order to guarantee a predefined service level. Maggio et al. [66] start with a simple (stateless and linear) model of the system that assumes a monotonic relation between allocated resources and application target performance. Then, they synthesize a Deadbeat controller, which is a common choice in the context of standard control theory, to track the target performance signal. They study the transient behavior of this controller by setting different parameters to estimate the approximation of the input signal and to decide if the controller can effectively regulate the system despite its variations. When they cannot determine the suitability of the current model, they further refine the model by introducing more complex techniques and see if the new models are conclusive.

Maggio et al. improve the basic deadbeat controller by adding an identification block that supports the online estimations of the unknown system parameters [68]. They extend the controller by means of a Kalman filter and a Recursive Least Square algorithm. They observe that adopting more complex solutions, i.e., solutions that use more parameters to describe the relationship between the controlled and target variable, increases the difficulty to prove suitable control properties.

Other theoretical control solutions leverage multiple models. A good example of such controllers is the approach of Patikirikorala et al. who propose a particular instantiation of Multi-Model Switching and Tuning (MMST) adaptive control [82]. In a nutshell, they define several (fixed) linear models that describe the system behavior in different working conditions, and synthesize different single-model controllers following the classic control theory. The overall controller monitors the system variables, computes an error metric for each of the models, and enable only the single-model controller that correspond to the minimum predicted error.

Threshold and rule-based Threshold-based policies are popular in current industrial applications, as they are simple and intuitive to understand. They are applied by defining lower and upper limits on target metrics. Behaviors outside the ranges trigger reconfiguration actions. Threshold-based controllers are used in many companies such as Amazon[1], RightScale[2] and Scalr[3]. In many controllers currently used in industrial applications, upper and lower bounds are defined by the customers referring to low level metrics, such as CPU usage. Customers define also the corresponding reconfiguration actions. Dutreilh et al. [24] experiment with static threshold-based policies that

[1]http://aws.amazon.com/autoscaling/
[2]http://www.rightscale.com/
[3]http://scalr.net/

rely on high level metrics, such as response time. They define control policies based on upper and lower thresholds, on fixed amounts of virtual machines to be either allocated or deallocated, and on pairs of "inertia" durations that support scaling up and scaling down periods. For instance, if the response time exceeds the upper bound, the controller allocates virtual machines, and inhibits itself for the corresponding inertia interval. If the response time drops below the lower bound, the controller deallocates virtual machines, and again inhibits itself for an inertia interval.

Rule based approaches extend threshold solutions by considering different types of events, and allowing rules to trigger other rules following the common ECA (event, condition, action) paradigm. An interesting rule based approach is *Claudia*, a rule-based controller for virtualized services proposed by Rodero-Merino et al. [93]. Claudia is more general than most rule based approaches, since it considers service lifecycle events and user defined variables, called Key performance indicators (KPIs), as well as business level metrics that holistically describe the status of a complete virtualized service and eventually triggers rules that reconfigure the system. Claudia combines three different types of rules, all defined by Cloud users: *Scaling rules* that resemble traditional threshold based approaches and change the number of allocated virtual machines, *reconfiguration rules* that act at deployment time and choose the size and type of virtual machine to be deployed, and *business rules* that constrain the automatic scaling behavior with respect to running costs by limiting for instance the total number of running virtual machines. Business rules consider also Cloud federation concerns, for instance by *migrating* virtual machines from one Cloud provider to another. Claudia monitors the virtual service execution and periodically tries to fire user defined rules that eventually trigger control actions.

Approaches based on Analytic Models Controllers based on analytic models use utility functions that combine system performance metrics and business considerations, like revenue and resource usage cost, to find desirable system configurations. They compute system performance metrics (typically the response time) through either different queuing models or analytical representation of queuing models. Analytical approaches differ each other mainly in terms of the chosen formalism, the complexity and accuracy of the models, and the policy adopted to solve the optimization problem.

Benanni and Menascé combine queueing models and combinatorial search to dynamically allocate resources to application environments [76]. They associate a local controller to each application environment and use queueing network models (open models for transactional systems, closed model for batch processing system) to predict the performance metrics of the application environment. Each local controller computes a utility measure by combining performance metrics, service level agreements and penalty functions, and sends the computed utility to a global controller that uses a combinational search to identify the final resources allocation of the entire data center, i.e., of all the application environments. While exploring the configuration space, the global controller interacts with the local controllers by suggesting potential new resource allocations, and the local controllers respond with updated values of local utilities. The queueing networks and their parameters are defined at design time.

Huber et al. introduce an architecture level descriptive model in the context of the Palladio framework, from which they derive performance models that are used by self-adaptive controllers to predict the system behavior [11; 39]. When the input workload changes, the control algorithm adds resources (virtual machines) to the system in order to eliminate all actual and predicted service level agreement violations, then the controller removes the resources that are underutilized. Huber et al. assume the availability of the architecture level models and estimate the parameters of the model at design time, for example, resource usages, routing/calling probabilities, service times, and possible usage scenarios and user classes. Once the model is in place, an automatic procedure transforms it into a predictive performance model, i.e., a queuing network, that is used on-demand by the control algorithm.

Bi et al. [13] use an mixed queueing model to simulate multi-tier applications and define a non-linear optimization problem over it to dynamically decide on the system at per-tier. The mixed model combines an M/M/c queuing model, for the *front-end*, with several M/M/1 queueing models for the *per-layer* virtual machines. The optimizer uses the model to calculate the number of resources to provision at the each tier according to the target end-to-end response time for the tier that is assumed to be agreed with customers.

Dejun et al. [44] address Web applications modeled as acyclic compositions of services using a what-if-analysis and negotiation among the composed services. Each service estimates its own performance variation in the case of changes of the allocated resource or workload, and pass the estimate up through the invocation tree to produce local decisions that are incrementally aggregated all the way up to a root controller. Dejun et al. model the performance of each single service as a M/M/n/PS queue, and consider performance variations for configuration changes of a single VM (+1 or -1 machine per service). The controller adjust the predictions by estimating the service time for the queuing networks at runtime using a feedback control loop: A threshold on the prediction error of the system performance triggers a new estimation of the service time by using the latest measured response time. Dejun et al. show the effectiveness of the proposed controller for different types of compositions, and claim that a service level agreement for the front-end service allows for finer control of the composed services than service level agreement thresholds on each component.

Approaches based on black box and surrogate models The use of analytic modes is limited by the need of accurately defining a model structure and computing many parameters. Black box and surrogate models address this problem by generating the models from data gathered during system executions, and thus obtaining models that correspond to the system by construction. We will see in the next section that black box and surrogate models can be derived and adjusted at runtime, thus increasing the flexibility and adaptability of the controllers. Here we survey the main models that are proposed to be tuned before the systems deployment, typically at testing time and during model training.

Vasic et al. use a workload signature based on Hardware Performance Counters to cluster workloads, and associate the identified clusters to previously measured appropriate resource allocations [108]. A proxy collects and computes workload signatures both in the training and control phase, by replicating a fraction of the entire application workload that is directed to a profiler for sampling and measurement. At runtime, a profiler computes the incoming workload signature, a classifier associates it to a class, and the controller applies the recorded resource allocation in a single control action. The controller uses a metric of the *certainty* of the association of the workload to a cluster as an indication of the need to trigger a new training of the classifier. The approach also measures the *interference* of other applications running in the same infrastructure, where interference is defined as the ratio of the performance achieved in production w.r.t. the performance for the same workload signature and resource assignment measured at training time. In a nutshell, the system works as a cache for previously observed combinations of workload signatures and resource allocations. In case of a cache *miss*, the default policy is to bring the controlled system to its maximum resource allocation to prevent service level objective breaches. The limits of the validity and assurance of the control derive mainly from the choice and appropriateness of the metrics used to compute the signature and cluster the workloads. In their experiments, the authors report that the clustering typically results in only few workload classes while applications can normally have very large workload and configuration spaces.

Trushkowsky at al. address the on-line reconfiguration of storage systems in response to workload changes under stringent performance requirements [104]. The controller manages SCADS, a key-value store that offers eventual consistency and an easy partitioning of the key-value stores, hence natively supports replication and elasticity [6]. Two specific issues make the problem hard: 1) to scale a data-intensive system, data items must be moved or copied across instances, impacting negatively on service performance, and 2) high percentiles of response time have much higher variance (and therefore are much harder to control) than the center of the distribution (average response time). The latter issue can cause oscillations in classical closed-loop control. Trushkowsky at al. tackle these issues by introducing a model predictive control: The controller refers to a model of the system and its current state to compute the optimal sequence of control actions, executes the first action of the sequence, and then recomputes the optimal sequence of control actions to chose the next actions. The system performance model coupled with workload statistics can predict whether a server is likely to meet its service level objectives. In the case of predicted violations, the controller either spawns new server instances or activates “hot” standby ones, and copies or migrate data bins using a copy-duration model for planning. Trushkowsky at al. builds the server performance models using statistical machine learning (SML) on data obtained through *off-line* controlled experimental runs with steady state workloads. They claim that simple changes in workload will not affect the accuracy of these models, that however can degrade over time if the application or the underlying data change consistently, for instance when an individual request returns more data than during training. In this case, the off-line models would have to be rebuilt in production. The approach uses a linear classification model with logistic regression to predict whether a given workload mix (get and put operations) and intensity are likely to cause service level objective violations. The copy-duration model is obtained off-line through the linear regression of samples gathered by running copy operations on servers under different workload rates.

Sharma et al. [95] present Kingfisher, a cost-aware system to scale elastic applications. Kingfisher relies on several models that capture capacity, costs and reaction time concerns, and a linear optimization to solve the cost- transition-time- aware control problems. Pricing models, elasticity mechanism models (migration, replication, etc. on the different platforms) and server capacity (for different resource allocations) are all obtained empirically at design time. The controller uses the models at runtime to solve the integer linear program that accounts for both infrastructure and transition costs and derives appropriate control actions. Kingfisher assumes that applications have a multi-tier software architecture where each tier has its own quality of service requirements that must be met by provisioning sufficient capacity. Moreover, Kingfisher assumes the availability of a perfect forecaster that is defined as forecaster that knows the workload in advance, as well as the perfect estimation of the per-tier peak-workloads. To solve the integer linear program in reasonable time and for not trivial applications, Kingfisher employs a greedy heuristic with a bounded worst case.

Approaches based on heterogenous models Some approaches try to address the limitations of the different techniques by suitably combining heterogeneous static techniques.

Lim et al. [62] combine a *cloud controller* that manages the compute infrastructure with an *application controller* that manages the resource assignments for each application to satisfy a given performance goal. They propose to design application controllers based on classical integral controls that add and remove virtual machines based on average virtual machine CPU utilization. Integral control can be proved stable for a continuous control signal. Unfortunately the interface between the cloud and application controller consists of a coarse grained actuator, since one cannot add a fraction of a virtual machine, thus causing possible oscillations of the controlled system. To mitigate this effect, Lim et al. use a proportional thresholding technique, where higher and lower thresholds become smaller as system size increases, rather than a fixed target value for CPU utilization. They also use linear regression to model the relationship between the CPU utilization and the cluster size.

Jung et al. [46] focus on controllers that take into account the costs of system adaptation actions considering both the applications (for example the horizontal scaling) and the infrastructure (for example the live migration of virtual machines and virtual machine CPU allocation) concerns. Thus, they differ from most cloud providers that maintain a separation of concerns, hiding infrastructural control decisions from cloud clients. The controller relies on an *estimator* that uses 1) automatic off-line experimentation to build a *cost table* quantifying performance degradation for each type of control action and workload, 2) layered queue networks to predict the performance of each system configuration given a workload (model parameters are estimated offline), and 3) an ARMA filter to estimate the duration for which the current workload will remain stable (i.e., within a band B). The estimate of the duration of the stability of the workload is used to decide whether expensive (long term) control actions are worth or not. The controller searches through the graph of all possible control actions to find the sequence of control actions that maximizes a utility function that takes into account benefits and penalties expressed in terms of the application service level objectives, as well as the relative impact of all the control actions.

Summary considerations on static approaches Approaches based on control theory provide high guarantees in terms of assurance: Stability and dynamic properties of the controller and controlled system can be proved in rigorous mathematical terms. Accurate system identification further increases the control reliability. In particular, classic single-model approaches require simpler proofs and provide a clear separation between system regions where the control offers formal guarantees (viability regions) and where not. While, multi-model approaches are more flexible and are meant to cover wider viability regions. In Multi model switching and tunining, each controller behaves as in the single-model case, but no formal guarantees are usually offered at the level of overall control logic that switches them on and off. Moreover, the high assurance level of classic controllers is based on strict assumptions of the behavior of the controlled system that may not be always demonstrated in practice.

Rule and threshold based approaches can be verified, at least to some extent, formally. Verifying threshold based controllers is generally simpler than verifying rule based controllers, because threshold based controllers presents a less complex set of constraints, while full-fledged rule-based systems may have several conflicting and interdependent rules. Rules are commonly set manually under the assumption that they are able to capture main phenomena and characteristics of the system. They remain stable during the runtime and rely on the assumption that the system evolves only as predicted. Their event and condition clauses clearly identify the system viability zones that, however, remain fixed and may not are able to capture unplanned systems evolution.

The approaches based on different queuing networks are based on the assumption that the chosen model provides a sufficiently accurate representation of the considered system and that the system is inherently *stable*, that is, it does not present emerging or unpredicted behaviors. They also make several assumptions on the system behavior, its relevant components, and the statistical properties of the workload. Moreover, all the models require a set of parameters that are estimated at design time with no provision for adjustment at runtime. Simple models require to estimate few parameters, hence they are easier to configure, but may not be very accurate. On the other side, structurally complex models can be more accurate with respect to the system structure, but require to estimate more parameters, and thus demand extra time, and effort. Moreover, they do not always result in higher accuracy, because of the simplifying assumptions that enable analytically solutions. Richer models (for instance, layered versus plain queuing networks, or mix-aware versus mixed oblivious solutions) typically offer more flexibility, that is more possibilities of closer adaptation to the system, at the cost of higher complexity. In principle, they should be able to provide better assurance (at least in terms of a closer resembling representation of the modeled system). In practice the high number of parameters that need to be correctly estimated might attenuate the expected improvement.

Black-box surrogate models provide an assurance level lower than analytic models, because the quality of these approaches totally depends on properties that are difficult to formalize and measure at design time. For example, the quality of such models depends largely on the amount and quality of the data collected from the system runs, their distribution in the input/output features space, the training/fitting procedures adopted. Being dependent only on data collected at design time, these models may not reflect accurately the real production environments that may be different from testing environments. Designers must assume that the drift between testing and staging environment is negligible. Moreover, they assume that the system behavior remains stable at runtime. In a sense, these approaches leverages black-box surrogate model because of they capability to learn relations between system variables that are unknown or too complex to estimate precisely; however, they do not leverage this inner capability outside the design time activity, thus limiting the flexibility of the controllers.

Finally, approaches that combine heterogeneous models increase the flexibility with respect to the single static approaches, at the price of reducing the provided assurance level.

Dynamic approaches

Dynamic approaches rely on models built from monitoring data that can be automatically updated at runtime, or alternatively, use rules that are both discovered and adjusted while the system is running. We use the term dynamic approaches to indicate approaches based on models of the controlled system that are tuned at runtime, after system deployment. We refer to these models as dynamic models to stress the runtime nature of their construction.

These approaches privilege runtime flexibility over assurance. They produce an accurate representation of the modeled system by characterizing the system behavior through metrics collected from the actual system execution. Typically, they build an initial version of the model at staging-time from a set of training samples, and then update the model continuously at runtime while the system is running. Thus, they can adapt to situations that arise only at runtime resulting in high flexibility. However, they provide little support for analysis and formal assurance, and rely mostly on assumptions about the system behavior that may not be always easy to check statically and enforce dynamically. Thus they provide a lower assurance level than analytic approaches.

Different classes of dynamic approaches are characterized by the type of surrogate model they adopt, ranging from several forms of regression (for example, linear, quadratic, LOESS and Kriging) to machine learning techniques (for example, neural networks and reinforcement learning). Different model types imply different query and update strategies, to account for instance for the possibility of incrementally updating a model or the time and computational complexity of a complete re-training.

Dynamic approaches typically model either a single or a combination of system performance metrics as a function of some endogenous system properties that represent the system *configuration* in terms of both system characteristics, for example, number of allocated VM instances, CPU cores or threads, and exogenous factors, such as for instance the workload applied to the system or the influence of other services colocated in the same infrastructure. The approaches surveyed in this section often differ in terms of the considered metrics and the characterization of the workload features (for instance, workload intensity or service mix), but share some important features. They are highly flexible, and adapt easily to the measured system behavior, because of the ability to learn from and adapt to emerging behaviors, and this is extremely important for instance when the configuration or workload space is too vast for extensive exploration at staging time. They impose few requirements on the experience and knowledge of the modeled system functioning and behavior, because the samples required for training the models come from externally measurable system features.

The adaptive nature of dynamic approaches is both their main feature and their Achille's heal: The ability of constantly change and adapt makes them particularly well suited to highly dynamic systems and, at the same time, makes it hard to assure the quality of the resulting system, and to estimate the correct partition of model training effort between staging time (*bootstraping*) and on-line learning.

In the following, we present the main approaches of this category distinguishing between approaches based on surrogate models and approaches based on machine learning.

Approaches based on surrogate models Surrogate models are build from sample executions of the modeled system, and describe the behavior of a system in terms of relations between input and output features that can be continuously updated with monitoring data. They are commonly used to describe either the steady-state behavior or the mid-to-long time horizon behavior projections. Self-adaptive controllers use surrogate models to predict the near future, given both the current and the estimated values of the input features. Some controllers use surrogate models also to support optimization procedures that explore the system configuration space to find the most suitable system configurations. Less commonly, surrogate models are used to simulate and track the evolution of the system state under possible control actions, to plan for the most suitable ones. Such control strategy aims to maximize the control utility over a *receding* time horizon.

Bodik et al. [14] use statistical machine learning, and in particular smoothing splines and local regression (LOESS), to build performance models of the controlled system. Bodik et al.'s models represent response time as a function of workload intensity and system configuration. The controllers increase the robustness and the adaptivity to changes, by means of *model management* techniques (i.e., online training and change point detection) that update the model, track its quality and eventually rebuild it from scratch. The models are trained with data obtained online from the production environment. The controllers are conservative: They start from the maximum allowed allocation of resources, and decrease the allocation, while incrementally learning optimal configurations, to minimize service disruptions in exploration mode. Building models entirely from online samples simplifies the training phase, but may result in slow convergence of the models to new and possibly temporary system behaviors.

Approaches based on machine learning Machine learning techniques are commonly divided in model-*based* and model-*free* techniques, depending on the use of models. Both classes of techniques are exploited to build self adaptive controllers. The most popular control solutions that refer to model based techniques use artificial neural networks (ANN), while popular model-free techniques use reinforcement learning (RL) and clustering applied to the discovery of control rules. In model based solutions, the accuracy of the results depend on crucial choices such as the model structure and the training data, thus no a priori guarantees can be enforced. In model-free solutions, the level of assurance depends on the learning rate, the instability/evolution pace of the controlled system and the size of the action-configuration space, which is proportional to the amount of possible control actions.

Artificial neural networks use training samples to build a model of system dynamics that can predict the system reaction to different inputs. The structure of the network and the quality of the training data are critical to the performance [68] and must decided by the designers off-line. After the initial supervised training that sets all the internal parameters, artificial neural networks can be updated on-line by a back-propagation procedure based on punishment-reward concepts. Maggio et al. [67] implement a neural network to control the amount of resources allocated to process at the OS level that guarantees a given service level. Although the context is different from Cloud computing (where artificial neural network based controllers have not been used yet) the basic concepts of modeling and dynamically allocate resources are similar. Islam and coauthors [42] train a neural network with samples of CPU usage taken from running virtual machines under predefined workloads, and use the neural network to predict the level of CPU usage at runtime in face of unknown workloads.

Controllers based on reinforcement learning learn directly optimal control policies, that is, the best set of actions to apply whenever the system enter a state, and thus do not require a model of the system. At runtime, reinforcement learning-based controllers adopt a trial-and-error learning strategy and apply (at least at the beginning) random actions. Effects on the system, and controllers utility function, resort either in action reward (if the actions increased the utility) or punishment (if the actions damages the system, thus lowering the utility). Reinforcement learning solutions suffer from poor scalability in the action-state space and from long convergence rates. To alleviate these limitations, Li and Venugopal [60] propose a distributed implementation of reinforcement learning based self-adaptive controllers. In this solution, each processing node, i.e, a virtual machine, is an independent entity and incorporates a local controller that runs a Q-Learning algorithm. Local updates to the model are pushed to the other controllers via a distributed hash table, so that collaborating controllers can learn the state-action-reward model quickly. At run time, each controller takes scaling decisions based on the shared model and aims to maximize the reward function while model updates are continuously published.

Xu et al. [118] propose a dynamic approach based on fuzzy rules. The controller applies fuzzy modeling to learn the relationship between workload and resource demand, and uses clustering to update the rules at runtime. The controller is organized in two levels: At the lower level, local controllers are associated to physical resources and decide about the resource needs of the virtual applications deployed on the node; at the higher level, a global controller receives all the resource needs from the local controllers, and solves the resulting global optimization problem to decide the final resource allocation. Local controllers estimate the needs of resources at regular intervals for each virtualized application by means of fuzzy inferences: Each controller receives workload data, *fuzzifies* them, triggers the fuzzy rules, and finally produces the output *crispy*. At the same time, controllers analyze the monitoring data to derive new fuzzy rules, adapt existing ones, and remove not optimal rules from the knowledge base. Fuzzy rules are obtained by filtering and clustering raw monitoring data as they reach the controller: Data are filtered if they refer to mappings that lead to service level agreement violations, and then are clustered based on the density of surrounding data points. Eventually, a single rule is associated with each cluster. The controller defines an initial set of clusters, thus a set of fuzzy rules, offline using data from staging experiments, and updates the fuzzy rules at runtime, by adapting the size and number of clusters.

Summary considerations on dynamic approaches The assurance of surrogate-based self-adaptive controllers relates to (1) the ability of the models to accurately represent system behavior also in the presence of noisy and missing data, that is, when the quality of data interpolation and regression decreases, (2) the speed of convergence of the learning process, and (3) the accuracy of the quantification of the uncertainty of the model predictions. For example, models that are updated online and frequently, that do not need a large training set, and that can provide an accurate measure of confidence for their predictions, provide higher assurance than models that are updated infrequently and cannot provide any confidence interval for their predictions. Controllers that continuously monitor the quality (accuracy, prediction error, etc.) of the models, and account for completely rebuilding of the models whenever necessary can adapt faster to emerging system behaviors than controllers that merely update the models with new data.

Control solutions based on machine learning techniques are in principle the most flexible solutions, since they can learn any kind of relations by either directly modeling the system or capturing the optimal control policy. However, this flexibility come at the cost of the difficulty of proving stability, convergence or any other properties important for control purposes. In particular, artificial neural network and reinforcement learning solutions do not provide any automatic means to evaluate the goodness of their fit nor their predictions, and designers have to either believe in them or not, and employ some external mechanism to track their error to retrain them whenever it is possible.

Both surrogate-based and machine learning solutions can be useful where the complexity of the controlled system makes it infeasible the construction of any satisfactory analytical solution or white box model, or when designers have no a-priori information about the behavior of the controlled system.

Hybrid approaches

Hybrid approaches combine design time and runtime techniques and aim to conjugate the flexibility of models used at runtime with the assurance gained with models used at design time. Static and dynamic approaches can be combined in many different ways, resulting in different blends of assurance and flexibility. We distinguish two classes of hybrid approaches depending on the adopted merging strategy: (1) approaches that augment static solutions with learning and (self-)adapting capabilities, and (2) approaches that complement dynamic solutions with static models to modulate the effects of learning on the control behavior.

Approaches that augment static technique with learning capabilities aim to increase the level of flexibility while maintaining a high assurance level. Approaches that combine dynamic solution with static model aim to improve assurance while limiting the loss along the flexibility dimension.

Static approaches augmented with learning capabilities Static approaches augmented with learning and self-adapting capabilities increase the flexibility of the underlying static technique while maintaining high assurance level. They try to augment the size of viability zones by allowing the control solutions to deal with unseen and emerging behaviors that may differ from the design time assumptions. Augmented static solutions either relying on static models with online parameters tuning, or on static models that are rebuilt while the system is running.

Approaches that augment static models with online parameter tuning are based on a core static model whose parameters are updated at runtime using monitoring data. The update and learning processes proceed in parallel with the controller activities. Woodside, Zheng and Litoiu [117] propose a model-based feed-forward solution centered around a layered queueing network model of the system whose parameters are recomputed online. The controller uses the model to predict the system performance depending on the monitored workload, and to optimize the system configuration, in terms of server configurations and resources allocation. The controller adapts the parameters at runtime by means of an Extended Kalman Filter: The filter keeps updating the parameters until the residual error is below a threshold, and in this way the controller relies always on an accurate model of the system. The tracking filter greatly improves the robustness of the control algorithm in the presence of parameter drift. Moreover, extended Kalman filters have proven optimal properties when the relations between variables are linear, and are expected to have near-optimal properties if nonlinearities are involved depending on the local properties around the operating point.

Urgaonkar et al. [106] propose a dynamic capacity provisioning model for multi-tier Internet applications that uses queuing networks to determine the provisioning of resources for each tier of the application. Differently from the previous work, Urgaonkar et al. use online monitoring data to estimate the session arrival rate, the session duration and other parameters that are fed to the queuing network for the predictions. The proposed controller combines predictive and reactive methods to determine when to provision resources, to cater for respectively long-term / cyclic and short-term / unpredicted variations in the application workload. The controller computes long-term provisioning with the queuing network model where each tier is modeled as G/G/1 queue, and tiers are linked with replication factors to describe how the workload demand is distributed inside the controlled system. The controller deals with short-term variations of the load by means of a *sentry* component that implements admission control policies.

Singh et al. [98] use a queueing network to model the system as well, but they rely on mix-aware provisioning techniques to handle non-stationarity in workload mix and volumes. The controller employes k-means clustering to automatically classify the workload mix and uses the queuing model to predict the server capacity and support configuration optimization. The “workload class” is the parameter estimated online by the controller that keeps the model up to date. The initial clustering is computed off-line following an iterative and empirical process. On-line clusters are adjusted (split/merge) whenever the error of the estimated cluster predicted mean service time is greater than a given threshold with respect to the mean service time monitored by the mix-determiner. Maximum number and size of clusters are specified beforehand. Similar to the other approaches in this group, Singh et al. assume that the system can be modeled as a pipeline of independent tiers, for which per-tier service level agreement can be defined, and per-tier demands can be derived from the incoming one. The clustering allows to determine precisely the different types of requests in the workload improving the accuracy of results form the queuing model.

Approaches that rebuild the model at runtime use a control model that is modified or completely rebuilt at system runtime, either periodically at a fixed time interval or whenever some indicative metric (for instance, a prediction error) crosses an acceptable threshold value.

Jung et al. [47] propose a rule based approach, where rules are automatically generated by means of a machine learning process. The controller uses the rules as in static approaches: It monitors system variables, for example, the workload, and triggers the control rules when necessary, for instance when there is a match with the workload intensity, to change the system configuration. The controlled systems are modeled by means of layered queuing network models whose parameters are estimated offline, at design time. The controller then uses the layered queuing network models in a two step discovery process that is carried out at runtime in parallel with the control loop: The controller (i) randomly chooses a set of input workloads, searches for the corresponding optimal system configurations, and encodes the results as rules, and (ii) interpolates all the data via a Bayesian classification algorithm that derives a decision tree saved as policy. The decision tree covers the whole configuration space – thus not only the rules obtained in step (i) – and the learning algorithm can be configured to prune or merge, similar subtrees and configurations to simplify the rule set. The process produces a new decision tree with a finite number of leaves, i.e., system configurations, with a high degree of predictability and verifiability because the new tree encodes all the possible system configurations a priori enabling further decision to be taken at *business* level. The optimization procedure is based on a heuristic gradient search and considers both the system configuration, i.e. the number of replicas for each virtual machine, and their placement on a set of physical nodes. It assumes that the system utility monotonically decreases as resources are deallocated. The quality of the control, measured in terms of utility, depends on three critical factors: The model accuracy, the number of workloads considered during the optimization, and the "compactness" of the decision-tree, as more compact trees are less accurate.

Quiroz et al. [89] propose a decentralized online clustering approach to detect patterns and trends in resource demands for jobs in Grid systems, and use this information to optimize the provisioning of virtual resources. The control is fully decentralized: In each control window, the clustering algorithm analyzes the incoming jobs and produces a number of target virtual machine classes. The number of virtual machines that must be provisioned is proportional to the volume of the incoming job for each class. Jobs are either assigned to the available virtual machines as they arrive or wait for the right virtual machine to start. Eventually, each local controller triggers the creation of new virtual machines to process the waiting jobs. Controllers rely on a model to estimate application service time based on Quadratic Response Surface Model (QRSM). The QRSM is fitted using long-term application performance monitoring data, and it is used at runtime to provide feedback about the quality of the clustering, i.e., the appropriateness of requested resources for the incoming jobs and their ability to meet QoS constraints. Given the actual resources, the QRS model is used to predict the response time for the estimated workload. This prediction is compared against the quality of service requirement and the controller uses the model to adjust the class attributes computed by the clustering. The controller then uses the quality measures obtained from the QRSM as an oracle to re-trigger the evaluation of the clusters with the decentralized online clustering algorithm.

Dynamic Approaches with Static Fall-back The lack of assurance of purely dynamic models derives from the dynamic nature of the data used to build and tune the models, that cannot be statically verified by definition. It is difficult to provide assurance proofs of controller behavior for approaches based mainly on runtime measurements, that heavily depend on the availability and quality of the data collected at runtime. Dynamic approaches with static fall-back aim to improve the accuracy of these models by complementing the dynamic model with analytic approaches that might provide a reasonable alternative in the presence of low quality of the prediction based on the runtime model.

Malkowski et al. [70] propose a *multi-model* controller that combines the horizontal scale controller originally developed by Lim et al. [61] as the static approach with an *empirical model* obtained from runtime monitoring data. The empirical model uses a throughput vector space (i.e., a list of throughput values, one for each application interaction-type) to represent the combination of configuration, workload, and performances achieved by the system in a 30-seconds time frame. The empirical model is used to find the smallest (cheapest) system configuration that satisfies the service level objectives, and is located within a threshold value in terms of Euclidean distance in the throughput vector space with respect to the predicted workload. In other terms, it selects the smallest configuration among the set of *visited* configurations that were able to withstand a predicted workload intensity (or a comparably “close” intensity) without violating service level objectives. The controller switches to the static approach when the empirical model cannot find a visited configuration within the distance threshold.

Tesauro et al. [101] employ both static and dynamic techniques in an approach that combines the strengths of reinforcement learning, artificial neural networks and queuing networks. A static technique is used when the reinforcement learning is in learning mode; in this period, the controller resorts to the queuing networks to control the system and collect training data at runtime. To speed up the learning, the data are used to train a non-linear function approximation, in this case an artificial neural network, of the Q-function that is used to obtain the learning rewards. Tesauro and co-authors represent the Q-function using neural networks instead of traditional look up tables to encode the state-action rewards. Artificial neural networks interpolate the collected samples and reduce the need of large training set, improving the controller scalability. By combining reinforcement learning and queuing networks, controllers can avoid poor performance during the training activities, because all the data are collected using the queuing network policy that provides an acceptable quality level of the control with respect to the random control actions of the reinforcement learning exploration. At the same time, controllers can improve their accuracy because steady-state queuing models are unable to take dynamical effects into account while the reinforcement learning can take into account dynamic effects such as transients and switching delays. Once the reinforcement learning is ready, the controller releases the queuing network and adopts it. In this period, parameters of the queue network are continuously updated based on measurements of system behavior, and when necessary, the controller can switch back to it and start the learning process again.

Summary considerations on hybrid approaches Approaches that augment static technique with learning capabilities share the adoption of an analytic system representation (either queuing networks or layered queuing networks) whose structure remains unchanged. They achieve a limited degree of adaptability by estimating one or more parameters online. This caters for situations in which there is a need for minor adjustments with respect to design time expectations on system behavior, but the limits imposed by the initial queuing network model prevent the control from adapting to any possible emergent behavior related for instance to non-modeled system bottlenecks. More flexible are approaches that rebuild the model at runtime by combining some static model, either a linear queuing network or a response surface model, with a learning/discovery mechanism that changes the control logic, rule- or clustering-based, respectively. The static model is derived from design time knowledge or long-term application historical data, and provides the fixed reference for the derivation/optimization and evaluation of the runtime-generated control logic. In a sense, the static model defines fixed boundaries for the controller behavior that make it predictable (i.e., within the boundaries), while the runtime adaptation aims at optimizing the control with respect to the measured system behavior.

Approaches that combine dynamic solution with static models achieve high flexibility by using empirically obtained data to model emerging (and possibly unexpected) behaviors such as cross-layer data-interdependencies. High flexibility aims at a more precise representation of the system behavior geared towards the realization of *more efficient* (i.e., in terms of resource usage, service level objective violations) controllers.

To compensate for the dynamic nature of black box models, and their dependency on possibly high varying runtime monitoring data, analytic approaches (typically queuing networks) are used as a safety net to constrain the controller actions. In a sense, analytic models provide the base controller behavior upon which dynamic solutions can improve.

Summary considerations

We summarize the distribution of the presented approaches in the diagram of Figure 3.1. Approaches are identified by the name of the first author and the citation number, and the diagram shows them over the flexibility/assurance space. Over the horizontal axis we identify the three main groups, static, dynamic and hybrid. The (red) grid partitions the space according to the level of flexibility in the horizontal direction, and the modeling choice adopted (or rules) over the vertical direction. Our work is highlighted by means of tick black lines, and the work presented in this thesis is highlighted by means of a shaded area.

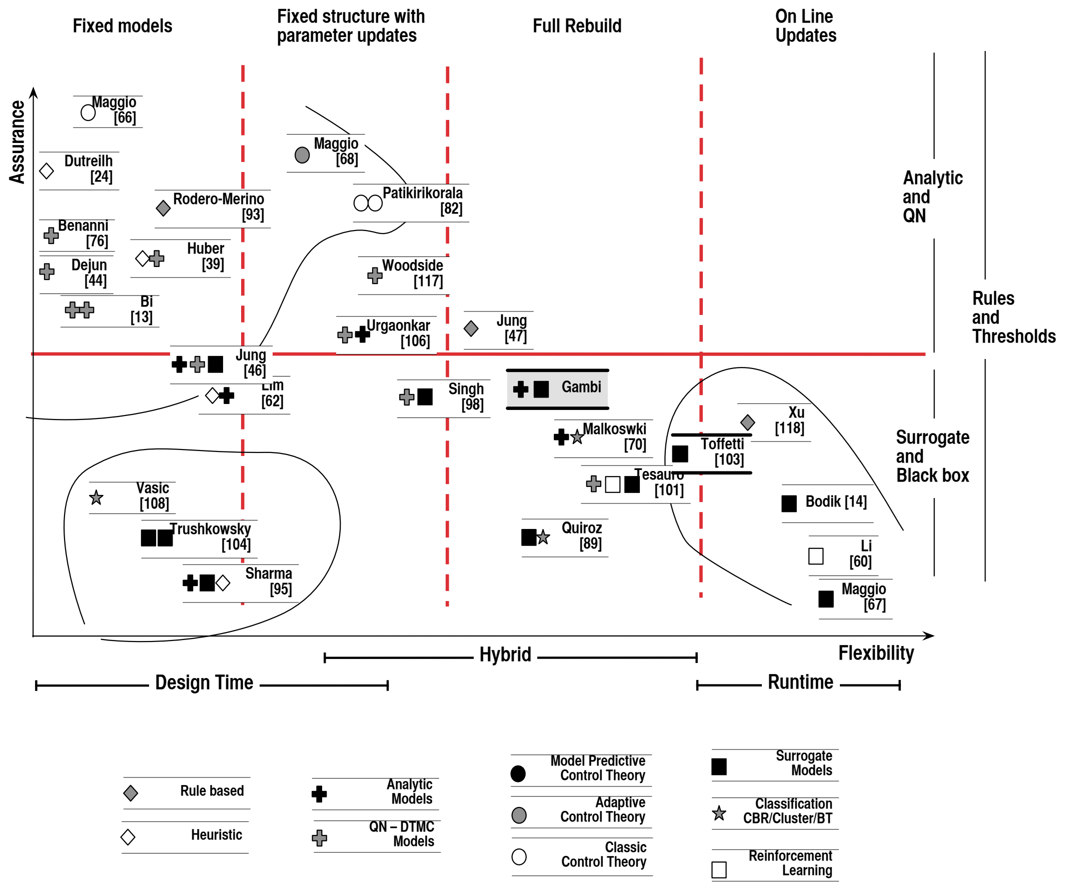

Figure 3.1. Controllers for the Cloud

Kriging based controllers compared to the other approaches are flexible, trained by full rebuilt of the model and provided limited assurance when used alone [103]. To improve the assurance level of the Kriging based controllers with couple Kriging models with analytical models, and this results in the final positioning of our work in the diagram. We present more details on the combination of Kriging and analytical models in the next chapter.

3.2 Surrogate models for predicting performance

In this section, we focus on work that proposes to use black box and surrogate models to study the behavior of software systems and support the usual activities belonging to Performance Engineering (PE). This work is in contrast with the more traditional approaches that exploit white box models, such as queueing networks, Petri nets, and other architectural models. In particular, here we consider only those models that describe the average system behavior at steady state, i.e. time invariant models.

Surrogate models are known is several communities, such as Engineering, Aviation, Statistics and Physics. For instance, Gramacy, Lee and Macready propose gaussian process trees in the simulation of fluid dynamics to design NASA reentry vehicles [35], while Mariani et al. use artificial neural networks to optimize the design of multi-processor systems-on-chip [71]. The survey of Wang and Shan provides an excellent overview of the different uses of surrogate models in engineering [110]. However, their use in the fields of Computer Science is still limited: They are mainly used in computer simulators and machine learning problems. Only in few cases the authors recognize their modeling power and use surrogate models outside their traditional applications.

All the considered work shares the basic motivations for using black box in the context of performance prediction: Software systems are too complex to be accurately described with white-box models, and the necessary knowledge to build them may not be easily accessible. Using data taken from running the system under study may lead to more precise and accurate models than the one obtained assuming predefined system behaviors, and, surrogate models are powerful tools that can capture the multi-variate nature of the relationships between the system configurations and resource allocations, with the system responses that typically real systems show.

In the context of performance prediction, surrogate models are typically used to support two main activities: design space exploration, on one side, and change-point identification, on the other side.

Design space exploration

During the design activities, developers must understand the behavior of the system under development in several situations: Systems usually come with a series of configuration options and parameters that have to be set before the release, and that impact on the final behavior of the system. For example, different configurations may be suitable only for a subset of possible execution contexts, and may require specific re-tuning in order for the system to fulfill user requirements.

In this context, developers use surrogate models to capture the behaviors of interest of the system and to support an exploration of the space of alternative system configurations. For example, developers can study how the performance metrics change along with the changing configurations by visualizing the response surfaces. This can yield insights into the effects on performance indexes of the different configurations under various operating conditions, and can be used by designers to evaluate design decisions and cost tradeoffs. Moreover, surrogate models enable automated exploration of the configuration space aiming to find optimal configurations for a given set of user requirements and some available information about the target environment or the usage scenarios. They can be coupled with Genetic Algorithm (GA), simulated annealing, or gradient search methods in model-assisted optimization methods [71].

In the domain of software system performance prediction, notable work was done by Happe et al. [36; 111; 114] that use multi adaptive regression splines (MARS) to model the behavior of message oriented middleware. In their work, Happe and co-authors use MARS to correlate how the system performances change while changing configurations such as the average message size, the message arrival rate, the number of allocated threads, and the amount of physical resources allocated to participating nodes. The same authors also compare the abilities of regression splines, Kriging models and other surrogate models in predicting performance of software systems [113]. Regression splines and Kriging models provide comparably accurate predictions in most of the cases.

Westermann and coauthors investigate also the use of surrogate models, in particular regression splines and Kriging models, in automated Goal-oriented performance test [112]. The idea is to leverage surrogate models to automated performance tests of software systems, in line with the traditional use of surrogate models for design optimization.

Zhang et al. [120] compare how Bayesian networks and artificial neural networks perform in modeling the response time of Web services with respect to resources allocation. They confirm the general intuition that ANN have more expressive power than bayesian networks, but are far more difficult to manage; in contrast, BN are fairly long to build.

On the same line, Correa and Cerqueira [19] compare several models commonly used in statistical machine learning. They compare decision trees, several implementations of Bayesian networks, and support vector machines with radial kernel in the study of Map-Reduce systems. Correa and Cerqueira conclude that accuracy of SVMs over-performs the one of other models, at a cost of training time that is three order of magnitude longer.

Ganapathi et al. [32] model the multivariate relationships between the query features available before runtime with the actual performance features measured using the Kernel Canonical Correlation Analysis, a kernel function technique taken form the machine learning community. This work is notable because the proposed model deals with multiple outputs at the same time.

Similarly to the above mentioned work, we use Kriging models to capture the relations between system performances, resource allocations and system configuration. Compared with the models proposed so far, Kriging models balance better the trade-off between accuracy and training time, as they are faster to train than SVM and BN, they are easier to manage than ANN, and they reduce the risk of over-fitting that may appear in models with predefined structure as full quadratic models.

The work summarized here focuses on the accuracy of the predictions more than the effort in setting up the models. The different approaches can tolerate long training time, as they are not explicitly used in the context of on-line control of the software systems that they model. In theory, many of the proposed approaches can be used in the context of model based control of elastic application over Cloud IaaS, either as global models or in conjunction with other models. However, the actual instance of surrogate models that can be used depends on the requirements of each case and application, especially the frequency of the controller.

This thesis is the first that proposes the use Kriging models for performance prediction of software systems. In fact, in most of the related literature, Kriging models are used to model expensive computer simulations of physical systems, such as aircrafts or microchips, more than software systems themselves, as discussed in Section 3.3.

At the time of writing, the only work that we are aware of and that propose to use Gaussian processes is the work by Aboulnaga et al. who use Gaussian processes to predict query response times as a function of query mixes, database configuration parameters, and allocated resources, with the aim to support query schedulers in running queries [1; 96] Notable is the combination of multiple Gaussian process models to obtain the sought predictions: A high level Gaussian process model maps query *logical* characteristics to query response time, while a low level Gaussian process model maps database configuration and resource allocations to the hyper-parameters of the high-level model.

Change point identification

Other than exploring the configuration space of software systems, surrogate models can be used to identify *interesting* operational points such as optima, crossover points, or the bounds of the effective operating range for a particular design configuration.

For example, Shivam et al. [97] use a polynomial models in planning stress-test experiments to find peak rates and saturation points of file servers. They combine a binary search algorithm with the polynomial model to identify the maximum input rates that the system can sustain without violate constrains on response time for a given configuration.

Similarly, Osogami and Kato [78] use full quadratic functions to model the response time of multi-tier Web systems and drive the performance tests

In these approaches, models guide the design space exploration towards the optimal (or interesting) system configuration: The model interpolates the data collected so far and suggests the next test cases.

Differently, Leitner et al. [57] use artificial neural networks in the context of Web service compositions to predict if the instances of the service composition that are been currently executes would respect their SLA, given the monitored data. The ANN is invoke repeatedly as the execution proceeds and more data are collected: At every check-point identified by the developers, the ANN predicts if the final QoS of the entire composition for the running instance will comply with its SLA.

Kriging are well suited for implementing model based optimizations of systems by leveraging the uncertainty measures associated with predictions as show by Jones et al. [45] and Forrester [27]. In theory, we can leverage Kriging models also to plan the optimization of the design of the experiments while studying the behavior software systems, thus reducing the time to build an initial model of the system to control. This is interesting, but out of scope of the thesis, and we do not consider it any longer in this document.

3.3 Control solutions based on Kriging models

This thesis proposes Kriging models to describe the steady state system behavior of software systems and use them in a model based self-adaptive controller. This is the first work proposing such a control based solution. Outside the context of Computer Science, in particular in the context of theoretic control of physical systems, other authors proposed to use Gaussian processes. Differently from our proposal, they aim to capture the dynamics of the system instead of its steady state behavior, and use these models to reason on the transitory aspects of the systems. These authors rely on the continuity and predictability proper of physical systems to develop their work, and whenever necessary, they resort to first principle models to define the controllers.

Kocijan et al. [51] proposed to use Gaussian process to model system dynamics and to use the model in the context of predictive model based control: The controller uses the model of the system evolution to plan for the optimal control strategy via a limited lookahead strategy as in the case of optimal strategy planning. The model is trained once off line by stressing the system along some predefined “trajectories”.

This work was later improved by Petelin et al. [88] who provide the controller with a self-adaptation loop for the model: At each iteration the controller updates the model (without the hyper-parameter optimization) with a subset of the new data that is chosen to maximize its information gain, as in the traditional EGO algorithm proposed by Jones et al. [45]. In particular they code the control signal to apply in the system as the one that minimizes the model variance as estimated by the model.

Nguyen-Tuong et al. [77] presented an approach to combine multiple “local” Gaussian process models in the context of learning the dynamics of a moving a robot arm; the authors identify the challenge of having too many training data when modeling such a complex system, and solve it by clustering the data along the locality dimension and creating a set of local models.

Notable work is done by Deisenroth and co-authors that apply Gaussian process in the context of robotics to learn the behavior of physical systems and control them [21]. In particular, Deisenroth et al. combine Gaussian processes and Reinforcement Learning techniques [22]. The intuition is that Gaussian processes can be used to learn a probabilistic model of the system behavior and optimize the Reinforcement Learning exploration of the state-action, thus reducing its limitations and costs [23]. In a later work, they improved the original proposal with Gaussian proces based smoothing and filtering, showing that hidden variables in the systems can be easily capture by a Gaussian process filter. In this setting, the Gaussian process filter outperforms traditional Kalman based filters [20].

The work summarized in this section represents a good basis for promising future research and motivates more research in modeling the dynamics of adaptable software systems. Models of the systems dynamic are more accurate than steady state models, especially at short time intervals, and, in a sense, suit better traditional approaches to control, like the ones proposed in classic control theory. Controllers can exploit models of system dynamics also to plan control strategies that account for the transitory periods between the application of several control actions, which may have long durations, as in the case of Cloud based elastic systems[4]. However, modeling the system dynamics introduces non trivial challenges related to models training, model update and management of time dimension that still remain open [88]. For example, designers need to discover and capture the elasticity of the systems, therefore they need to carefully plan staging time experiments that stress the systems in multiple ways [43]. Moreover, designers cannot assume anymore the continuity of systems dimensions and states, and they cannot resort to first principle models during the design of controllers.

[4]In our context, workload may change in the order of seconds, but the implementation of the control actions (i.e., add and remove VMs) may take up to five minutes [59]

Chapter 4

Kriging-based Self-Adaptive Controllers for the Cloud

In this thesis, we take the point of view of the service providers that stipulate SLA contracts with final users that impose commitments on the behavior of providers applications in terms of QoS. Service providers need to maintain an acceptable level of services to avoid violations of SLA by dynamically provisioning their systems with enough resources despite fluctuations in the workload, instability of Clouds platforms, and the complexity of elastic applications.

We present a novel solution to the problem that is centered around model based self-adaptive controllers that plan fast and safe control decisions on-the-fly: Models capture the relations between important qualities of the systems as a function of the system configurations, environmental conditions and other relevant metrics. Controllers leverage the models to identify risky situations and plan suitable control actions to maintain the QoS of the controlled systems to acceptable levels while using the smallest possible amount of resources to run them.

We target model-based self-adaptive controllers because they can deal with complex systems, recurrent working conditions and unexpected situations, ever changing environment and system evolution, that are non-trivial challenges pertaining to dynamic resource provisioning.

The ability of controllers to maintain the expected QoS depends on the adopted modeling choice: the accuracy and robustness of the model limits the effectiveness of controllers; the time to query the model limits the control frequency; and the time to update the model limits self-adaptiveness and flexibility. The novelty of our solution is the use of Kriging models as core elements of these controllers to implement effective, efficient and robust control policies.

Kriging models are statistical, multidimensional models that can accurately capture the behavior of systems, provide timely predictions paired to confidence measures, and that can be retrained on-line to implement self-adaptiveness by means of model adaptation. We sample running systems and we let Kriging models learn the relations between their behaviors with the allocation of resources and environmental conditions. Kriging models balance the trade off between model accuracy, query time, and update time that is central to efficient and effective model-based self-adaptive controllers, and provide a measure of uncertainty on their predictions that enables the definition of robust control policies. For these reasons, we believe that Kriging models are the most suitable choice.

In this chapter, we discuss the main reasons that lead us to formulate this hypothesis and we present details on the proposed solution. In the next chapter we will report the experimental results that prove the validity of the hypothesis.

During the validation, we consider one instantiation of the dynamic resources allocation problem in the context of Cloud computing. In detail, we consider elastic applications that run on IaaS Cloud platforms that enable service providers to allocate or deallocate resources at the granularity of virtual machines.

4.1 Model-based self-adaptive control

In general, model-based controllers use abstract representations of systems, their behaviors, or both, via models to plan for suitable control policies. For example, a controller may use a model of the software architecture to reason on the deployment of components on processing nodes, a queueing model to reason about the allocation of resources to software components to guarantee a predefined response time, or it can use them together.

Models relate control objectives to controlled and uncontrolled variables, and disturbances. For example, a model that describes the end-to-end performance of virtualized systems as a function of the system configuration and workload can relate the average system response time (the control objective) to the amount and type of virtual machines allocated to the system (the controlled variable), the amount of requests waiting to be served (uncontrolled variable), and the intensity and mix of requests of incoming workload (the disturbance).

Controllers can *react* when control objectives are not fulfilled, for example after the occurrence of an SLA violation, to take the system back to a good state, or can *pro-*act and maintain the control objectives fulfilled, that is, avoid the occurrence of SLA violations. In this thesis, we pursue the importance of proactive control and we design controllers that use models to achieve pro-activeness by means of two main activities: early warnings and configuration planning.

Models yield predictions to generate warnings about risky situations that may lead to SLA violations, and eventually, may trigger controller actions to achieve control objectives. For example, a model can estimate if the current allocation of resources to a server is enough to sustain the monitored workload, and the controller may plan changes of the allocated resources if the estimation to prevent an estimated violation.

If a prediction of the workload is available, it can be combined with the model to obtain long term warnings.

Timely availability and accuracy of the predictions are of paramount importance to achieve effective control. Inaccurate predictions may cause the controller to intervene when not necessary, and may cause oscillating behaviors, waste of resources, or even worse, they may cause the controller not to intervene at all, thus leading to violations of SLAs. If predictions are not promptly ready when needed, the controller may have a *drifted* representation of the system status, and it may be late in implementing control actions, thus decreasing their effectiveness.

Once the warnings trigger, controllers leverage models to plan a suitable control strategy. Controllers use models to simulate the *expected* behavior of the system in case one or more control actions are implemented: They simulate changes of the system configuration and check if the resulting behavior would comply with the SLA. For example, a model can simulate the system response time with a variable allocation of computational resources to the system, under a constant workload, and the controller can chose the resources allocation that maintains the predefined response time while minimizing the amount of resources used. As before, configuration planning demands strong requirements on model accuracy and the time required to obtain predictions.

4.2 Why Kriging models?

The problem of autonomic control for dynamic resource allocation presents some non-trivial challenges that pose stringent requirements on the models that controllers exploit. For example, natural fluctuations in the workload, sudden spikes in the service demands, and variety and complexity of the controlled applications, demand for models that should be accurate enough to capture the behavior the controlled systems under various workload regimes, fast enough in providing predictions thus avoiding to impair controllers reactiveness, generic to model heterogenous systems, and scalable with system complexity.

Clouds introduce additional challenges that demand for new requirements to be met by the models. For example, Clouds are shared among several users and their concurrent activities may result in uncontrolled changes inside Clouds infrastructure, like the deployment of new virtual systems or the shutdown of physical nodes, that may impact on the external behavior of the running systems. Clouds use heterogeneous physical resources to run virtual machines, and this may result in a slightly different behavior if instances of the same virtual machine are run by different physical servers. Clouds are opaque by design, that is, final users have no information about their architecture, nor internal state. These challenges demand for robust models that adapt to transient or unexpected behaviors of the controlled systems, tolerate inaccurate, noisy or imprecise data about controlled systems, do not rely on extensive knowledge about Cloud internals, and deal with the uncertainty associated with any opaque system.

Designers of autonomic controllers pose other requirements on the models: In principle, models should derive from a solid theory, and their capabilities should have been proven before. They should be easy to setup and configure during the development of the autonomic controllers, and their interpretation should be intuitive during testing and validation activities.

In the previous chapter, we saw that approaches and relative models proposed so far in this context have strong limitations that may result in inapplicable, inefficient or ineffective controllers. Instead, we propose an approach based on Kriging models that have capabilities to tackle those challenges more suitably. Kriging models enable us to design self-adaptive controllers that improve on the state-of-the-art solutions proposed so far. In this section, we introduce and discuss the reasons that make Kriging models such a sensible choice for the problem of model-based self-adaptive control in the Clouds. In the next chapter (Chapter 5), we present the experimental results that support our hypothesis

Kriging models are global, i.e., they span over the whole input space, and multi-modal, i.e., they may be non-monotonic, therefore they can capture in one solution the behavior of systems across different workload regimes. This is different from classic regression models that are commonly preferred for local predictions.

Kriging models belong to the family of black-box models, that is, they model a system only based on its *external* behavior. For this reason, they are more suitable than white-box models to capture the behavior of systems running, especially in Clouds, where the knowledge about system internals is generally not available and accurate white-box models are difficult to set up and maintain. Even if Kriging models do not require additional knowledge about the system, they can accommodate any additional knowledge about the system behavior whenever is available, and increase their quality. Moreover, being based purely on data, Kriging models are generic, therefor they can model the various systems running in Clouds with small or no effort. In practice, the complexity of Kriging models and the effort to manage them depend on the amount of samples used for the model fitting more than on the complexity of the modeled systems. Thus, Kriging models scale well while adding dependent variables, but suffer when the amount of samples increases.

This results in a critical trade-off where we need to balance the accuracy of the model, which increases as the training set increases, and the time for querying it, which increases with the size of the training set.

Like other black-box models, the accuracy of Kriging models depends on the choice of the independent variables, i.e., the model *features*, and the training samples. In this work, we assume that the most important independent variables have already been identified and we build models using them. However, as explained in the related literature, one can use Kriging models also to perform a sensitivity analysis on the available systems' metrics to decide on the best features to use in each model. This is possible thanks to the intuitive interpretation of Kriging models that is based on the value of their hyper-parameters. In fact, the hyper-parameters associate different weights to the input dimensions, and in this way, the model captures the relative importance of each input feature. This behavior is very different form any strategy based on the standard Euclidean distance that treats all the input dimensions at the same way.

We empirically discovered that the amount of data needed to build a Kriging model with an accuracy sufficient for controlling elastic applications is *relatively* small, and this results in an easy-to-manage model that can be retrained on-line [31]. Similarly, we empirically discovered that Kriging models provide predictions in milliseconds, thus they are suitable to be used on-line within the control loop of autonomic controllers [103].

Kriging models compute the values of the predictions by weighting the values of the training samples modulo their mutual distance: If the training data cover the sampling space sufficiently, the quality of the model will be high. The accuracy of the model depends also on how noisy the training data are: For simulated systems, where there is no noise in the data, Kriging models provide a perfect interpolation; for real systems, where the monitoring data can be in general very noisy, Kriging models can balance data interpolation and regression, thus reducing the over-fitting of the model.

Kriging models are supported by a solid theory based on statistics. Their interpolation capabilities have been proven effective in many domains, such as geo-statistic, surrogate modeling, avionics and microchip design, where multidimensional and multivariate models are common. This enables Kriging models, and the controllers employing them, to leverage concepts, results and techniques from the statistical domain. Thanks to this, Kriging models can provide a measure of uncertainty that is associated with predictions and that can be used to define a confidence interval around them. The confidence is computed in a well defined way, and depends on the amount and the quality of the data around the prediction point and the complexity of the unknown function. We can interpret this measure in different ways: Either there are not enough samples to build a reliable model nearby the prediction point, or the system behavior close to the prediction point is more dynamic than the remaining parts of the model. Additionally, Kriging models do not *degenerate*, that is, they pair predictions that are distant from the training data with the maximum level of uncertainty. Thanks to these properties we can design robust and consistent control strategies that account for the uncertainty in the data and do not show unexpected, counter-intuitive reactions when applied in (yet) unknown working conditions.

Kriging models are *non-parametric*, that is, they do not require a predefined structure to be specified beforehand.[1] The *internal* structure of Kriging models is inferred directly from the data provided during the training and later either adjusted or recreated during the model updates. Therefore, Kriging models are easier to setup than traditional regressors or artificial neural networks that at the contrary require the def-

[1]Note that non-parametric models are not *parameter-free* models, for example, Kriging models have hyper-parameters.

inition of a predefined model structure. Moreover, the monitored data used to build them is easily recoverable from the running systems, either through active monitoring or analysis of application logs, and this make Kriging models even more easy to build.

4.3 Kriging models for performance prediction

We use Kriging models to approximate the *steady-state* relations between the indexes of the performance of the system, which are the dependent variables, as function of both the system configuration, input workload, and other system metrics, which are the independent variables.

The choice of adopting a model that describes the steady behavior of the system is in line with related work both on Cloud computing and traditional performance engineering. However, it differs from the classic control theory that prefers models of system dynamics rather than of the steady state. We motivate our choice with two reasons: First, by employing steady state models the quality of the control in many cases is sufficient for the problem at hand. Second, considering the complexity of elastic applications in the Cloud, it is difficult to obtain an accurate, yet manageable, model that captures the systems dynamics right as *first principles models* would do. In this sense, steady state models are simpler and, at this stage, pay off better than complex models of the system dynamics.

A Kriging model can predict a single dependent variable that depends on many independent variables. In this work, we are interested in capturing performance indexes, such as average system response time and throughput, therefore we define a different Kriging model for each of them to predict their value simultaneously. In doing so, we consider performance indexes as not correlated and we use the same independent variables for each Kriging model. Models that can predict multiple outputs, like the one proposed by Ganapathi et al. [32], can be more accurate than ours, but are also more complex to manage, therefore their applicability to our domain is still a subject of discussion.

We translate controlled variables, non-controlled variables, and disturbance into several input features of the models: To account for the system configuration, we define an input feature, denoted as $\#VM_x$, for each type of VM. To account for the workload, we define an input feature, denoted as RC_y, for each type of new user request received. Similarly, to account for the requests currently inside the system, we define an additional input feature, denoted as QL_y, for each type of user request.

This situation is summarized by the following set of expressions that are related to a generic system with n types of VMs, and m types of user requests, where we want to predict the average response time and throughput for the different types of requests:

$$
\begin{aligned}
avgRT_{i=1..m} &= f_i(\overbrace{\#VM_1,\ldots,\#VM_n}^{\text{resources allocation}}, \overbrace{RC_1,\ldots,RC_m}^{\text{environm cond}}, \overbrace{QL_1,\ldots,QL_m}^{\text{other metrics}}) \quad (4.1)\\
avgTx_{i=1..m} &= g_i(\underbrace{\#VM_1,\ldots,\#VM_n}_{\text{system conf}}, \underbrace{RC_1,\ldots,RC_m}_{\text{input workloads}}, \underbrace{QL_1,\ldots,QL_m}_{\text{queued requests}}) \quad (4.2)
\end{aligned}
$$

where f_i and g_i are Kriging models.

Each application and SLO metric has a different model consisting of a different set of features. For example, Figure 4.1 shows a Kriging model of the average response ($AvgRT$) time of a two-tier Web service that can scale its application server tier ($\#AS$), therefore:

$$AvgRT = f(\#AS, RC)$$

This model captures how $AvgRT$ changes as the workload RC (measured in `req/sec`) and $\#AS$ change. Figure 4.1 shows the model as a three dimensional surface on the left, and in a top view on the right side. In the figure, different colors represent the different values of the output features.

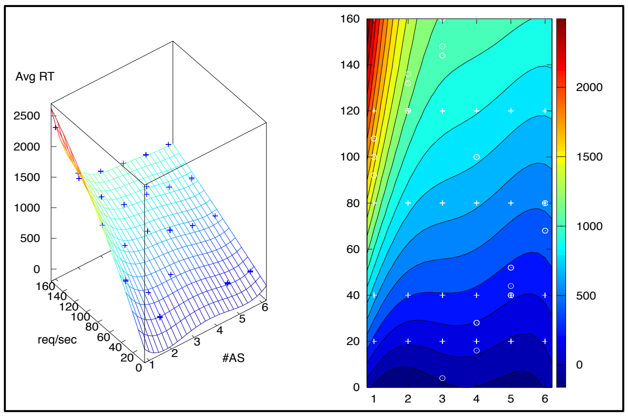

Figure 4.1. A Kriging model for predicting performance [31]

Figure 4.2 shows a Kriging model for another virtualized system (see [103] for more details). The plots in this figure refer to the average response time of one type of requests (`AvgRT`) and the corresponding variance in log-scale (`sig`). The average response time is computed with respect to the requests that reach the system in the same monitoring interval. This means that in each monitoring interval, given the amount of requests per type received, and the current state of the system captured by means of the queue length, the models predict the average response time of these particular requests. `AvgRT` is modeled as a function of the three types of VMs that comprise the system, and the number of concurrent clients invoking it. The number of concurrent clients is an approximation of the request rate because the generation logic of requests remains fixed during each run, i.e., the amount of requests generated is proportional to the number of concurrent clients, and the request mix is constant.[2] Therefore, the model has four input dimensions. Figure 4.2 shows a projection of the original space along the three dimensions of the system configuration.

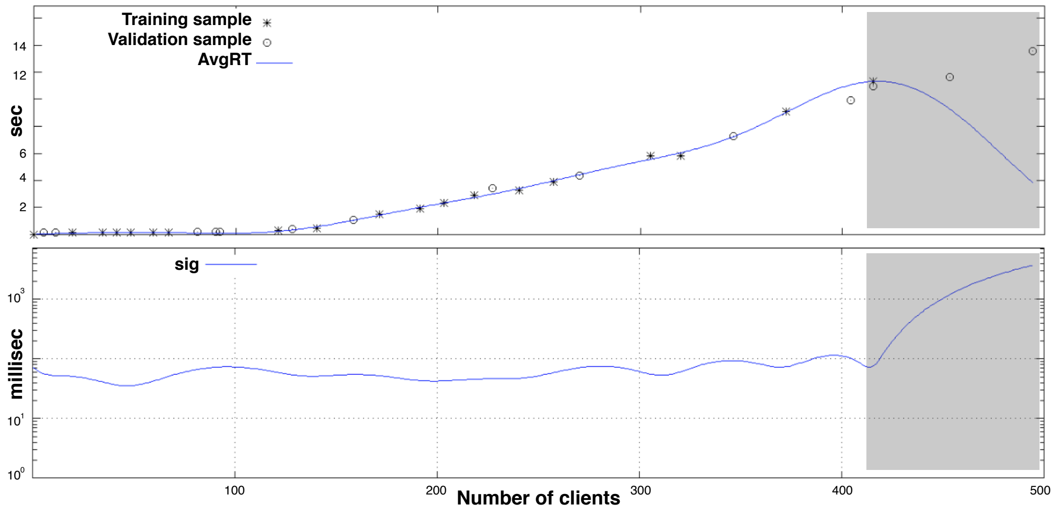

Figure 4.2. A Kriging model showing the average response time of a system (upper plot) and the confidence of its prediction (lower plot) as function of the concurrent number of clients. The plot shows the model of the system behavior for just one of the possible system configurations.

Plots in Figure 4.2 give an intuition of how the output variable and its confidence interval change over the input features space, according to both the value and distribution of training samples. The left side of the plot is obtained from multiple training samples and shows the system under “light” load, the middle of the plot is obtained with fewer samples and shows the system under sustained load, while the right side of the plot is obtained with sparse samples and shows the system under heavy load. In the first case, the predicted response time is in the order of millisecond and its variance is relatively low; in the second case, the predicted response time increases but the variance is still low; and in the last case, which corresponds to the shaded area, the predicted response suddenly drops as the load increases. This last case is counterintuitive because the response time of the system should increase, therefore the model in this region is inaccurate. This situation can be explained considering the training data that include just one sample in this region, and the *interpolating* nature of Kriging models that adapt to the model trend away from the training data[3]. In the shaded

[2]The behavior of the final users is modeled by means of a Markov chain that includes the invocation of all the possible operations.

[3]Kriging models behave differently from other models, such as regression trees, that implement data *extrapolation* instead.

region, the inaccuracy of the model would lead to ineffective control if not properly considered.

Fortunately, Kriging models have a clear mean to identify these situations and account for the uncertainty of its predictions. In fact, by inspecting the lower plot, we can clearly see how the value of `sig` increases of almost two orders of magnitude. By taking this into account, one expects that predictions from the model may be inaccurate, can take corrective actions and implement a more robust form of control.

For modeling system performance using Kriging models, we have to assume that the correlation among the input features is *isomorphic*, that is, the correlation depends only on the distance between the observations and not on their absolute value or location. Despite the fact that did not formally verified this assumption, the results of the empirical validation suggest that, at least in the investigated case studies, it holds.

4.4 Kriging models for robust control

When the controller identifies a risky situation that may lead to SLA violations, or an over allocation of resources, it computes a set of control actions given the currently monitored variables leveraging Kriging models. The planning process is performed in two step: 1) Find a suitable system configuration, i.e., *target* configuration; and, 2) derive the sequence of control actions that lead the system from the current configuration to the target one.

During the planning, the controller reasons in terms of system configurations. This reduces the space of possibilities compared to cases like the one based on reinforcement learning in which controllers consider basic control actions and their combinations to modify the controlled system. This also avoids complications due to contrasting effects of different control actions, such as adding and removing similar VMs. Moreover, considering system configurations as first-class citizens allows to deal with all the system bottlenecks simultaneously without going through the process of stepping between shifting bottlenecks one at a time [69].

The planning is governed by control policies that aim to find system configurations that are *valid*, i.e., respect any architectural or business constraints posed on the system, and that avoid or recover from SLA violations. When the controller finds more than one configuration, it chooses one according to a configurable second level policy. Control policies differs in how they search the configuration space to find valid configurations, how they choose the target configuration, and when they apply it.

In this work, we define a “lazy” control policy that acts on the system only whenever a violation is about to happen, that is, it is predicted to happen. The intuition behind this choice, even if not formally stated, is that each control intervention has a cost, and the lazy policy aims to minimize this cost as much as possible. To do so, the controller leverages early warnings at every control cycle and acts whenever the model predicts a violation of SLAs given the actual monitored variables. If no warnings are raised, the controller inspects the model to identify unnecessary VMs and eventually remove them.

Assuming a risky situation is identified, to derive the target configuration, the controller enumerates all the possible system configurations and filters them if there are structural or business constraints.[4] An example of constraint is a limit on the number of VMs. In this case, one could have a maximum number of allocated VMs below a threshold to limit the costs of running the system. Enumerating all the feasible configurations is possible because the space of system configuration includes few dimensions and their combination results in a tractable space. Then, the controller removes from the set of possible configurations the ones that the model predicts will lead to SLA violations. To implement this check the controller queries the Kriging models using as input the currently monitored values and the values corresponding to the chosen configuration and compares the prediction with the QoS constraints. The controller orders the remaining configurations by their costs. And it chooses the system configuration that has minimum costs. In case of equally expensive configurations, we use as second-level policy to pick the first configuration found.

Alternative control policies can be defined by implementing the planning steps differently from what we explained. Here we give only a few examples and we left the investigation of the best possible combinations of planning activities to future work.

Instead of relying on monitoring data at the beginning of a control period, one can use predicted values, e.g. of the workload, and then inspects the possible evolutions of the system similarly to what we did. Taking into account historical time series and periodic behavior has the advantages of being more proactive by enabling long-term predictions. However, using predictions instead of monitored values could be error prone. In fact, if these predictions are not very accurate their combination with the models may lead to wrong results. Moreover, in general, the longer is the prediction horizon the greater may be this inaccuracy. And this limits the extent of controller proactiveness.

Instead of enumerating all the possible system configurations and process them one at a time to filter the result set, one can devise smarter explorations of the system configuration space or reduce the amount of configuration considered according to some heuristic. For example, one can follow a gradient based search, use Genetic Algorithm, or other techniques. However, in our scenarios, the amount of potential system configurations is limited and complex techniques for exploration may not improve the speed of the controller.

Instead of focusing only on costs to rank valid configurations and then promote the first configuration found as target configuration as we did, one can devise other strategies both to rank potential system configurations and to chose among equivalent configurations. For example, one may prefer to have an equal allocation of VMs along the different components running the system, while another may prefer to have a

[4]Even if theoretically possible, we do not assume any filter on system configurations in our case studies.

configuration that is able to scale-up better along one dimension, i.e., VM type, more than the others.

As another heuristic, a controller may privilege configurations that have already been adopted in the past over configurations that are totally unknown, even if the latter may be more promising from a QoS-wise point of view. The ability of Kriging to provide confidence intervals on predictions is a requisite towards the implementation of more robust control policies that tolerate and account for uncertainties in the control loop.

We make our implementation of the lazy control policy robust by leveraging this feature of Kriging models every time the controller inspects them: Whenever controllers query the models, they combine the value of the prediction and its expected variance in a ratio, and check if the resulting value is below an accuracy threshold. If it is, then the controller considers the prediction reliable and proceeds as explained before, otherwise the controller reject the prediction. Intuitively, the ratio captures how much the controller trusts the Kriging models: the smaller the threshold value, the more conservative is the controller, i.e., the less trust in average is put on the Kriging models. In our implementation of the robust control policy, the controller resorts to a second model for obtaining the prediction on the system behavior every time this accuracy check fails. In particular, we implement the second model as an analytical model that assumes a linear correlation between the response time and the length of the queue. And we empirically found that a value of the accuracy threshold equals to 0.4 gives already enough accurate results. This strategy works because the prediction error and the confidence measure are correlated, meaning that when the confidence is low the model actually tends to make wrong predictions. More details can be found in Appendix A.

The space of alternative implementations is open and one can implement different strategies both to compare the predicted values and variance, and to react whenever the reliability of the prediction is not satisfactory. For example, when designers cannot resort to the second model, controllers can look for system configurations that have better accuracy even if they are not optimal in the costs/QoS dimensions, or choose configurations that are very capable, thus reducing the risk of under-provisioning the system.

One can define exploratory control policies that privilege control actions taking the system to those regions with higher uncertainty and collect new monitoring data. Finally, one can also implement strategies that use both the predicted values without checking the reliability of predictions: For example, one can follow a conservative strategy and consider only the pessimistic values of the predictions by penalizing the predicted value proportionally to its uncertainty, and then use the penalized prediction to drive the control policy.

As just stated, in this work we adopt a solution based on multiple models that we use according to the following strategy: If the value of the variance (denoted as `sig`) that Kriging models give along the prediction is below a configurable threshold, then we use the value predicted by the models, otherwise we resort to the second model.

We work under the assumption that the analytical model has a lower average accuracy than the Kriging model but that it performs better where Kriging have no training data yet. This solution is conceptually similar to the one proposed by Malkowski and coauthors [70]. Both solutions have the potential advantage to provide a minimum level of prediction accuracy by choosing the model with the expected lower prediction error.

Moreover, it is interesting to note that such combination leads to an overall improvement of the controller. In fact, the analytical model may lead the system in operating regions that are not yet discovered. This in turn may result in the collection of new monitoring data and an increase of the accuracy of Kriging models. With more accurate models, the controller resorts less frequently to the analytical model and improves the overall effectiveness of the control policy.[5]

Once the controller finds the target configuration it must decide which control actions must be implemented. To derive these control actions, the controller compares the current and target configurations, and for each type of VM, it computes how many VMs it must add or remove. At the end, all the control actions are then applied in one solution.

4.5 Self-adaptiveness by model adaptation

Self-adaptive controllers change their control strategy at runtime, possibly without human intervention. Model-based controllers plan control strategies according to the models that they use to describe the systems. Modifications to these models lead to changes in their planning. Therefore, we can adapt self-adaptive controllers by modifying the underlying models.

We follow this approach to self-adaptation and adapt the Kriging models by retraining them on-line as new monitoring data are collected. The controller uses a configurable sampling selection policy to decide which monitoring data are retained, and how to pre-process them before updating the model. For example, we adopt the simple approach of averaging the last five samples for a given combination of inputs at every control cycle and use the resulting value as training sample. In this way, the model is retrained with the latest collected data and we do not accumulate a large amount of samples.

Other strategies can be implemented as well, for example, one may use the variance and the count of samples for input combinations to match the (local) complexity of the system behavior with the density of data around that point. This is similar to the approach proposed by Ankenman, Nelson and Staum [4]. Alternatively, one can track

[5]The analytical model can be updated too with new monitoring data, for example using Kalman filters to track system service times as described by Zheng et al. [116]

the prediction error and retrain the model only if its value is above a given threshold. This approach is commonly found in related work and it has the advantage to avoid frequent retraining of the model, if the threshold is set to the right value.

Building Kriging models implies two step: Dataset update and hyper-parameter optimization. When we retrain the models, we always perform both steps. However, when the system behavior is stable, the second step may be avoided during model adaptation assuming that the current hyper-parameters are good enough. This is for example the line followed by Sheikh et al. [96]. The strategy to update only the training dataset requires less effort and is faster than the strategy we propose, however it makes also stronger assumptions about the quality of the initial hyper-parameter optimization and about the stability of system behavior. These assumptions may be violated in our application domain, i.e., Cloud based applications, thus it may lead to inaccurate models and non-effective controllers. Moreover, accumulating new samples without removing old ones may fail to capture evolutions of behavior that are hidden by the data describing past behaviors, and reduce the speed of the controller.

In fact, the time of retraining Kriging models is cubic in the size of training set. Thus keeping the size of training data set small is important. Fortunately, two facts reduce the importance of this problem: First, the actual amount of data needed to build *accurate enough* models for our application domain is not prohibitive; second, one can adopt a range of optimizations that reduce the time to build the model. For example, one may pre-process the training data by performing data-binning before building the model.

Figure 4.3 gives an intuition on how Kriging models evolve during system runtime. The figure shows three consecutive snapshots of a model built in one of our case studies: The model at the left is trained with six samples, the one in the middle with twelve samples, and the one at the right with thirty samples.

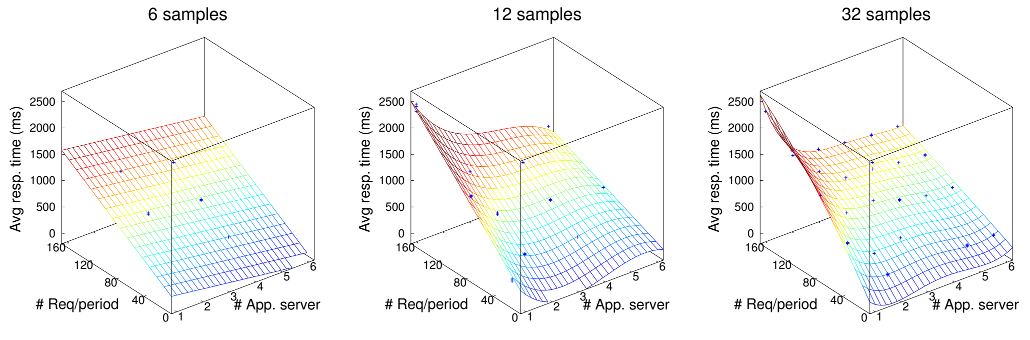

Figure 4.3. Runtime evolution of Kriging models

Intuitively, one can see that the controller will act differently in these cases by checking that for the same workload and the system configuration, the output of the model changes. For example, assume that the maximum average response time, as specified in the SLA, is two seconds. For the maximum input workload in the figure (160 req/per), the model on the left predicts an average response time of 1600 milliseconds and the controller will not change the current system configuration. For the same inputs, the model on the right predicts 2500 milliseconds, and the controller will trigger the allocation of additional VMs. Therefore, by dynamically adapting the model the controller does indeed update its strategy.

4.6 Suitability of Kriging models

During design time, we collect an initial set of training data, and we train Kriging models with samples obtained by averaging the value of monitoring data over long intervals to approximate the system steady-state behavior. At runtime, the controller collects the "instantaneous" data, uses the sample selection policy to filter and combine them, and eventually, retrains the models.

It must be noted that the controller uses the models at runtime to predict the system performance given as inputs the instantaneous values of the KPIs and not their average. Therefore, predictions of Kriging models will be accurate as long we can assume that the system is in a steady state that mean that (i) the system is under light or moderate load, i.e., its utilization is below the saturation point; (ii) the workload changes slowly enough to let the controller react and avoid, even temporary, system saturation; and, (iii) that the effects of control actions are immediately evident on the system behavior.

When all these conditions are met, steady state models, such as the Kriging models for performance prediction presented in Section 4.3, can be safely used both from early warnings and planning with simple control policies, like the one presented in Section 4.4. Unfortunately, in the context of Cloud based applications these assumptions are not always valid. In fact, the incoming workload may change quickly, the deployment and removal of VMs may take up to several minutes, and in general the current state of the system may impact on its behavior.

When these assumptions are not met, there are several consequences that depend on the actual working conditions: For example, the effects of the load and the speed of variation of the incoming workloads are inversely proportional to the scale of the system, so in less capable configurations the effects are noticeable, while in very capable configurations are negligible. Similarly, the faster the execution of control actions the more the predictions on system behavior are accurate, with increased accuracy for big configurations.

Given these considerations, some authors propose "palliative" solutions that consist of avoiding very small system configurations or using *hot replicas* ready to be plugged in the system. In both cases, the operational costs of the system increase as a consequence of running more machines than needed.

To soften the set of working assumptions of our controllers while avoiding these palliative solutions, we improve our initial strategy as described in the next section.

4.7 Model predictive control with Kriging

We improve the initial solution based on Kriging models by explicitly considering the system state and the delayed effects of control actions, and by simulating the expected evolution of the system in the next future, i.e., after the applications of control actions, in accordance to model predictive control.

In this work we focus on performance, therefore we represent the system state as the amount of requests queued in the system and waiting to be processed, as this measure concisely summarizes the past history of the system. We add a new input feature to the Kriging model for each type of user request whose value corresponds to the length of the corresponding waiting queue. It must be noted that these queues are an abstraction over the system implementation that may have no separated queues. By explicitly considering the system state, the accuracy of the models increases, especially when the system has heavy load and the length of waiting queues impacts more on performances. Moreover, Kriging models scale well with the number of dimensions, and additional input features do not increase the effort for managing the models with respect to the initial configuration.

The final performance of the controller depends not only on the accuracy of the models, but also on their ability to capture (or simulate) the evolution of the system state. This is important when the controller has to account for any transitory effect, for example between configuration switching, and compensate for the inaccuracy of invalid modeling assumptions.

The controller needs to known the amount of time for a control action to impact the system behavior to properly simulate the system evolution in the future. For example, before implementing a scale up action, the controller needs to understand what will be the system state at the time the new VMs will start to elaborate queued requests; on the other hand, before scaling down, the controller needs to understand if the previously queued requests will be elaborated, thus if it is “safe” to remove VMs. We empirically measured the average time needed to change system configuration, i.e., to start one or more VMs and wait for them to start processing requests and to remove one or many VMs, and we input this value in the control policy. In practice, different configurations may be reached after different times, and the controller should consider this difference while planning; however, in this work, we assume a constant time to move between any system configuration leaving further investigations for future work.

The control policy assumes a known duration of control actions and the stability of the workload inside each control cycle, and predicts the state that the system would reach by considering the control actions already triggered and the control actions it can potentially start. The model predictive control policy works in two steps: First, it simulates the changes to the system state, i.e., how the waiting queues evolve, due to the already triggered control actions and assuming a stable workload; second, from this predicted state, it proceeds as we described in Section 4.4 with the simulation the effects of potential changes of configurations. We compute incrementally the state that the system would reach due to the running control actions over the delay we expect to implement any control action under planning. We start form the monitored state at the beginning of the control interval, and we compute the future instants when control actions will take effects, i.e., when the system will actually change its configuration. As many actions may be scheduled for execution, this results in a set of intervals between consecutive configurations. Then, we use Kriging models to understand how many requests would be processed between any of these configurations and combine these values with the amount of incoming requests that we will expect over the same intervals, for example, the average value of incoming requests over the last ten control intervals.

To simulate how a queue evolves, we input the current configuration into the Kriging models and we fix the incoming workload to a single request. In this situation, by simulating different queue lengths, the model predicts the average time to complete that single request in each of the cases. Given a fixed time horizon, for example the time to reach the next system configuration, we can approximate the number of requests that can be processed by searching the maximum value of queue length such that the predicted response of incoming requests is smaller than the considered horizon. Once we obtained this approximation, we can compute how the value of the queue length evolves due to that system configuration. This procedure is repeated iteratively for all the system configurations that were scheduled before the current control period, and at each iteration the variation of the queue length is accumulated.

At the end of this procedure, we know if the queues increase, decrease or stay stable, and also by how much they are supposed to change, between now and the time any of the new control actions will take effect. By taking into account the system state and its evolution we can plan more suitable control actions that compensate for the transitory effects on the system behavior due to delays of control actions and unmet working assumptions.

In case of scaling up, this procedure results in the smallest configuration that is able to sustain the incoming workload and the load due to the already queued requests. In case of scaling down, the procedure needs some minor adjustments to avoid *resource thrashing*, i.e., removing VMs right after they start and starting new VMs right after they stop. We want to avoid scaling down actions if a scaling up is taking place. This is needed because scaling up actions taken in the past account for the effect of “future” incoming requests that may have been supposed to queue up, and it is a consequence of letting the controller apply control actions much faster than the physical system reaction time. We adopt such more complex strategy in order not to hamper the proactiveness of controllers, in fact, the alternative solution would be to pause the controller in between the applications of control actions, that is in the order of minutes in our application domain.

If we were to remove VMs that were just started, it may happen that the requests queued at the time those VMs were started would still be in the system without the necessary resources to run. In summary, by removing such VMs one will invalidate the predictions made in the past control intervals where the corresponding control actions were issued, reducing the effectiveness of control. Moreover, if the scaling down is too aggressive it may happen that the queues fill up too quickly before any other control action will complete due to the approximations done to account for the incoming workload.

Consequently, whenever a planned configuration tries to scale down the system, before implementing such modification, we run two checks: First we compute the smallest configuration that would tackle the expected incoming load without considering any state evolution, we compare it with the one obtained via the control policy just presented, and we take the most capable one. Then we check if the chosen configuration is able to consume the actually monitored workload. If this is the case, then we found the target configuration, otherwise we choose a slightly bigger configuration and we repeat the process.

4.8 Controller architecture and prototype implementation

To implement the Kriging-based self-adaptive controllers we instantiate the generic architecture of model-based self-adaptive controllers introduced in Chapter 2 and we refine it by adding the software components for building Kriging models and automatically adapt them at runtime. Figure 4.4 shows the conceptual architecture of our prototype.

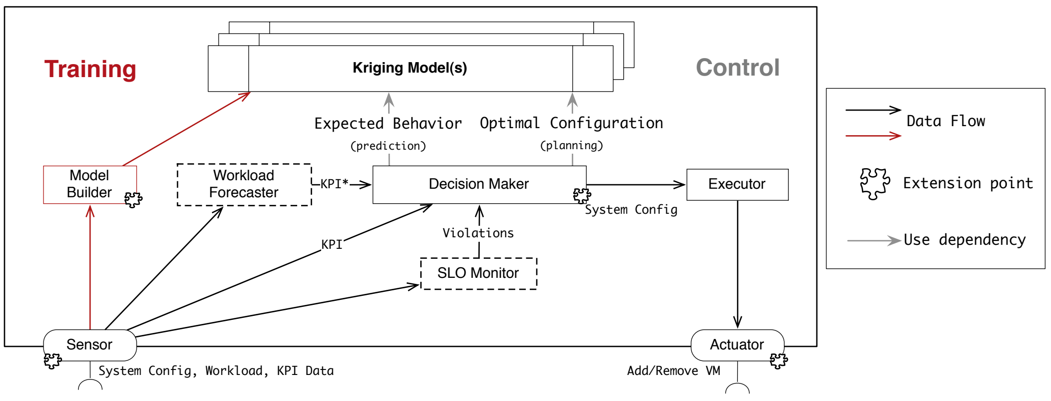

Figure 4.4. Conceptual architecture of Kriging-based self-adaptive controllers

The controller executes two concurrent loops: the *control* loop and the *training* loop. The control loop involves several components: It starts with *sensors* that collect monitoring data from the application; these data are forwarded to the *decision maker* that implements both early warnings and configuration planning. It interacts with the *Kriging models* to perform both of them. Once the target configuration is ready, the decision maker invokes the *executor* that computes the basic control actions to apply. The executor then sends the sequence of control actions to the *actuator* that in turn forwards them to the Cloud infrastructure. The training loop implements model self-adaptiveness and involves the sensors to gather monitoring data, the *model builder* that elaborates them to prepare the training set, and that trains the Kriging models.

The *workload predictor* and the *SLO monitor* are optional components that can be executed to improve the effectiveness of the control loop by providing mid- and long-term predictions about incoming workloads and to feed back into the controller the amount of SLA already violated.

We implement this conceptual architecture in a prototype written in Java. The implementation follows a *plug-in* architectural style such that controller's components can be easily replaced. At the moment, we have implemented plug-ins that use the Octave engine[6] for building the Kriging models, plug-ins for sensors and actuators to bind to several Cloud infrastructures, such as Reservoir [7], Eucalyptus [8] and Amazon EC2 [9], and plug-ins that implements different control policies, such as our lazy control policy, our model predictive control policy, as well as simple rule-based control policies. We also implemented a flexible monitoring framework, based on JMS Pub/Sub and its implementation provided by Apache ActiveMQ [10], for easily collecting and distributing monitoring data.

We use this prototype to carry out the empirical validation of Kriging-based self-adaptive controllers, as we describe in the following chapter.

[6]http://www.gnu.org/s/octave/
[7]http://www.reservoir-fp7.eu/
[8]http://www.eucalyptus.com/
[9]http://aws.amazon.com/
[10]http://activemq.apache.org/

Chapter 5

Evaluation

In this chapter, we evaluate the contributions made by this thesis in three steps: We begin by evaluating the predictive abilities of Kriging models in the context of elastic applications; we proceed by evaluating the suitability of Kriging models as core elements of autonomic controllers; finally, we conclude by evaluating the performances of Kriging-based self-adaptive controllers for the Cloud.

The evaluation process is based on a set of experiments that we designed to answer the research questions introduced in Chapter 1. For each research question, we describe the experiments that we designed, and we discuss the results that we obtained. We report additional details about experiments in Appendix B.

5.1 Experimental settings

In this section, we briefly describe the testbed infrastructure and the case studies that we used to conduct the evaluation process.

We chose the Reservoir Cloud as middleware for IaaS. This choice is motivated by our need of experimenting with real systems, the availability of a private physical infrastructure to our disposal, and the past experience that we gained by working in the Reservoir EU project[1], a leading research initiative on infrastructure virtualization and Cloud computing that involved several industrial partners.

The Reservoir Cloud has a layered software architecture that accounts for: A central point of control, called Virtual Environment Execution Manager (VEE Manager), manages a set of hardware resources, called Virtual Environment Execution Hosts (VEE Hosts), that in turns, run the virtual machines, called Virtual Execution Environments (VEEs). VEEs that belongs to the same Cloud user collaborate to implement complex applications or services. A component called Service Manager controls the lifecycle of services, and exposes to the outside world the APIs for monitoring and control the

[1]http://www.reservoir-fp7.eu/

virtualized infrastructure. Our controllers connects to these interfaces to receive monitoring data, and to implement their control strategies.

A Service manager, a VEE Manager and a set of co-located VEE Hosts comprise a Reservoir site, as Figure 5.1 shows. In principle, Reservoir sites can join to form federated Clouds, but in the rest of the evaluation we use a single site configuration.

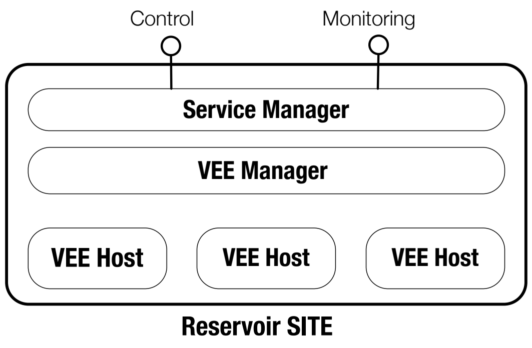

Figure 5.1. Logical architecture of a Reservoir site

We deploy the service manager and the VEE Manager on a 2-vCPU Debian *Lenny* virtual machine that we run on a dual-core CPUs Dell desktop. We deploy a single VEE Host in a 24-core physical server. The available resources on which VMs can be allocated limits the size of the system configuration space, therefore, in our experiments we can run virtualized systems that use at most twenty-two cores. We leave the remaining two cores of the server free for background OS tasks.

Case studies

During the evaluation, we use different systems as case study. We choose these systems because they are representative of the systems that typically run in Cloud infrastructures.

In particular, these systems are: the Sun grid engine,SGE in brief, a batch job processing system typically used for scientific computations; the Doodle Web service and the Rice University Bidding Systems (RUBiS), traditional n-tiers Web applications; and, the DoReMap system a composition of RESTful Web services.

Each of them has different requirements and SLOs, workloads and scalability needs that we briefly describe in the remaining of this section.

Sun grid engine The Sun grid engine is a job batch processing system with a master-slave architecture. Clients submit jobs to the master node that in turns dispatches them to executor nodes. Each executor node can executes one job at time and notifies the master node at completion. Executor nodes can be dynamically added and removed, thus they form an elastic execution layer. We configure the master node to have a single queue of jobs, and we run it on a 2-vCPU virtual machine. Executors instead run in virtual machines with a single vCPU. The SLA for the SGE contains a single service level objective that defines a threshold of two minutes on the average jobs completion time. We measure the SLO over a five-second sliding window. The average job completion time is between five and six seconds. Figure 5.2 reports the logical architecture of the Sun grid engine.

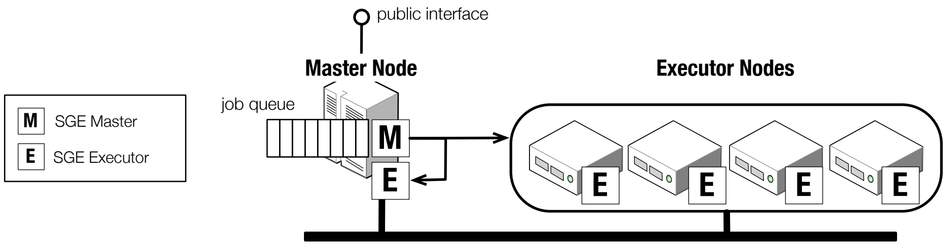

Figure 5.2. Logical architecture of the Sun grid engine

Web application We consider the Doodle Web service [2] as an example of common n-tier web application. The Doodle Web service is a 2-tier implementation of a voting polling service. A load balancer forwards incoming requests to the inner tiers of the system that process them. Eventually, servers issue requests to a database tier. Application servers form the elastic tiers of these applications. We configure the Doodle Web service to use a 2-vCPU virtual machine to run a central database and 1-vCPU virtual machines to run application servers. The SLAs of the application defines a maximum threshold of one second on the average response time for each type of request. Figure 5.3 reports the logical architecture of the Doodle Web service.

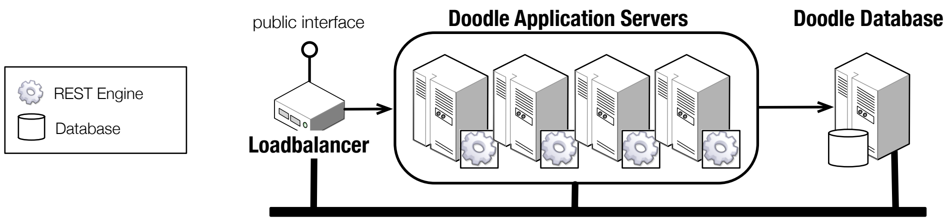

Figure 5.3. Logical architecture of the Doodle Web service

DoReMap DoReMap is a RESTful Web service composition originally presented by Pautasso [85]. It contains a loadbalancer component that distributes the incoming requests to a battery of JOpera composition engines [86]. Engines compose two atomic

[2]This is a clone of a real Web service available at `http://www.doodle.com/`

RESTful Web services: The Doodle Web service and a search engine, called RestaurantSearch, that is inspired to Yahoo!Local[3]. The final service composition offers context aware voting polls. We use a 4-vCPU virtual machine to run the loadbalancer and other infrastructural components for monitoring, a 2-vCPU virtual machine to run the database of the Doodle Web service and JOpera engines, and 1-vCPU virtual machines to run the other components. All components but the loadbalancer and the database of the Doodle Web service can be scaled dynamically. The SLA of DoReMap defines thresholds on the maximum average response time for the different operations at the composition level. Figure 5.4 reports the logical architecture of the DoReMap system.

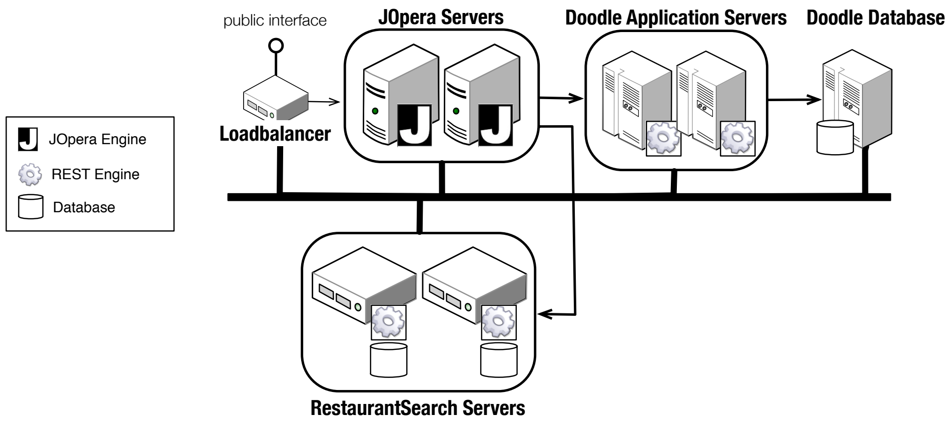

Figure 5.4. Logical architecture of DoReMap

In the remaining of the chapter, we present extensive results for the Sun grid engine case study, while we report results for the other case studies in Appendix B

5.2 Kriging models for prediction

To our knowledge, this is the first time that Kriging models are used to describe software systems running in the Cloud. Therefore, we need to evaluate the ability of Kriging models to correlate monitoring data and accurately capture the behavior of such systems, hence, addressing our first research question:

> **RQ-1**: Are Kriging models able to capture the behavior of elastic applications run by Cloud infrastructure accurately enough?

We adopt the following procedure to conduct the evaluation: We sample the system behavior through a batch of controlled experiments; we partition these samples into one training and one validation sets; we build a Kriging model using the data from the

[3]http://local.yahoo.com/

training set and predict the validation set with the model; we compute the prediction error by comparing the real and predicted values.

To stress the system, we generate synthetic workloads drawn from Poisson distributions with constant mean and fixed request mix. We discard monitoring data at the beginning and at the end of each run to avoid the undesired effects of components start-up and shutdown and we obtain the final samples by averaging the output of the run.

Accuracy of Kriging models for the Cloud

We evaluate the overall accuracy of Kriging models by computing widely used error metrics: the mean absolute error (MAE), the root mean square error (RMSE) and the mean absolute percentage error (MAPE). Having a validation set $\mathbb{V}$ of n samples, $\mathbb{V} = \{(\mathbf{x}_i, y_i) | i = 1, \ldots, n\}$, where $\mathbf{x}$ denotes an input vector and y denotes the corresponding scalar output (i.e., the dependent variable), and defined $\hat{y}_i$ the model prediction corresponding to input $\mathbf{x}_i$, the error metrics are computed according to the following formulas:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{5.1}$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \tag{5.2}$$

$$\text{MAPE} = \frac{100}{n}\sum_{i=1}^{n} \left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{5.3}$$

We present here the results for the Sun grid engine case study, and we report in Appendix B results for the other case studies.

We run the batch of experiments and we collect 636 training samples and 91 validation samples, therefore $n = 91$. We build a Kriging model for the average response time as function of the number of executor nodes, incoming requests, and queued jobs using the 636 samples. For reference, we plot this Kriging model in Figure 5.5 for three (out of twenty-two) system configurations, with 5, 10, and 15 running executor nodes. The plots show the average response system time on the z-axis, the number of queued jobs on the x-axis, and the number of incoming requests on the y-axis. We predict the 91 validation samples with the Kriging model to obtain a MAE of 5742.56 millisecs, a RMSE of 9395.30 millisecs and a MAPE of 15.42%.

We consider Kriging models accurate enough if the MAPE error is below 20%. To put the accuracy metrics in the context of the case study we compare error values against the SLO on average response time, which is equal to 120 seconds. In SGE case study, the value of MAPE is below the accuracy threshold, and both the values of

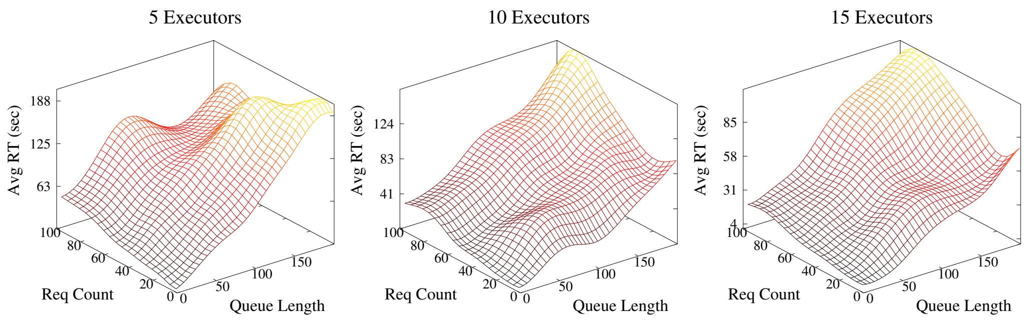

Figure 5.5. Different projections of the Kriging model for the Sun grid engine case study

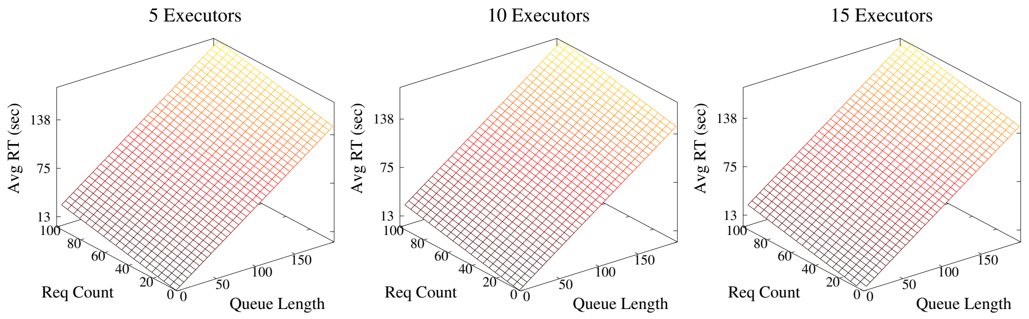

Figure 5.6. Different projections of the Multidimensional linear regression model for the Sun grid engine case study

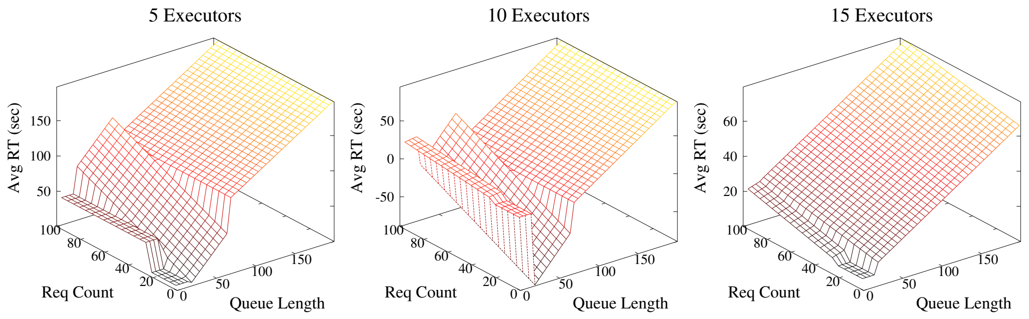

Figure 5.7. Different projections of the Regression tree model for the Sun grid engine case study

MAE and RMSE are relatively small compared to the threshold on the average response time of 120 seconds, meaning that the impact of the prediction error on the quality of the control should be tolerable. We must note that these samples are obtained from running systems and are inherently noisy, and this justifies the fact that prediction errors cannot be eliminated completely.

Limited to the SGE, these results are promising and show that Kriging models can capture the behavior of the considered application reasonably well. As reported in the appendix, we obtain similar results also for the other case studies that we considered. In light of this, we can use Kriging models to capture the steady state behavior of systems deployed over Cloud infrastructure.

Comparison with other models

To have a more comprehensive view about the predictive abilities of Kriging models, and also to support our claims on the advantages derived from using them with respect to other solutions, we compare Kriging models with other models.

We select the alternative models among the ones that can be found in literature, and for which we could find an available implementation. In this way, we can reduce the bias of the results as well the effort of conducting the evaluation. We consider the following *black-box* models for the comparison: multidimensional regression models, M5 regression trees, and multivariate adaptive regression splines.

In the comparison, we consider also an analytical model, as instance of white-box models, and a combined model that we obtained by mixing together the Kriging model and the analytical model as we explained in the previous chapter.

Multidimensional linear regression Multidimensional linear regression models take the form of $y = X * b + c$ and are trained via least square error minimization. For the evaluation, we use the implementation provided by the statistics package[4] of Octave. Figure 5.6 shows the plots of multidimensional linear regression models for SGE configured with 5, 10, and 15 executors. As before, we plot on the z-axis the predicted average system response time, on the x-axis the number of queued jobs, and on the y-axis the number of incoming requests.

M5 Regression trees M5 Regression trees in the continuous case correspond to the widely known classification trees in the discrete case. They divide the multidimensional input space into a set of planes that correspond to the leaves of the tree. Regression trees are trained in a two step process that discovers an initial node and recursively grows to maximize each branch via standard deviation reduction, and then minimizes the tree by pruning each leaf that minimizes the estimate of the expected error. For

[4]`http://octave.sourceforge.net/statistics/index.html`

the evaluation, we use the implementation of M5 regression trees provided by Jekabson [29]. We plot in Figure 5.7 the model for the selected configurations of the SGE application.

Multivariate adaptive regression splines Multivariate adaptive regression splines are a combination of basis functions that are placed over the space of the input features in special points called *knots*. They are trained via a two-phase procedure: The forward selection iteratively adds basis functions that give the largest reduction of the training error, and the backward deletion simplifies the model by deleting the least important basis function at each time. In the evaluation, we use the implementation of multivariate adaptive regression spline provided by Jekabson [30]. Figure 5.8 shows the model for the selected configurations of the SGE application.

Analytical model The analytical model simulates the behavior of an ideal multi-server single queue. We implement the model and manually set the *service time* parameter as the empirical mean value of the job execution time. It must be noted that we were able to implement this model because the Sun grid engine has a software architecture that can be easily translate into a multi-server single queue model, and because we are considering only one type of job. Figure 5.9 shows the model for the selected configurations of the SGE application.

Combined model The combined model mixes Kriging and analytical model using a threshold on the value of `sig` as a switch between the models: When the confidence of predictions is too low, the combined model resort to the analytical model, otherwise it behaves like a normal Kriging model. We set the value of the accuracy threshold equal to 0.4. More details can be found in the previous chapter where we firstly introduce combined models. Figure 5.10 shows the model that we obtained using the selected training set.

These models implement different interpolation and regression techniques and they result in different response surfaces, as the previous figures highlight. In particular, multiple linear regression models result in a multidimensional plane, M5 regression trees partition the whole space in linear subspaces, multivariate adaptive splines and Kriging models interpolate the data smoothly. The analytical model results in several planes with “jagged” boundaries that smooth away from on side of the model to the other. This shape results from the particular definition that we choose for the dependent variable, i.e., the average response time: Under light loads, with empty queue, and especially for configurations that use several executors, the shape of the response is like a step function. This derives from the parallelism of the system: More executors complete jobs at the same time at the average time of the system is close to its minimum response time.

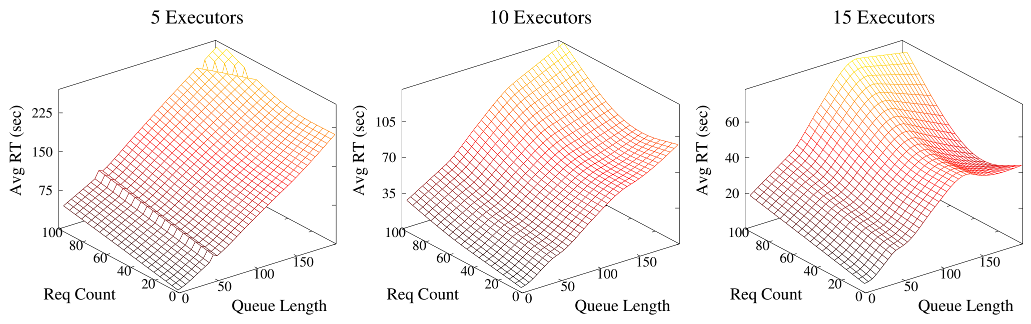

Figure 5.8. Different projections of the Multivariate adaptive regression splines for the Sun grid engine case study

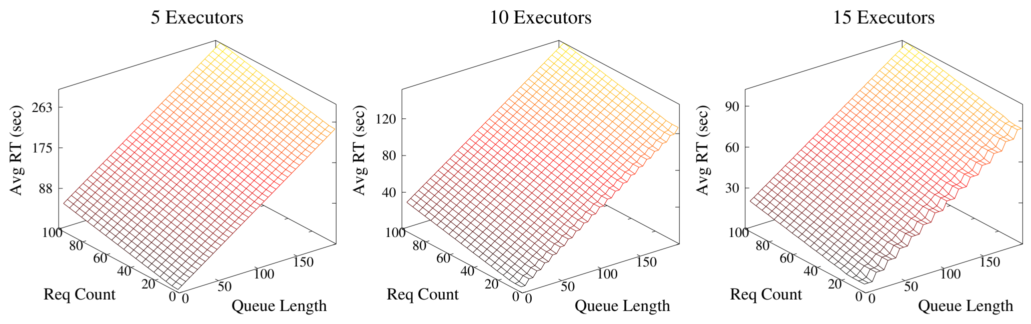

Figure 5.9. Different projections of the Analytical model for the Sun grid engine case study

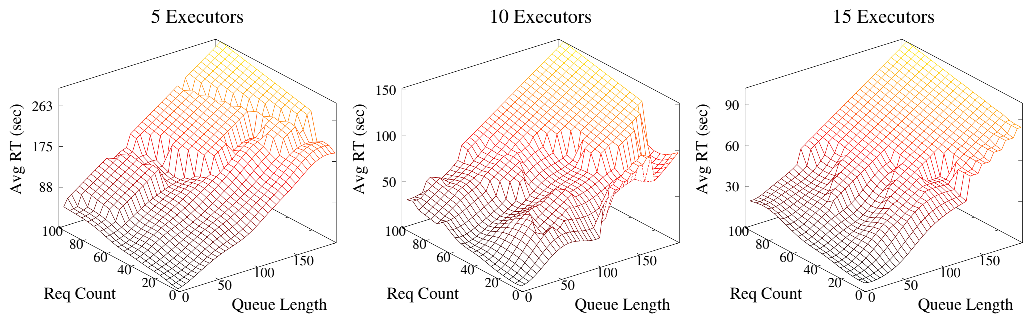

Figure 5.10. Different projections of the Combined model for the Sun grid engine case study

Under heavy load, this effect is smoothed out because more jobs stay in the queue, and the job waiting time takes over the processing time. Finally, in the combined model we can notice smooth areas that correspond to the use of Kriging model, and more linear sections where the analytical model is used.

To evaluate the accuracy of the models we compute prediction errors by following the same setup of the previous experiment: We train (when necessary) a model with the original training set, we predict the validation set, and then, we compute the MAE, the RMSE, and the MAPE. We compare the value of error metrics across models to draw conclusions about the accuracy of Kriging with respect to the other models. We summarize the results on the overall accuracy of the models in Table 5.1. In the table, the first column identifies the considered model, the second, third and the fourth columns report the values for the MAE, RMSE, and MAPE. To ease the comparison, we report here also the results that we obtained with the Kriging models.

Table 5.1. Comparison of models prediction ability

Model	**MAE** (msec)	**RMSE** (msec)	**MAPE** (%)
Kriging model	5742.56	9395.30	15.42
Multidimensional linear regression	11043.50	15194.46	43.91
M5 Regression trees	8269.63	14215.82	13.98
Multivariate adaptive regression splines	**2822.78**	**4113.11**	**9.52**
Analytical model	5166.68	7312.49	13.13
Combined model	4125.63	6165.03	10.95

We can make the following observations limited to this experiment: If we consider the MAPE alone, all the models but the multidimensional linear regression model show acceptable accuracy. However, when we consider also the other error metrics we can se that multidimensional linear regression and regression trees show relatively poor accuracy compared to Kriging models, while the remaining models show better performance.

Multivariate adaptive regression splines are the best model in the group. They show very good accuracy in this experiment because we smoothed training and validation data to model the steady state behavior of the system, and splines capture trends in the data very well. However, in the next sections we will show their limitations when we model systems on-line and in realistic settings. This strengthens our choice of using Kriging models. Combined models in particular outperform both Kriging models and analytical models that comprise them, suggesting that model combination based on `sig` is an option worth consider.

These results highlight some interesting trends, thus are a good starting point for further evaluations. They are not general because we derive them using a single partition of the available data into training and validation sets, and this may cause, especially for non-parametric models both false positive and false negative results. For this reason, we compare the models also using a different process (and a different metric).

In particular, for the comparison we run *K-fold* cross validations and then we compare the prediction errors of each model [28]. K-fold cross validation is a widely adopted technique to evaluate the quality of models, and it is generally used whenever only a single dataset is available. For example, it is used when the cost of collecting two separate sets of data for training and validation is prohibitive.

The technique starts by randomly partitioning the initial data set in K subsets, where K is a design parameter and it can vary between 1 to n, the size of the sample set. Once the subsets are ready, it proceeds by repeating the following activities for all the subsets: (i) Remove from the original dataset the chosen subset; (ii) train the model with the remaining data; and, (iii) predict the left out subset.

At the end of the process, we collect all the n predictions and compute the cross validation error (CVE) using the following formula:

$$\mathrm{CVE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.4}$$

where, y_i is the real data and $\hat{y}_i$ is the corresponding prediction. CVE is quadratic and in this example is expressed in millisecond, thus its absolute value may be of high magnitude. Nevertheless, for the sake of the comparison, smaller values identify more accurate models.

We run the cross validations increasing the value of K at each iteration. Figure 5.11 summarizes the results as bar plots: Each bar corresponds to a different model and bars are grouped according to the value of K.

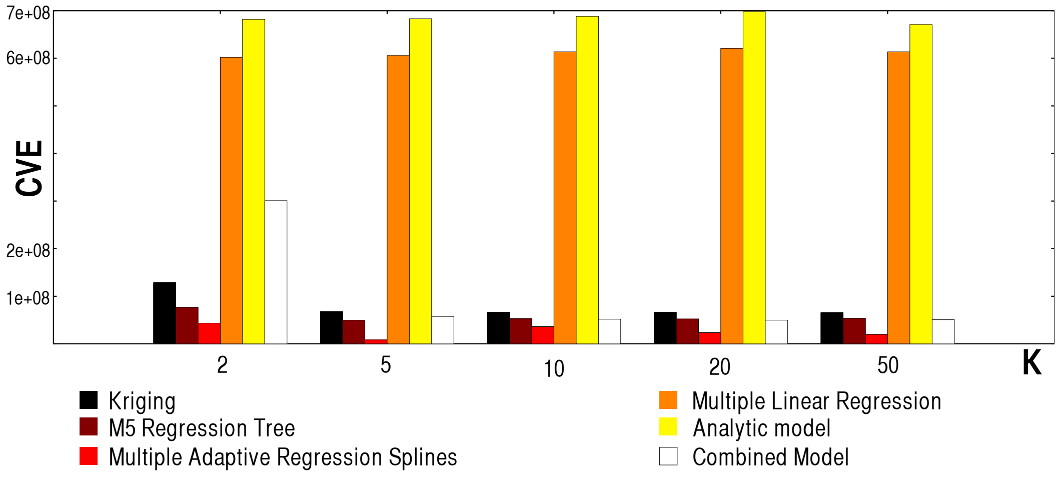

Figure 5.11. Comparison of models cross validation error

From the data in the bar plots we can make the following observations: Analytic and multiple linear regression models behave constantly worse than the other models. From the previous evaluation step, we expect this result for the multiple linear regression models, but not for the analytical model that performs worse than the expected. This can be explained considering the fact that the analytical model is mostly linear while the system may not be in some part of the input space, and this results in poor predictions. In the previous experiment, the validation data were distributed mainly over the main diagonal and this hid the "true" accuracy of the model.

The other models improve with increasing value of K as we use more data for training, and stabilize after a while to comparable values. Differences are in how fast they reach a stable value and the absolute value that they reach at the end: Between $K = 2$ and $K = 5$, Kriging models and combined models have a strong improvement while M5 regression trees have only a small improvement. Multivariate adaptive regression splines improve as more training data are used as well, but they do not show a monotonic reduction of the error that instead oscillates. We verified that this particular behavior is consistent across different runs and datasets.

With these results we can refine our previous conclusions: Simplistic linear regression models and simple analytical models may be not suitable to entirely capture the behavior of elastic applications in the Cloud. On the contrary, the other models are more suitable for this purpose. And, if they are provided with enough training data, behave comparably good to each other. Finally, we confirm the ability of the combined model to gain accuracy by suitably switching between Kriging and analytical models.

To have a different perspective on the accuracy of the models, we plot actual and predicted values against each other in a dispersion graph. These plots highlight how closely the actual dataset and predicted datasets agree, and we use them to qualitatively investigate the results on model accuracy just obtained.

Figure 5.12 shows the plots for the various models for the same value of K (i.e., $K = 20$). As reference for the interpretation, we plot the *ideal* predictor, i.e., the $y = x$ function, as thick solid line. Furthermore, to have an intuition on how globally each model behave, we also plot three bands that correspond to 10%, 20% and 30% of the value of the ideal predictor. The lines that define these bands are identified by means of different line patterns.

Thanks to these plots, we can qualitatively asses the behavior of each model by investigating the distribution of the prediction points with respect to the ideal predictor: Data close to the ideal predictor indicate accurate predictions, while data points distant from it highlight poor predictions.

From the plots in Figure 5.12, and limited to this run of experiment, we can make the following observations: i) Kriging models have reasonably good performance: around 96% of the predicted points lie within the 20% band. They show worse performances for very small and very large values of the output variable: For small values Kriging models are conservative and for large values are optimistic. This may happen because for small values of the output the model is not able to filter out the noise,

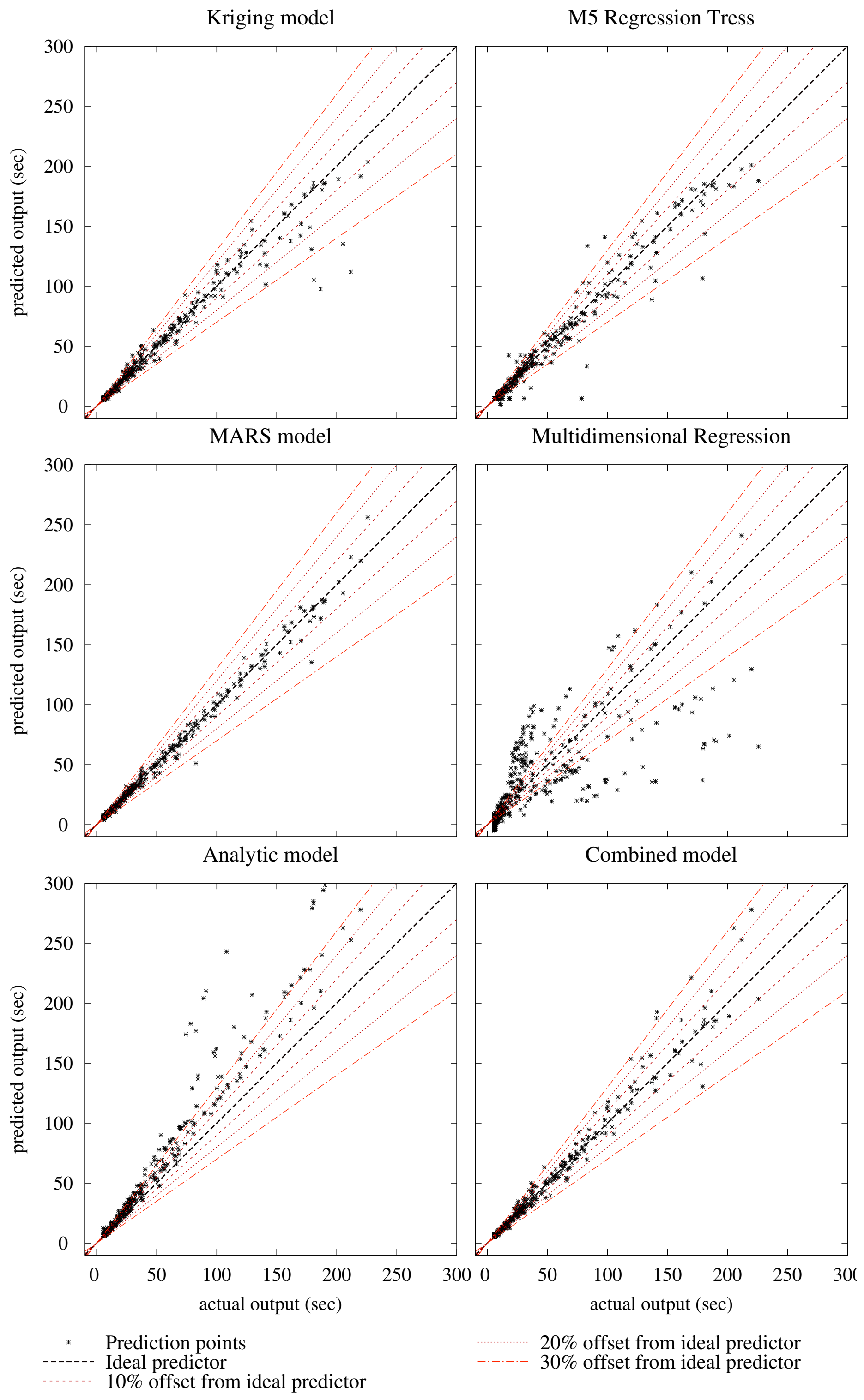

Figure 5.12. Actual vs predicted output plots for comparing alternative models

while for large values of the output the model has not enough data points, therefore is not able to correctly extrapolate information. ii) M5 regression trees show smaller accuracy than Kriging models: around 89% of the points predicted by the regression tree lie within the 20% band. From Figure 5.12 we see that regression tree shows slightly more dispersed predictions than Kriging models, thus a bigger amount of wrong predictions; however, these wrong predictions impact less on the cross validation errors than the ones of Kriging models because in the case of M5 Trees they cluster around the origin. iii) Multivariate adaptive regression splines are very accurate: around 97% of the predicted points lie within the 20% band, and almost no points lie outside the 30%. This strengthens our previous idea that splines can accurately capture the behavior of this application. With respect to Kriging models we see that splines can better smooth the noise for small values of the output, and can extrapolate better than Kriging models for large values of the output. iv) Multidimensional linear regression models are inaccurate as most of the time data points are distant from the ideal predictor: only around 15% of the predicted points lie within the 20% band. This explains the bad performance of this models according to the cross validation error. v) Analytic models are inaccurate as well, even if in a different way: around 77% of the predicted points lie within the 20% band, and all the data points are above the ideal predictor meaning that the model is very conservative. This trend is more evident for larger values of the output than smaller values, and this explains the high value of the cross validation error. vi) Combined models instead show the benefits in terms of accuracy of using Kriging models with analytical model: around 95% of the predicted points lie within the 20% band, and few points, especially for large values of the output, lie outside the 30% band. In fact, this combination can mitigate the optimistic and pessimistic behavior of the component models, and is more accurate than each single component model.

Summary of the results

From this initial the evaluation we can say that Kriging models (i) predict the behavior, i.e., the performances, of Cloud based applications with acceptable accuracy that is comparable with the one obtained using alternative surrogate models, such as regression trees and multivariate adaptive regression splines; (ii) outperform simple linear regressions and simple analytical models; and, (iii) can leverage `sig`, i.e., the measure of the uncertainty of their predictions, to accept or reject the predictions before observing of the actual predicted value. We implement combined models using the value of `sig` as switch, and we show their improved performances with respect to its component models.

We must note that the validity of these conclusions may be threatened by the fact that we performed the investigation only by considering the Sun grid engine case study and a limited amount of training data points. For more complex cases, we expect that the scalability of Kriging models and the possibility of using the `sig` to ponder the quality of predictions will be more beneficial.

5.3 Kriging models for control

Proving that Kriging models can capture the behavior of Cloud based applications in an off-line setting is a necessary but not sufficient condition to prove that the same models are also suitable in the context of autonomic controllers, where decisions may have to be made on-line, under uncertainty, and with noisy data. Therefore, we need to prove that Kriging models tolerate: noisy data, provide timely predictions that do not hindrance the reactiveness of the controller, and can be easy adapted to emerging application behaviors if they appear. We devote the next sections to prove these three major points, thus proving that Kriging models can be used as core elements of self-adaptive autonomic controllers.

Robustness of Kriging models

When Kriging models are used to model real running systems, they elaborate data collected by the available monitoring system that may be imprecise or that may introduce noise in the data. This situation is different from before where we preprocessed the data that we collected during controlled experiments at staging time.

In these new settings, we need to prove that Kriging models remain sufficiently accurate to support effective analysis, that is, they still provide good predictions of applications behavior. In this way, we can answer our next research question:

> **RQ-2**: Are Kriging models robust with respect to monitoring data noise and precision?

We adopt the following procedure to evaluate the robustness of the Kriging models in real life settings: We deploy our target application in a Cloud infrastructure, and we apply synthetic workloads to it. Meanwhile, we collect monitoring data about the controlled and uncontrolled system variables. For example, in the case of the Sun grid engine, we collet in each monitoring period the incoming workload in requests per period, the length of queues in number of jobs inside the system per period, the system configuration in number of running executor nodes, and the average response time of the requests in a period. During the experiment, we simulate an hypothetical controller by changing the system configuration using the Cloud infrastructure API.

In this way, at the end of a run we collect a trace of real monitoring signals obtained via a realistic workload and working conditions. Then, we can replay the trace against the models to test the accuracy of their predictions in front of noisy and possibly imprecise data.

In practice, we use the models to predict the system behavior using the monitored data used as input, and we compare the predicted output against the monitored output. We train the models using the same training dataset of the previous evaluation steps, and as we did before, we measure the accuracy of models by computing the mean absolute error (Eq. 5.1), the root mean square error (Eq. 5.2) and the mean absolute percentage error (Eq. 5.3).

We generate different workloads and we adopt different settings of the Cloud infrastructure to explore the robustness of the models in different working conditions. The various combinations of these factors result in varying levels of noise and precision of the monitored data.

For example, we change workload intensity and variability to stress the application, and we use both slow and fast actuators to change the system configuration. In particular, *slow* actuators are the real actuators provided by the Cloud infrastructure. We call them slow actuator because they deploy and boot from scratch a new virtual machine every time that we invoke them. On the contrary, *fast* actuators reuse running virtual machines in a hot-replica fashion, thus saving time. Both the configurations represent valid and realistic settings: Slow actuator are the common case for IaaS infrastructure, while fast actuators can be found in hybrid or private Clouds.

We must note that, in any configuration, the actuators and the monitoring system are far from being ideal. In fact, while the Cloud infrastructure is changing the system configuration, the monitored values differ from the real ones because the monitoring system considers a new VM to be in running state as soon as the Cloud infrastructure allocates all the required physical resources to the VM, even if, at the application level, it is not yet processing any user request. The slower the actuator the bigger is the time needed to implement the system configuration, and the more will be the inaccuracy in the monitoring data.

We use Apache JMeter[5] to generate the workload for the Sun grid engine starting from predefined workload trace files, and a simple rule based controller to change the system configuration. To simulate realistic constraints in the resources that we can allocate, we force the system to use at maximum eighteen executor nodes.

Figure 5.13 shows the traces collected during the first experiment where we use the rule based controllers paired with fast actuators, while Figure 5.14 shows the traces collected during the second experiment where we use the same controller but with slow actuators. We collect new monitoring data every five seconds, which is the frequency of our monitoring system. In both figures, the first three plots refer to monitored *input* variables: The system configuration (first plot) expressed in terms of the number of running executor nodes, the length of queue (second plot) expressed in terms of requests inside the system, and the request count (third plot) expressed in terms of new requests received by the system in each monitoring interval. The plots at the bottom instead refer to the monitored *output* variable, i.e., the average system

[5] `http://jmeter.apache.org/`

response time, that it is the signal we need to predict using the models.

We obtained the first data set with fast actuators and by running three different workloads in a row: The first workload (`W1`) starts at the beginning of the experiment and lasts around third minutes. It generates two waves of requests of different amplitude. The second workload (`W2`) starts right after (around sample number 2000), and lasts for one hour and half. It is a step function with different slopes for the increasing and decreasing sections. After this, the last workload (`W3`) starts and lasts around two hours, till the end of the run. It generates small steps of different intensity. We obtained the second data set with slow actuators and by running three times an oscillating workload.

Table 5.2 summarizes the results about the accuracy of Kriging models using the first and second data sets. Each row of the table refer to a data set, and the columns show the aggregated error metrics that we computed over each run.

Table 5.2. Robustness of Kriging models.

Data set	**MAE** (msec)	**RMSE** (msec)	**MAPE** (%)
1) Fast actuators and heterogeneous workloads	18712.58	44990.96	29.96
2) Slow actuators and oscillating workloads	15918.71	22427.25	47.08

Compared to the results obtained in the previous experiment, we notice a general degradation of the accuracy of Kriging models. In general, we expected to see such outcome because the data monitored from the running systems are inherently more noisy, and because by dynamically changing the configuration we introduce some imprecisions in the data as we explained before.

To understand the implications of the realistic settings on the accuracy of Kriging models, and to better interpret the results of Table 5.2 we analyze the results of the experiment separately.

In the first experiment, we use fast actuators that we assume introduce very little imprecision in the data, therefore we expect a low impact on model accuracy. This however is in contrast with the data reported in the table for the first experiment. To explain the reasons of these results, we need to break down the experiment into each workload specific segment, and then, analyze them separately.

Table 5.3 shows the break down of the results: The first row refers to the error metrics computed over the entire experiment and is reported to contextualize the other figures. The following three rows show the errors computed over the different workloads that comprise the data set. It must be noted that each of the segments has different duration, thus aggregate values of the first line do not correspond to the mean value of the other figures.

According to the values in Table 5.3, for workloads `W1` and `W3`, Kriging models show reasonably good performances in terms of MAE and RMSE. For these two cases, Krig-

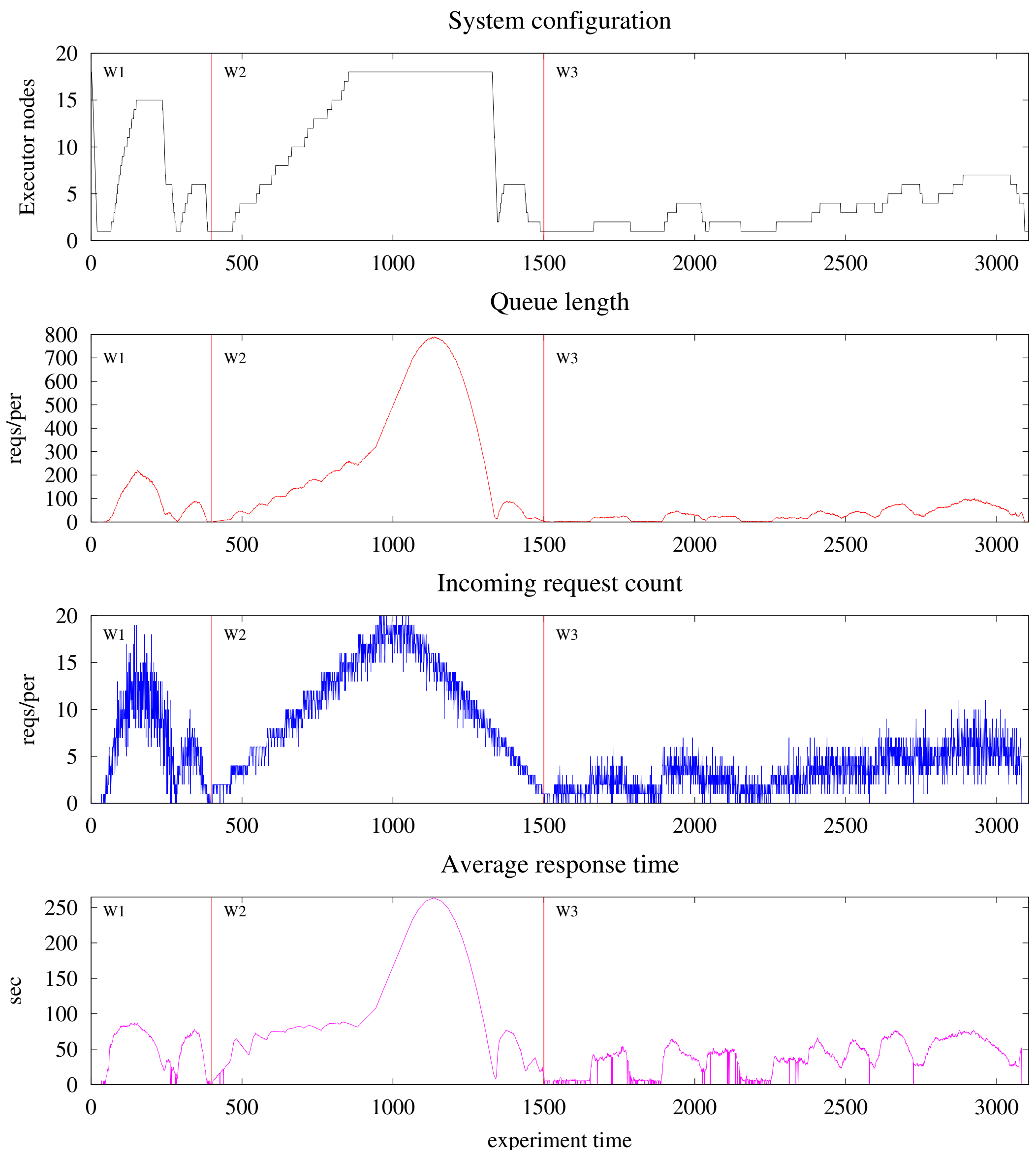

Figure 5.13. Dataset 1 for the evaluation of Kriging model robustness.

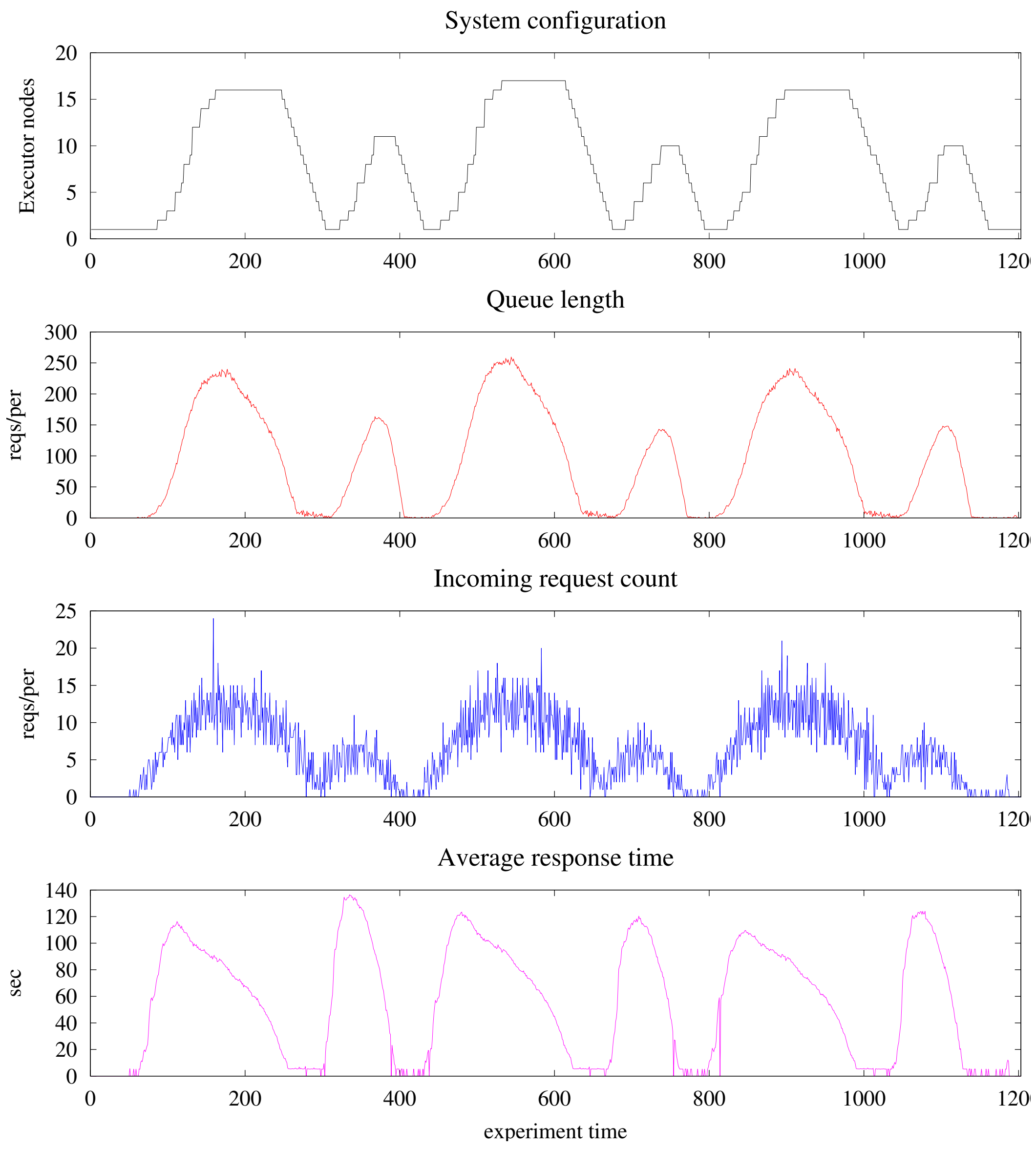

Figure 5.14. Dataset 2 for the evaluation of Kriging model robustness.

Table 5.3. Robustness of Kriging models: Break down of the first dataset.

Workload Type	**MAE** (msec)	**RMSE** (msec)	**MAPE** %
All	18712.58	44990.96	29.96
W1	5142.28	7035.68	14.19
W2	41315.93	72554.12	27.31
W3	4756.58	5528.42	35.59

ing models show the improvement over the case of slow actuators that we expected. If we look at MAE and RMSE, the model performs better with workload `W3` than with workload `W1`. This happens because workload `W1` has fast variations that trigger several changes of the system configuration, and this results in a degraded quality of monitoring data. Differently from this, workload `W3` has smaller intensity and fewer variations, thus resulting in more stable system configurations, less noisy monitoring data, and more accurate predictions. However, the small intensity of the load leads to a bigger value for the MAPE of `W3` than `W1`.

On the contrary, for workload `W2` we notice worse results. Workload `W2` generates a load that overcomes the maximal capacity of the system, i.e., the capacity of the system when all the available resources are allocated to it. We can see this in Figure 5.3 between samples 4500 and 6500. In this interval, we can see that the queue of waiting jobs (as well as the average response time of the system) increases over time, meaning that the system capacity is not enough to process all the jobs that arrive.

This is an unusual situation, and at training time we did not experience. As consequence, we did not collect any data about the behavior of the system during the staging time, and the Kriging models that are suited more for data interpolation than extrapolation, fail to capture the correct system behavior in this particular context.

We plot in Figure 5.15 the predictions of the Kriging models (dashed line) against the monitored output (continuous line) for the first validation set. By looking at the plot we can intuitively see how the Kriging models behaves in the different cases. For reference, we report also the evolution of the prediction error at the bottom of the figure. Both at the beginning (workload `W1`) and at the end (workload `W3`) of the run, we see that the signal predicted by the Kriging model is very close to the actual one. In the middle (workload `W2`), we notice that the predicted signal is close to the actual one at the beginning of `W2`, then suddenly it diverges while remaining constant for a while; finally, it recovers to the actual signal again.

We can explain the interval in which the predictions remain constant by considering the default choice of adopting a constant trend, usually zero or the empirical mean of the data, inside the Kriging model. We know that away from the training region Kriging models exploit this trend to make the predictions in absence of any other evidence. So by providing a constant trend we obtain constant predictions. We could also force

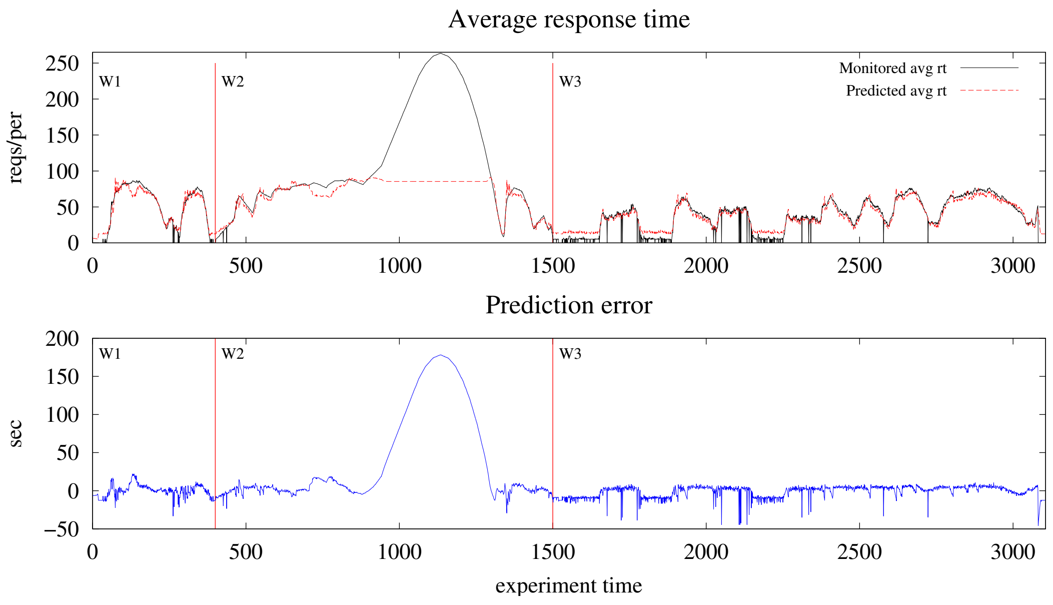

Figure 5.15. Kriging model predictions and prediction error on Dataset 1

other trends, but this assumes some *a-priori* knowledge on the behavior of the system that in general may be not available, indeed we avoid this option.

In summary, if we consider only the workloads `W1` and `W3`, we can conclude that Kriging models are robust enough to be used even for deployed systems and not only exact computer simulations.

This is valid provided that monitoring signals lies within the boundaries defined by the training samples. Regarding, workload `W2` we can say that if we collect more data at staging time we may increase the final quality of the Kriging models. Later in the document, we will show the improvement of Kriging models accuracy when we retrain the models on-line with the monitoring data collected form the running system.

It must be noted that suitably training a surrogate model is a common challenge. In fact, blindly relying on their predictions may be problematic, especially in case we did not explore sufficiently the input/output space during training. In our favor, we can remark that Kriging models are able to estimate the quality of their predictions in relation to the training data set, using the `sig` value. This allows more informed decisions about the need of more sampling data, and enables strategies that account for uncertainty in the model. For example, we use `sig` to switch from Kriging to an analytical model whenever the prediction accuracy is not satisfactory.

In the second experiment, we use slow actuators that we assume introduce imprecision in the data, therefore we expect a big impact on model accuracy than before. This is in line with the data reported in Table 5.2

To have an intuition of the effects resulting from using slow actuators with plot the average response time, the prediction that we obtained by Kriging model, and the corresponding prediction error in Figure 5.16. From the plot is evident that the model underestimates the value of the response time during scale up, while overestimates it during scale down. In particular, we see that the model shows a bigger error during scale up actions than during scale down. We can explain this by considering the specific implementation of the actuators used during the experiments: Slow actuators start several virtual machine instances in parallel while scaling up, but stop one virtual machine at each time while scaling down. This behavior was implemented by design to enable fast scaling-up as reactions to peeks in the load. The asymmetry of this behavior impacts differently on the quality of the predictions during scaling up and scaling down.

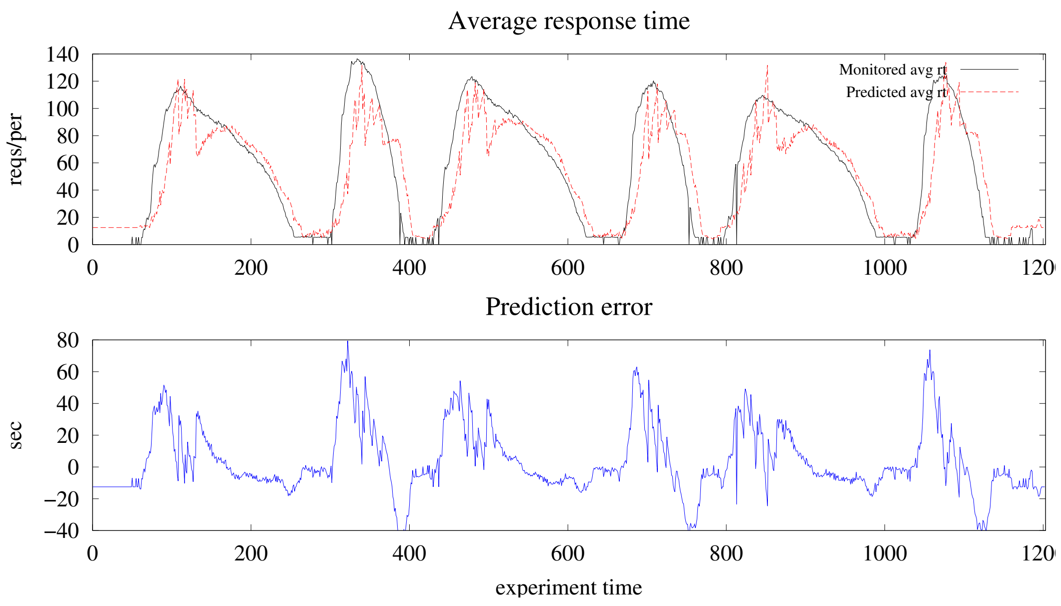

Figure 5.16. Kriging model predictions and prediction error on Dataset 2

Comparison with other models

To better contextualize the results obtained with Kriging models we repeat both the experiments with the other models, and we compare their error metrics.

Table 5.4 summarizes the obtained results: Rows correspond to models under investigation; for each model we report MAE, RMSE, and MAPE errors. Columns instead refer to the dataset used for the evaluation. In particular, we break down the first validation set and we report both the aggregated and the workload specific results. We round the values to seconds to simplify the reading. Finally, to ease the comparison, we report in the table also the results obtained with Kriging models.

By comparing the accuracy measures of the models in the first dataset (Dataset 1) we can see that Kriging models and combined models always outperform all other

Table 5.4. Comparing model robustness

Model		Dataset 1				Dataset 2
		All	W1	W2	W3	
Kriging model	MAE (sec)	18.71	**5.14**	41.32	**4.76**	15.92
	RMSE (sec)	44.99	7.04	72.56	**5.53**	22.43
	MAPE (%)	29.96	**14.19**	27.31	**35.59**	**47.08**
M5 Regression trees	MAE (sec)	36.77	17.12	69.50	16.56	23.39
	RMSE (sec)	65.64	22.45	101.73	24.87	31.32
	MAPE (%)	44.47	31.20	52.35	41.56	51.03
Multivariate adaptive regression splines	MAE (sec)	36.73	8.18	85.00	6.83	20.12
	RMSE (sec)	85.47	11.20	137.86	10.61	25.40
	MAPE (%)	67.64	29.76	62.89	79.94	56.36
Multiple linear regression	MAE (sec)	50.17	29.48	102.40	15.50	48.01
	RMSE (sec)	91.83	37.14	145.95	18.26	56.44
	MAPE (%)	59.77	52.63	77.83	47.76	107.72
Combined model	MAE (sec)	**4.91**	5.32	**5.0**	**4.76**	16.05
	RMSE (sec)	**5.75**	**6.55**	**5.77**	**5.53**	22.31
	MAPE (%)	**23.55**	14.58	**10.34**	**35.59**	47.62
Analytical model	MAE (sec)	15.80	16.46	11.87	18.61	**13.02**
	RMSE (sec)	19.13	21.24	15.43	21.05	**16.92**
	MAPE (%)	51.19	38.82	28.17	71.43	54.02

models, both in the aggregated and in the per-workload cases. This confirms our previous findings about the use of Kriging and analytical models together, and further highlights the benefits of such combination. The biggest improvement derives from the extensive use of the analytic model during workload `W2`, where we have a reduction of all the error metrics. Furthermore, in this experiments we do not let Kriging models to learn the behavior of the system by collecting additional monitoring data. In the next section, we will see that by updating the Kriging models this difference vanishes, as Kriging models reach similar accuracy to combined models.

We obtain further improvements in workload `W1`, while there are no improvements for workload (`W3`). This happens because all the predictions are obtained from the Kriging model that is already accurate enough.

By comparing the accuracy measures of the models in the second dataset we see that all the surrogate models but the multiple linear regression perform comparably good. In particular, Kriging models have the smallest MAE and MAPE, while the combined models have the smallest RMSE. We note that in this dataset analytical models perform better than the surrogate models according to MAE and RMSE. This can be explained by considering that the experiment puts the system in a region where inputs and output are linearly correlated.

If we limit our attention to workloads `W1` and `W3`, and compare the results obtained against the ones of the second experiments, we notice the degradation of the accuracy for all the models. This confirms that the use of slow actuators to change the system configuration introduces more noise in the monitoring data, and that decreases the quality of the data used by the models, and the accuracy of their predictions. Kriging and combined models behave consistently better than the other surrogate models, and in particular they behave similarly. This happens because, the oscillating workload of the second experiment takes the system always within known working conditions, therefore the combined models reduce more or less to "pure" Kriging models, and lose the relative gains that derive from the use of analytical model where predictions of Kriging models are poor.

In summary, considering the results presented so far, we can conclude that combined models are the best choice if one faces working conditions that were not considered during the training activity, with Kriging models coming after them.

The comparison of "pure" surrogate models highlights the profound differences between models for extrapolation, i.e., the regression trees, and models for interpolation, i.e., Kriging models and adaptive splines, and also the differences between the two interpolating models.

If we focus only on the results obtained by the models in known working conditions, we can draw some conclusions about their relative robustness, i.e., their ability of tolerating noisy and inaccurate data: Kriging models, combined models and regression trees tolerate the noise better than splines. Therefore, Kriging models, combined models and regression tress are more suitable to be used for modeling running systems *on-line* while regression splines should be the preferred choice for modeling system offline. Of course, this conclusion assumes that an analytical model is available, otherwise the combined model cannot be implemented.

We continue our evaluation to investigate the behavior of the models with respect to the other two main requirements that are important in the context of self-adaptive autonomic controllers: promptness of the models and adaptation abilities.

Promptness of Kriging models

Model-based controllers may inspect models more than a hundred times during a single control cycle. For example, they may use models for the analysis of monitored data and for planning the next system configuration to implement. This requires that models, in addition to be accurate, provide their predictions in timely fashion. In this way, controllers may have enough time to perform their activities without being stuck waiting for predicted values from the models.

Self-adaptive model-based controllers have also the possibility to retrain their models at runtime by elaborating new monitoring data. In this way, controllers can adapt their models as the system evolves, improving the accuracy of their predictions. This requires that models are easy to adapt and fast to train, update or adapt. If models are fast enough, controllers can always have the most updated representation of the system behavior without waiting a long training time that may harm controller's responsiveness.

Therefore, we need to prove on one side that Kriging models can make timely predictions, and on the other side that they can be trained at runtime with no (or small) impact on the control frequency. In this way, we can answer the third research question:

RQ-3: Are Kriging models fast enough to fit the self-adaptive control loop?

We adopt the following procedure to evaluate the time for *training* and *querying* Kriging models: We run K-Fold cross validation for various values of K and for each of the folds we measure the average time for training Kriging model, the average time for querying the model, and the cross validation error. After we elaborate all the folds for the same value of K, we compute the average values across them. By varying the value of K we can create training sets of different sizes and study how this impacts on training and querying time. We consider also the accuracy measure, i.e., CVE, to understand if the accuracy of the models improves as we use more data to train them.

To make predictions, Kriging models correlate the points to be predicted with all the data used during the training. This process has a theoretical complexity of $O(n^2)$, where n is the number of samples used for the training. We are interested in evaluating how the query time evolves as the number of samples (n) changes, and specifically, if it can harm controller reactiveness. This is important to study because in practice autonomic controllers collect new monitoring data and they may drop old monitored values, thus constantly change the size of their dataset.

The theoretical complexity of the training is dominated by the complexity of a matrix inversion operation that is $O(n^3)$, where n is the size of the training set. Again, we change the size of the training set to study the evolution of the training time.

We repeat these experiments using two different datasets: In the first experiment we used the original training dataset of 636 samples that we obtained by preprocessing the data collected at staging time. In the second experiment we build a bigger dataset that contains data collected while monitoring the running system; in particular, we do not perform any preprocessing nor smoothing of these data before using them to train the Kriging model.

Table 5.5 summarizes the results for the Sun grid engine case study that we obtained using a single-vCPU virtual machine running Ubuntu 10.10 and Octave 3.2.4 on a dual-core Mac Book Pro. The first column of the table identifies the dataset used for the evaluation, the second reports the size of the training set, and the other columns report respectively, the average training time (*T time*), the average query time per single query (*Q time*), and the cross validation error (*CVE*).

From the results presented in the table we can conclude that the query time of the Kriging models, in the order of few milliseconds, is compatible with the control period of controllers used in Cloud computing that is in the order of seconds. According to

Table 5.5. Promptness of Kriging models for the Sun grid engine case study

Dataset	Sample size	T time (msec)	Q time (msec)	CVE (sec^2)
Dataset 1	364	61.36	0.09	96.66
	582	82.30	0.22	50.32
	655	93.42	0.28	46.69
Dataset 2	1553	452.25	2.62	8.68
	2485	1507.66	6.45	8.02
	2796	2027.51	7.93	7.87
	3044	2577.10	9.41	7.71
	3100	2703.49	9.84	7.64

these data, Kriging models are suitable for an on-line usage to support both analysis and planning of autonomic controllers. Regarding the training time, we can see that it grows as the size of the training set increases, as expected. For this reason, we cannot say in general that Kriging models can be always retrained on line. In fact, if a controller keeps **all** the monitoring data, eventually the training time of the model will harm its responsiveness. Conversely, the average accuracy of the model increases as we use more and more training samples.

In reality, saving all the monitoring data in not needed, nor suggested, for two reasons: If we keep all the monitoring data in the model we are giving the same importance to old and new data. In the context of dynamic environments, this may not be the best approach because it may fail to identify new behaviors, as they are hidden, or at least smoothed out, by other historical data. Furthermore, not all the data have the same importance. In fact, a new monitoring sample does not always add new information to the model, and accumulating data for very similar working conditions may not give any advantage over a certain limit.

In summary, we can say that Kriging models can be retrained on-line, if we can assume the presence of an higher level management of the training data that filters out, or somehow pre-elaborates, the monitoring data.

We must also note that there is not always a strong need to retrain Kriging models at every control cycle. In fact, unless the system evolves faster than the control period we can tolerate less frequent updates of the model. Last but not least, we must say that these experiments consider only the (theoretical) worst case scenarios. In fact, we train Kriging models from scratch at every run, meaning that we perform the matrix inversion and the hyper-parameters optimization all the times. If we assume than the system is evolving slower than the retraining cycle, we can avoid to (re)optimize the hyper-parameters **and** the full matrix inversion, thus achieving a complexity of $O(n)$ [90].

Comparison with the other models

We repeat the same evaluation for the other models and we compare their results with the ones obtained with Kriging models. In this evaluation, we do not consider multiple linear regression models nor analytical models.

Table 5.6 summarizes the results that we obtained for the two datasets considered. For reference we report also the measures obtained with Kriging models.

The first column reports the size of the training samples, while the other columns report the average training time (*T time*), the average query time per single query (*Q Time*), and the cross validation error (*CVE*) for each of the models. In the table header, we abbreviate names of the models: M5RT stands for M5 Regression trees and MARS stands for multivariate adaptive regression splines.

Table 5.6. Comparison of model promptness for the Sun grid engine case study

Samples	Kriging			M5RT		
	T time (msec)	**Q time** (msec)	**CVE** (sec^2)	**T time** (msec)	**Q time** (msec)	**CVE** (sec^2)
364	61.36	0.09	96.66	530.48	0.31	72.36
582	82.30	0.22	50.32	755.90	0.31	63.95
655	93.42	0.281	46.69	875.84	0.34	61.95
1553	452.25	2.62	8.68	1916.96	0.51	6.90
2485	1507.66	6.45	8.02	2358.45	0.56	6.30
2796	2027.51	7.93	7.87	2655.19	0.58	5.94
3044	2577.10	9.41	7.71	2740.89	0.63	5.96
3100	2703.49	9.84	7.64	2697.09	0.98	5.89

Samples	MARS			Combined		
	T time (msec)	**Q time** (msec)	**CVE** (sec^2)	**T time** (msec)	**Q time** (msec)	**CVE** (sec^2)
364	8098.66	0.47	13.78	117.53	0.31	209.83
582	11438.00	0.61	39.90	135.82	0.64	49.56
655	12717.91	0.63	19.27	145.89	1.08	44.57
1553	6149.44	0.23	10.56	506.03	2.66	8.86
2485	7134.90	0.24	15.26	1515.96	6.51	8.18
2796	7474.14	0.25	15.23	2075.40	8.09	7.90
3044	8322.42	0.37	14.12	2638.69	10.20	7.73
3100	8821.75	1.69	13.78	2789.90	19.09	7.63

The table highlights important differences in the promptness of models as the amount and quality of training samples vary.

If we focus on query time, we see that all the models have the ability to make predictions within few milliseconds suggesting that all can be used on-line to support the analysis and planning activities of the controllers.

We note that regression trees and splines have an almost constant query time while the one of Kriging and combined models degrades as we use more samples for training. This can be explained by considering that both trees and splines, once trained, need a constant time to use their internal elements, i.e., rules in the tree and basis functions in splines. Spline however are not consistent across datasets, remarking again they sensitivity to the quality of training set. Differently, Kriging models consider all the training samples and the query time increases with the size of the training set. However, they are more consistent across datasets.

If we focus on training time, we see that all the models become slower if more samples are used for training, but the time spent by each model in training activities is different. In particular, Kriging models, combined models and regression trees remain in the order of seconds, while splines that reach almost ten seconds.

If we focus on CVE, we notice that all the models but the splines show a monotonic improvement of the quality as we use more samples for the training. For splines therefore it is not always justified the use of more data. As there are no means to predict if new data will improve the quality of splines, this makes splines difficult to rely upon in an on-line self-updating setting.

Given these results we can conclude that Kriging models and combined models can be retrained on-line with negligible impact on controllers reactiveness even if they belong to the “critical path” of the controller, i.e., if the controller retrains them inside the control loop. Instead, for the other models we cannot say the same.

Adaptability of Kriging models

We showed that Kriging models are robust with respect to monitoring data and are fast enough to be used on-line and adapted at runtime. The missing piece to prove that we can suitably use them as core components of model-based self-adaptive controllers is to show that Kriging models can adapt to the changing system behavior. In this way, we can answer the fourth research question:

> **RQ-4**: Can Kriging models adapt to the behavior of running applications by using the data passively monitored from them?

The need of adaptation in model-based self-adaptive controllers that employ surrogate and black-box models derives from two different roots: on one side, their initial training set may be not optimal, and on the other side, controlled systems may change at runtime. Therefore, it is important to understand how Kriging models perform in both situations and evaluate, on one side, how they improve over time with respect to an initial training set, and on the other side, how they can track changes in the external behaviors of the controlled systems.

Improvement of Kriging models accuracy over time

We adopt the following procedure to evaluate how Kriging models improve over time: We simulate the behavior of real systems using the traces that we collected during system runs. And we mimic the knowledge management process that self-adaptive controllers implement, by periodically retraining the Kriging models with monitored data. After each retraining we predict the behavior of the system, and we compute the mean absolute error and root square mean error. At the end of the run, we collect all these metrics and we study how they evolved over time: If they decrease, we can say that Kriging models are effectively improving. To have a global picture, along with the error metrics we monitor how the size of the training set and the corresponding training time evolve.

We evaluate the adaptability of Kriging models by computing the error metrics along two different validation sets that for reference we call *future* validation set and *total* validation set. The future validation set contains the data that appear, in time, after a model retraining, and it is meant to evaluate how Kriging models perform against non periodic (or repetitive) working conditions. This validation set mimics the behavior of a controller that learns the models as it goes. The total dataset, instead, contains all the data from the original trace and it is meant to evaluate how Kriging models perform whenever the same, or similar, working conditions appear. This validation set gives an intuition about the quality of the model if we considered the additional monitored data at the beginning; in a sense, it shows how the situation would be if we collected more data at staging time.

We repeat the same process under different experimental settings. For example, we change the frequency of model retraining (retrain period), the amount of data that are retained for each input configuration (retain count), and the aggregation strategy to preprocess them. We evaluate the results obtained with the following configurations:

- **No learning** This is the baseline for the comparison and we do not adapt Kriging models at all.
- **Raw (retain count - retrain count)** In this configuration we keep only a number of samples for each input vector that is equal to the retain count parameter, by replacing old samples with new ones, and we retrain the model at the frequency specified by the retrain period. For example, we run experiments using the `Raw (2-10)` configuration that keeps only the last two samples for each input vector and retrains the model once every ten cycles. We experiment also with the `Raw (5-10)`, `Raw (2-50)` and `Raw (5-50)` configurations to evaluate the effects of keeping a different amount of samples and to retrain the models at different frequencies.
- **Aggregated (retain count - retrain count)** This strategy is similar to the previous one. However, we preprocess the monitoring data by computing the average configuration that the system reached while implementing control actions. We chose this particular aggregation because we sample data at the beginning of

monitoring cycles, that is, when incoming requests are received, but the system may change its configuration during their processing, thus altering their theoretical average response time. We used the following configurations during the experiments: `Aggregated (2-10)`, `Aggregated (5-10)`, `Aggregated (2-50)` and `Aggregated (5-50)`.

We also repeat the experiment under different working conditions by changing the synthetic workloads and the implementation of Cloud actuators used during the collection of system traces. Figure 5.17 shows the three datasets that we use for this evaluation. As before, we deploy the Sun grid engine application in the Reservoir Cloud infrastructure and we collect traces from the monitoring system. The figure shows for each data set the evolution of the system configuration along with the synthetic workload that lead to it. We omit for brevity the signals of the other metrics of interest (queue length and average response time). The first and the last dataset are the same that we used in section 5.3 to evaluate the robustness of models, while the second dataset is new.

The first dataset applies to the system an heterogeneous workload, and we chose it to evaluate the ability of the model to adapt to possibly unknown conditions. The second dataset applies to the system a periodic workload, and we chose it to evaluate the ability of the model to improve in time whenever the application is subject to known working conditions. The third dataset applies to the system a different periodic workload, however this time with use slow actuators to change the system configuration; we chose this dataset to understand the impact of low quality data on the adaptation ability of the models.

For brevity, we report only a subset of the results. In particular, we omit the runs with retain count of two samples and with retraining period of fifty monitoring cycles.

We presents first the results about Kriging models adaptability in each dataset, and then we summarize the conclusions

Figure 5.18 shows the results for the first dataset. The first two plots report the evolution of the size of the training set and the corresponding training time, while the other two show the evolution of the mean absolute error computed over both the total and future validation sets. We do not report the evolution of the root mean square error because it follows a similar evolution to the mean absolute error.

We observe that the size of the training set grows similarly both from the raw and aggregated configurations, while (of course) it remains constant for the no-learn configuration. Accordingly the training time for the raw and aggregate configurations increases while it remains stable for the no-learn configuration. The size of the model increases almost linearly in the first half of the experiment as a consequence of monitoring unknown working conditions. Starting from the middle of the experiment we notice that this trend changes, as the model experiences known working conditions. The training time follows a similar curve, and we note small oscillations between the raw and aggregated configurations that are due to the different values of the training

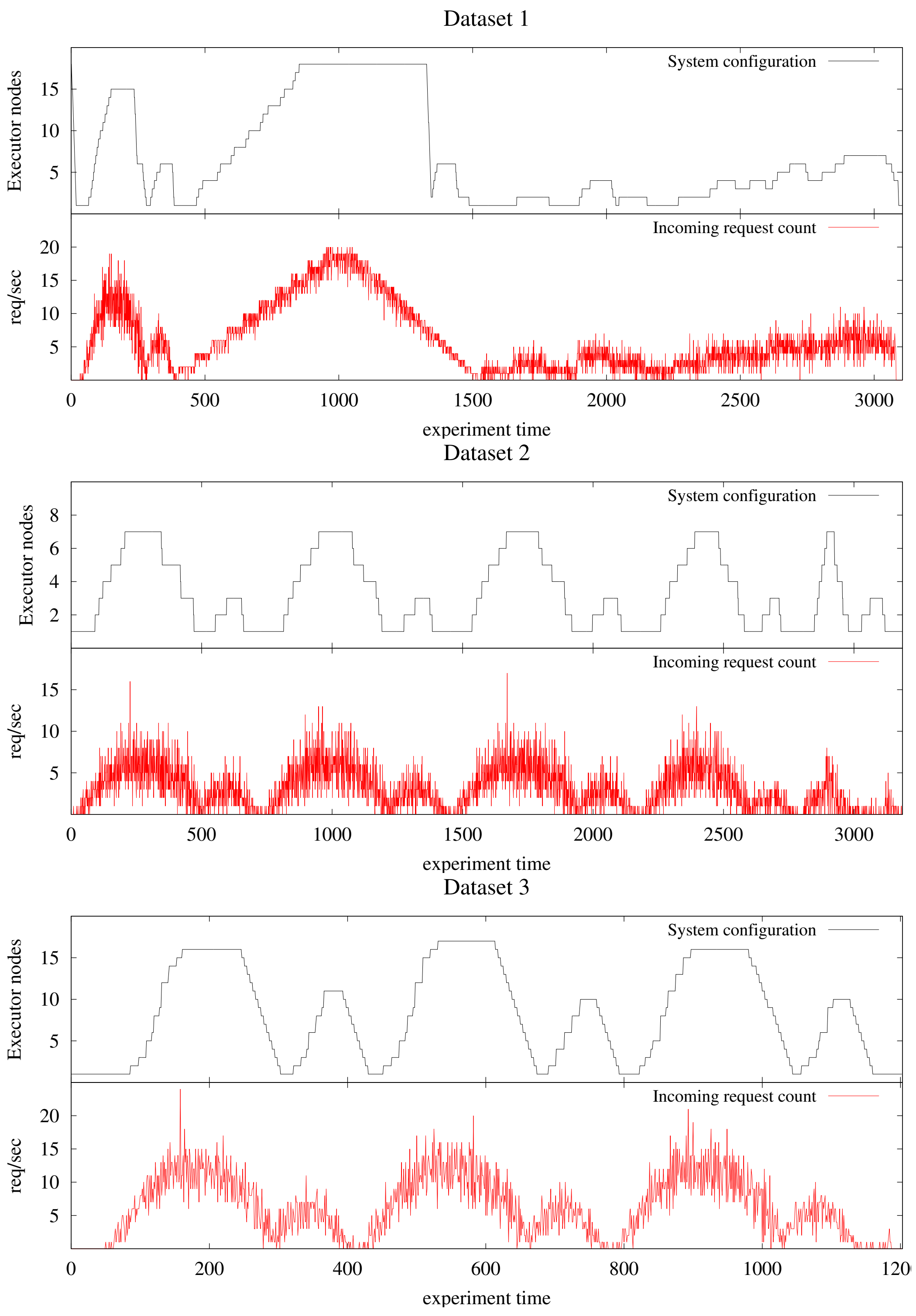

Figure 5.17. Data sets for evaluating the improvement of Kriging models over time.

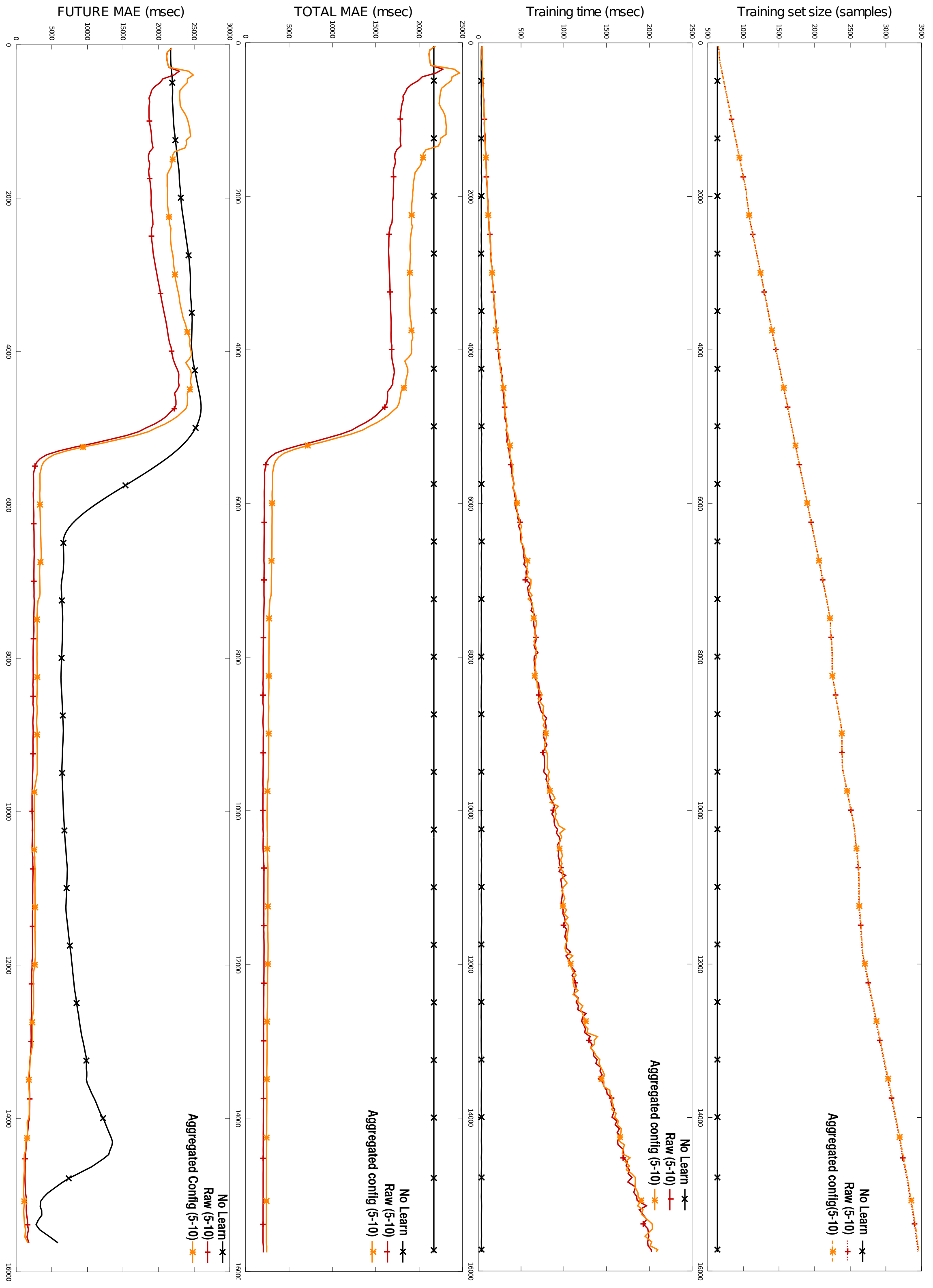

Figure 5.18. Adaptation of Kriging models: Heterogeneous workload and fast actuators.

samples. The important point is that even in the worst case the model can be retrained in at most two seconds.

By looking at the Total MAE we see that it is constant for the no-learn configuration while it decreases in the other configurations. We notice that at the very beginning of the experiment there is a degradation in the quality of the adaptable models with respect to the baseline, after which the errors decrease, and reach a stable value. This is a transitory effect of quick and intense changes in the system configuration that passes from one to eighteen executors in a short time and introduces noise in the monitored signals that are absorbed to some extent inside the model. Furthermore, such big changes tend to temporarily destabilize the normal behavior of the system causing bigger prediction errors than usual. After this transitory phase ceases, we see an almost monotonic decrease of the error. In particular we notice two facts: First, there is a big drop in the error, and second, the error stabilizes around its final value. The big drop is a consequence of the working conditions that stress the system over its limits (see the discussion in Section 5.3 about the model robustness against workload `W2`): First we took the system in extreme conditions where we have no training data, and consequently the prediction error is considerable. Then, as soon as we receive the corresponding monitoring data, the model learns the unknown behavior and quickly improves. The stability period after that confirms that the model learned and remembers the behavior of the system. We remark that the model error cannot be completely eliminated due the presence of measurement and system noise.

If we look at the MAE computed over the future validation set we can see some similarity with the previous case: We see the increase in the error at the beginning, then the drop and the stability. There are also differences. First of all the no-learn configuration does not have any constant behavior. This is due to the relative nature of this validation set: At the beginning of the experiment, the error is averaged over the whole experiment and tends to be more stable, while at the end this is not true anymore because we can consider less samples for computing the error. Noticeably, we have a drop in the model error also for the no-learn. Again, this is a consequence of the relative nature of the error: Before the drop, the model shows a considerable error due to its non optimal training, while after the experiment passes workload `W2`, there are no more unknown (and extreme) working conditions, causing the error drop. For similar reasons, we see that error increases at the end of the run as the effect of having few samples to compute the average value of the error. In any case, we see that the learning configurations improve over the baseline configuration, and models do not have a noticeable error increase at the end of the run.

Figure 5.19 shows the results that we obtained using the second dataset.

We see that the model behaves similarly as before also if we use a periodic behavior. In particular, the size of the training set for the adapted models increases over the experiment, but it remains smaller than the previous dataset. Consequently, the training time increases up to its maximum value, that for this experiment is around

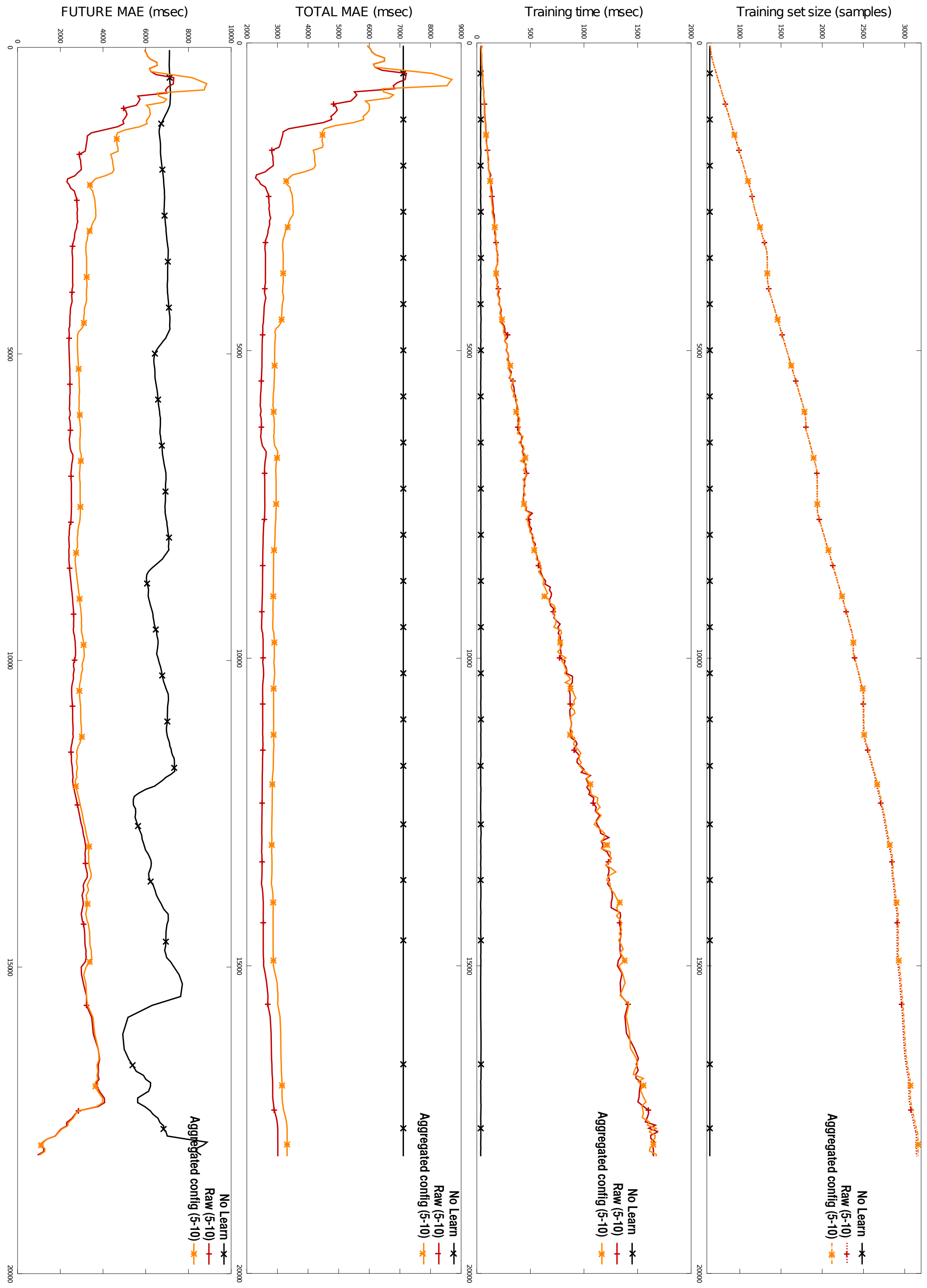

Figure 5.19. Adaptation of Kriging models: Periodic workload and fast actuators.

one second and half. We explain this growth of the training set by considering two points: First, the periodicity of the working conditions does not assure that monitored signals result in precisely the same values at each repetition because the monitoring system, the workload generators, the platform and the applications are real, not ideal. Second, we use a retain count equal to five and the strategy that updates the model in this experiment keeps all the data.

The Total MAE shows that when adapted the model does improve over time. At the first iteration of the workload we see an increase of the error, that is quickly followed by a decrease before becoming stable. The increase of the error is again a consequence of the sudden change of the system configuration, and its decrease is due to the model adaptation. Once this behavior is learnt, the error stabilizes. In the other iterations, we can see a similar behavior even if it is reduced in intensity. In the last repetition, the system experiences some transitory effects that we imputed to the Cloud platform and that change its behavior. We can see that the model learns the new behavior. As a consequence, the Total MAE increases, as we try to predict the "old" behavior with a model trained with the new one.

The Future MAE shows better this situation. We can see from its plot the periodic behavior of the working conditions, and we can also see the effect of computing the relative error by looking at the increasing intensity of the steps between iterations of the no-learn configuration. For the other configurations, we see that the error decreases sharply at the beginning and then it stabilizes. In particular, if we look at the end of the experiment, we see that the error increases a bit, and then sharply drops. If we compare this with the no-learn configuration, that is trained on the “old” behavior, we can have an intuition on how the system is changing and how the model is adapting to it.

Figure 5.20 shows the results that we obtained using the third dataset. Slow actuators introduce inaccuracies in the monitoring signals that degrade the performances of the models, and this impacts also the adaptability of the models that show a more oscillating behavior than before. However, in general the models that we adapt improve over time also with slow actuators, even if the MAE computed over the Future validation set show that there are less improvements over the no-learn configuration. Partially, this can be justified also considering that the duration of the experiment is shorter (almost half) than the duration of the previous experiments.

In summary, with these results we can conclude that for Kriging models improve over time while remaining manageable, with respect to training time and size of training set, for an on-line use. Furthermore, limited to the presented results, we can see that both the `Raw` and the `Aggregated` configurations improve over the `No Learn` configuration. In particular, the `Raw` configuration appears to be better than the aggregated one.

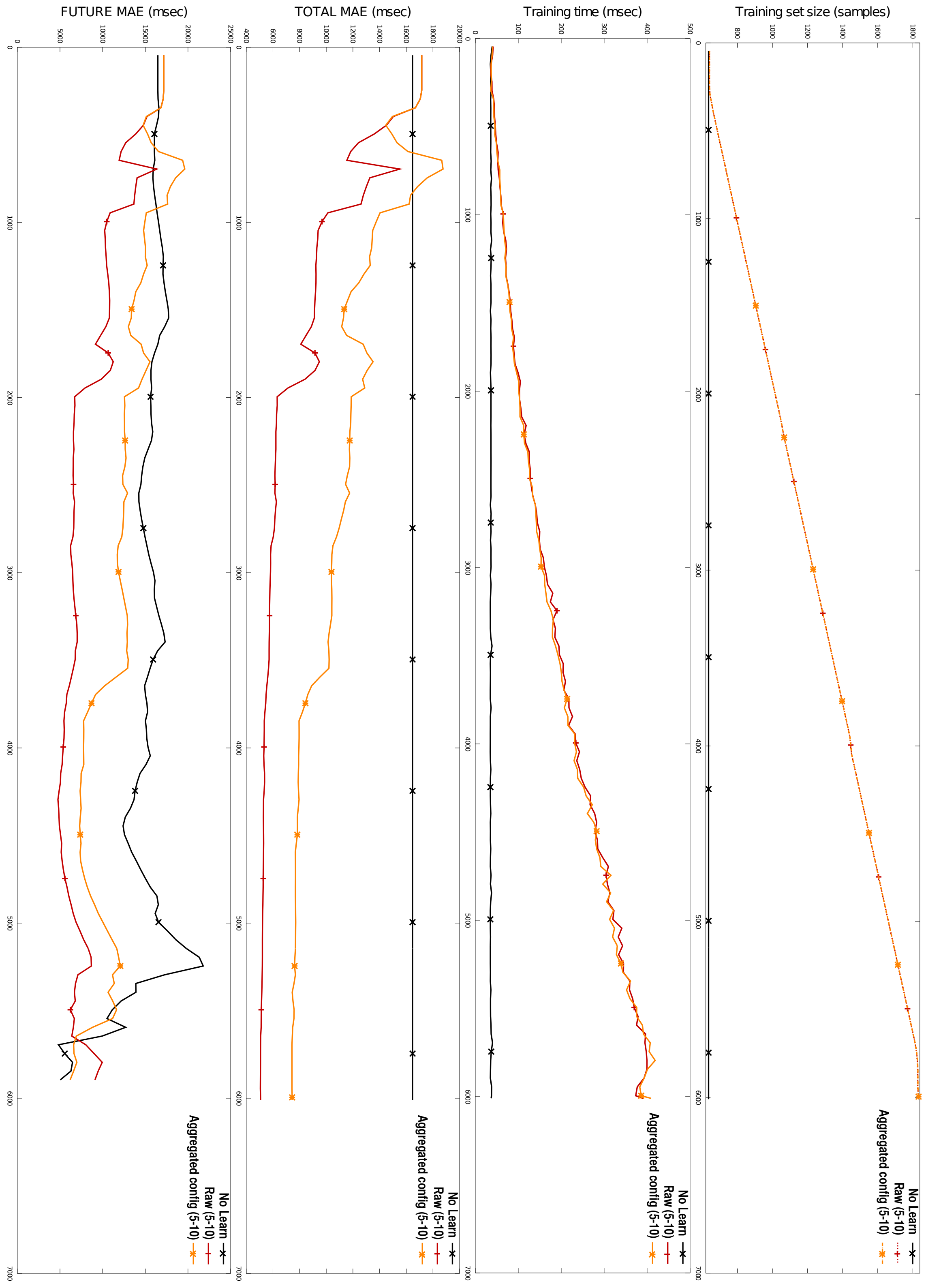

Figure 5.20. Adaptation of Kriging models: Periodic workload and slow actuators.

Comparison with the other models

We compare the results obtained with Kriging models against the ones obtained with regression trees, splines and combined models. We repeat the experiments using the three datasets presented before, and we use the same configurations for all the models: no adaptation (`No Learn`), adapt using monitoring samples (`Raw`), and average the system configurations over the monitoring sample (`Aggregated config`). For brevity, in the following we show only the configurations that retain at most five samples for each input tuple and that retrain the model once every ten monitoring cycles. For the sake of comparison, we report also the results obtained with Kriging models .

We must note that all the models start with the same initial training set, and they will accumulate the same input data over the experiment. For this reason, we do not show the evolution of the size of the training set because it is the same as the previous evaluation.

We present first the results about Kriging models adaptability in each dataset, and then we summarize the conclusions

Figure 5.21 shows the results for the first dataset. The first plot reports the evolution of the training time for the different models, while the other two show the evolution of the mean absolute error computed over both the total and future validation sets. We do not report the evolution of the root mean square error because it follows a similar evolution of the mean absolute error.

We observe that regression splines have an oscillating training time that could prevent their effective adaptation on-line. In fact, the training time ranges between fifteen to one hundred seconds and it has no clear correlation with the amount of data used to train the model. In any configuration, splines have the biggest training time of all the considered models. Regression trees come after with a training time that grows up to almost ten seconds in the largest configuration. Their adaptation on line may be critical only for applications that demand fast retraining cycles. Finally, we see Combined model and Kriging models that are the fastest. These models have the same behavior because the combined model does not retrain its analytical model, and the only retraining effort is due the Kriging model.

By looking at the Total MAE we can make the following observations: Without any retraining combined models show the smallest error, then come regression trees and Kriging and finally spline, as we showed in Section 5.3. If we consider the adapted configurations of the models, we see that there is a general improvement followed by a stable situation. This is similar of what we observed with Kriging and show that all the models can learn the system behavior if enough data (and time) are provided. In this example, we see that combined models outperform the other models, followed by regression trees, and finally by Kriging and splines. The differences between the behavior of each model appear during the transitory period, i.e., while they are learning: Kriging and combined models show a smooth and, more or less, constant improvement as data are added, and the error in both configurations is always below their baseline

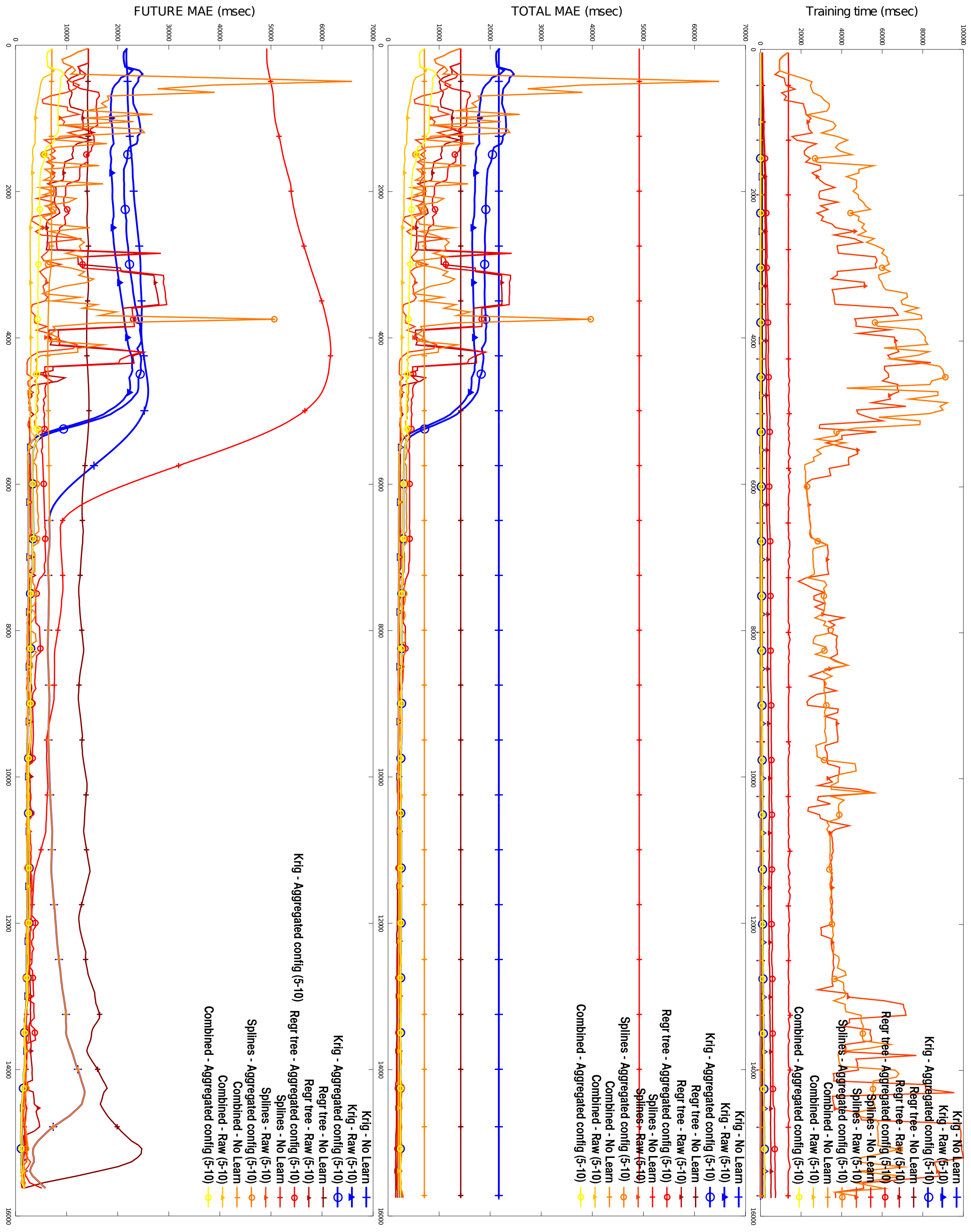

Figure 5.21. Comparing models improvement: Heterogenous workload and fast actuators.

case. Regression trees and spline show a different behavior that oscillates and sometimes that takes the error above their baseline cases. This can be critical because these models do not always improve while we add training samples, and a controller cannot really estimate (with no effort) the quality of the models while the system is running. In a sense, we cannot say for sure that the model is learning the behavior of the system by looking just at the error metrics obtained in the past. Furthermore, we have no direct mean to say whenever the data we have collected are enough and we can safely stop collecting data, as we can do with Kriging models.

By looking at the Future MAE we can make several observations. With no retraining, all the models but the regression trees show the characteristic behavior that we saw with Kriging models, that is, they grow, then drop and finally stabilize. Regression trees instead show an almost constant error over the experiment. Considering the relative nature of the Future data set this means that they make a very small error in predicting the behavior of the system under workloads `W1` and `W2` while they make bigger errors in workload `W3`. We explain this by considering the ability of regression trees to extrapolate data: During the first and second workloads, the model is able to capture the general trend system behavior and the model errors are compensated by the extreme conditions of the system. During the last workloads, where the working conditions are stable, the prediction errors cannot be compensated anymore; and this is a consequence of their inability to interpolate the data. If we consider models that adapt we see again that in general there is an improvement over time with strong oscillatory behaviors for splines and trees. Also according to this metric, combined models are the best options, followed by Kriging models, regression trees and eventually splines.

Figure 5.22 shows the results for the second dataset. As in the previous run, we see that the training time for regression splines is much higher that the others (circa 150 sec), and it strongly oscillates along the run. For the other models instead, the training time remains under ten seconds also in the worst case, with few or no oscillations.

Regarding the Total MAE, we see that adapted regression trees have the biggest error until the fully learn the system behavior (circa at sample number 2500). After that point, they outperform the other models, that in order are: Kriging and combined models first, and regression splines then. Moreover, we can see a constant improvement of the model over time. Compared to regression trees, the other models start with a smaller error at the beginning, but they follow a similar evolution, with a drop in the error value followed by a relatively stable behavior. It must be noted that in this example Kriging and combined models behave the same. This means that for the working conditions experienced by the system the Kriging model was already accurate enough, avoiding the use of the analytic model at all. The plot about Future MAE shows a similar behavior.

Figure 5.23 shows the results for the third dataset. With slow actuators, as we expected, we see a general degradation of the accuracy measures for all the models. Kriging and combined models perform comparably good to regression trees, while

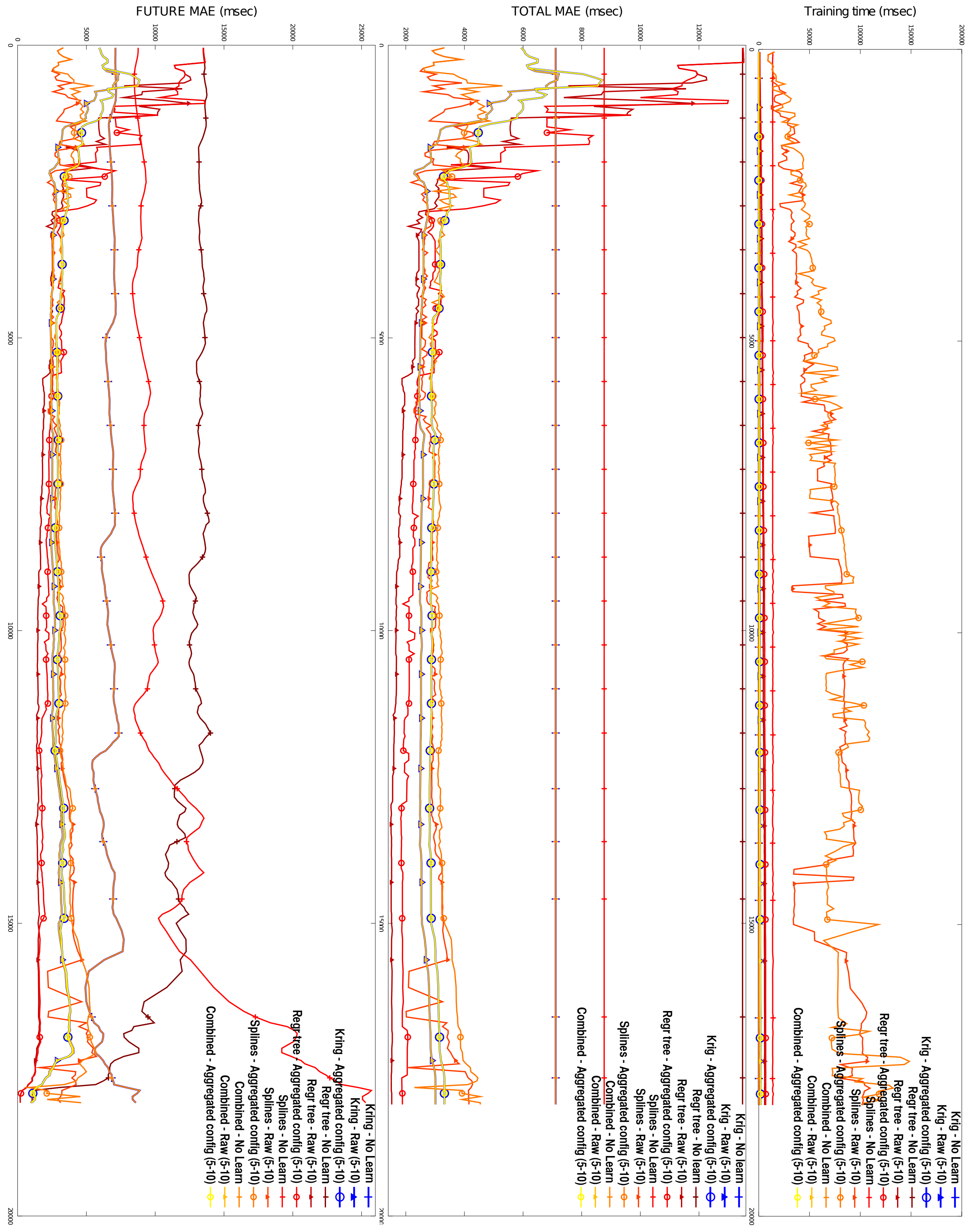

Figure 5.22. Comparing models improvement: Periodic workload and fast actuators.

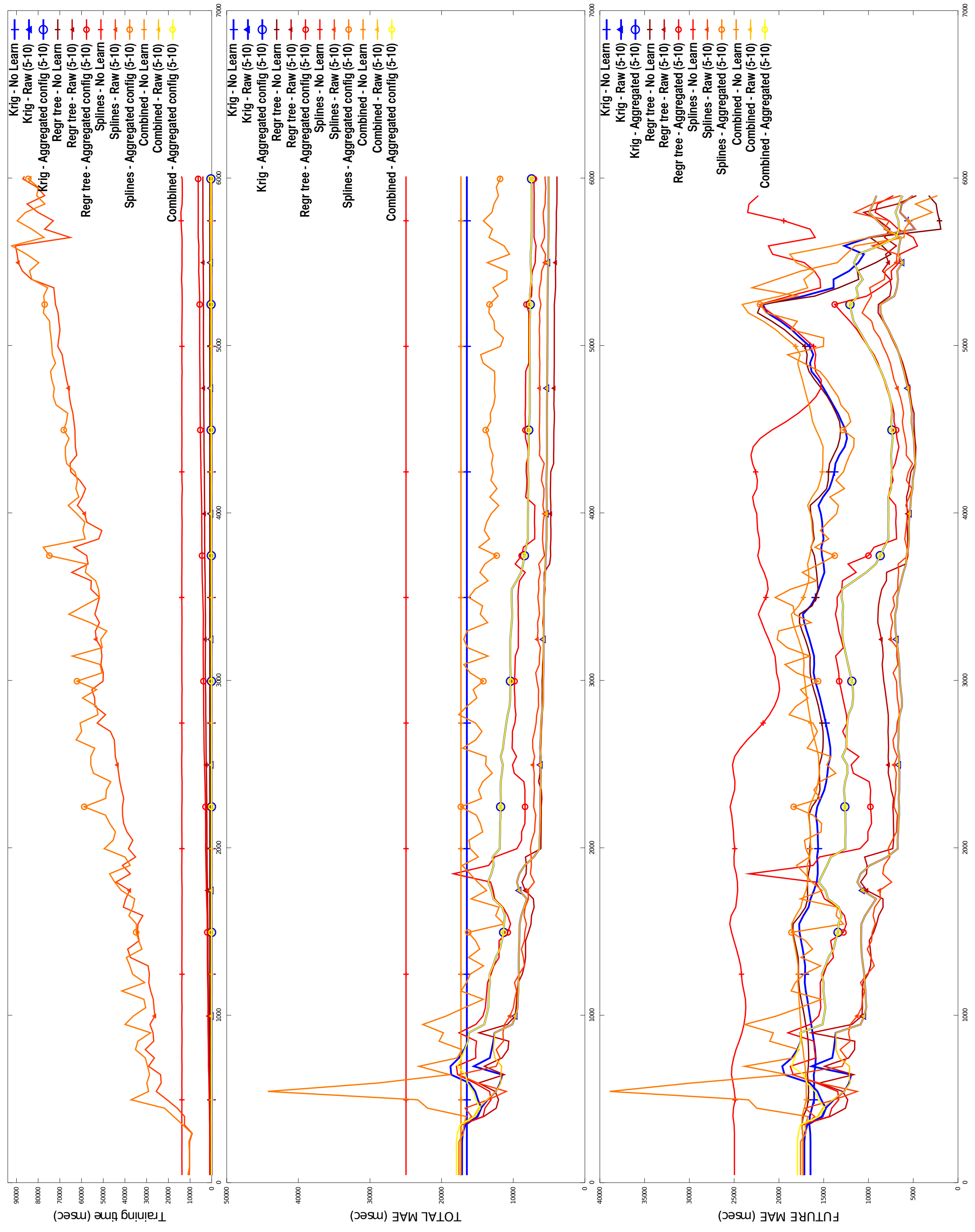

Figure 5.23. Comparing models improvement: Periodic workload and slow actuators.

splines generally perform worse than them.

In summary, these experiments confirm our previous intuitions: Regarding the accuracy measures, all the models behave more or less similarly along the run, that is, they improve as more data are collected and stabilize after a while. However, we see differences in the timing behavior and the evolution of the error along the experiment.

Multivariate adaptive regression splines do not scale well as more data are collected, while regression trees and combined models can be retrained within few seconds even with the largest training set. In particular, if the data collected are too "close" to each other in a spatial sense, splines models show the worst timing. We can see this by comparing the training time for the first and second run. Monitoring data are more sparse in the first run than in the second, while training time is smaller in the first run than in the second.

Regarding the evolution of the error, we can see that in the first run the combined model outperforms the other models. This happens because the analytic part of the combined model works at its best when monitoring data are missing. After enough data are collected the other models recover. From the same run, we can see also that the regression tree has a counter intuitive behavior: Both for the future and for the total validation set, it starts with a decreasing error, as we expect, but suddenly it grows to drop again later. This might happen because the new monitoring data led the model to create additional rules that are not pruned during the model optimization, leaving the model in an over-fitted configuration. Later as more data are available, the training procedure can prune these rules and it recovers. The second and third runs show that for all the models, once they learn the system behavior, they are able to maintain it also in presence of less accurate data.

Tracking system behavior

We adopt the following procedure to investigate how Kriging models can adapt to emerging system behaviors: We simulate the behavior of real systems using the traces that we collected during system runs. For this experiment, however, we change the system service time while it is running. We mimic the knowledge management process by periodically retraining the Kriging models. After each retraining, we use the models to predict the system behavior and we compute the mean absolute error and root square mean error over the total and future validation sets. At the end of the run, we collect the error metrics and we study how they evolved over time. To have a global picture, along with the error metrics we monitor how the size of the training set and the corresponding training time evolve.

We repeat the experiment under the different configurations that we used in the previous evaluation step.

We use a single dataset to run the experiment that Figure 5.24 reports. We obtain this dataset with the Sun grid engine application deployed in the Reservoir Cloud infrastructure. To simulate the system evolution we changed the service time of the submitted jobs by injecting an artificial delay that alters the original average service time of the jobs.

We use a simple rule based controller to change the system configuration at run-time. This controller uses a scale-up and a scale-down thresholds on the length of the queue to change the configuration. From the figure, we can see that this causes the controller to use more machines after we change the system behavior, as queues become longer after the system change.

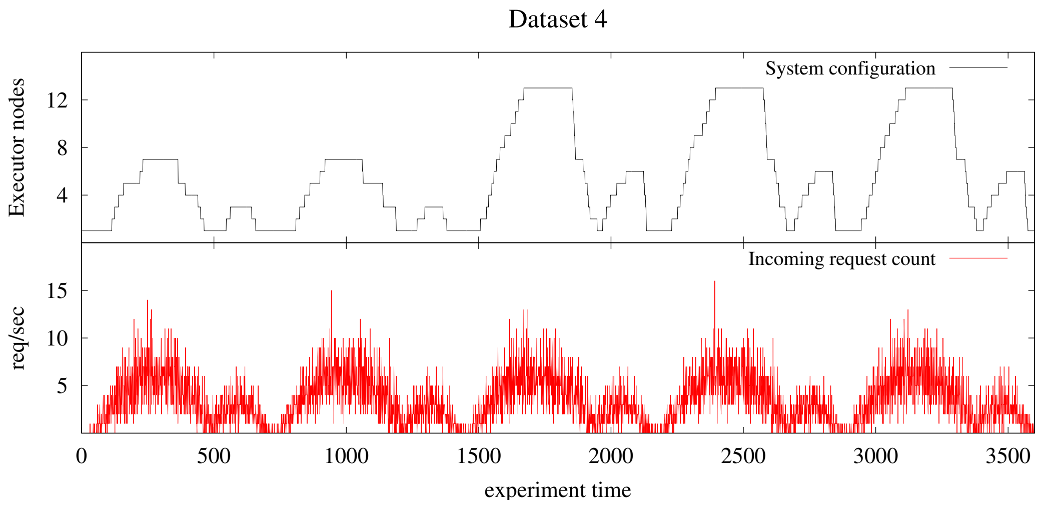

Figure 5.24. Dataset for evaluating Kriging models adaptation to emerging behaviors

For brevity, we report the only a subset of the results. In particular, we omit the runs with retain count of two samples and with retraining period of fifty monitoring cycles.

Figure 5.25 summarizes the results of the experiment. The first two plots report the evolution of the size of the training set and the corresponding training time, while the other two show the evolution of the mean absolute error computed over both the total and future validation sets. We do not report the evolution of the root mean square error because it follows a similar evolution of the mean absolute error.

Also in this experiment we see that same evolution of the models regarding the size of the training set and the training time. Again, these two figures of the model remain under acceptable conditions.

The Total MAE shows at the beginning the usual pattern: It increases for a short time, then it decreases and finally it stabilizes. When the experiment reaches the point where we change the system behavior we see again the same pattern.

The Future MAE evolves in a similar way. For the adapted models, it starts with a value, then it increases as we move close to the change point as effect of the relative importance of the new behavior against the old one, while we still use the old model. After we pass the change point, the error drops because the model learns the new

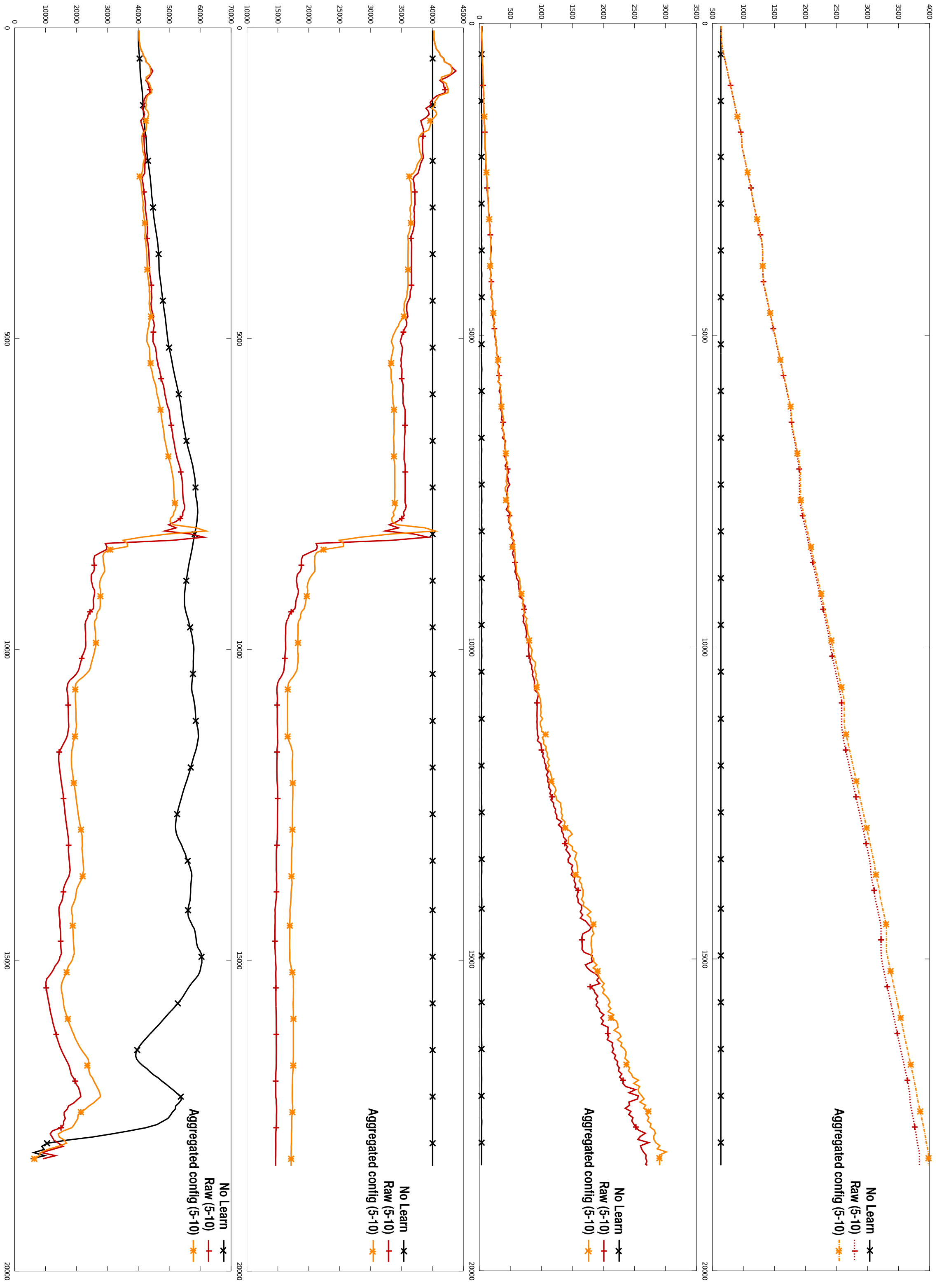

Figure 5.25. Adaptation of Kriging models: Periodic workload and fast actuators with emerging system behavior

behavior, but it oscillates. For the no-learn configuration instead, the error oscillates but remains high.

Compared to the previous experiments, i.e., without system evolution, we can see mainly a difference on the absolute value of the errors that are bigger now than before. This happens for mainly two reasons: First, we retrain the models by replacing old training samples with the new, but this does not remove all the old samples. In a sense, there is no hard switch to turn off the old model and start the new one. Second, we compute the error over the whole experiment, and as the model evolves, we predict both the old and new behaviors with that model: At the beginning we are good in predicting the first part, i.e., the original behavior, but not the second; at the end, is the opposite. This is especially visible in the Total MAE. No matter the absolute value of the errors, we see that it decreases, meaning that the model is improving.

Comparisons with the other models

To complete the evaluation we compare the results obtained with the Kriging models against the ones of the other models.

We must note that all the models start with the same initial training set, and they will accumulate the same input data over the experiment. For this reason, we do not show the evolution of the size of training set because it is the same as the previous evaluation.

Training times show the same results as before: Kriging models, combined model and regression trees scale well with the size of training set, while multivariate adaptive regression splines does not. In this particular run, in the worst case the training time of regression splines reached two hundred and fifty seconds confirming that they are unsuitable for on-line learning.

We see that all the models over the future and total validation set have similar behavior. In particular, regression trees show the smallest errors in the experiment. Regarding the Total MAE, the combined model perform comparably to regression trees in the first part of the run, while they follow behind in the second part. Splines show a similar behavior. This result highlights on one side, the similarities of splines and Kriging models that show bigger error after a change in the system behavior, and on the other side, the differences with regression trees that can learn a bit faster than the others.

Summary of the results

From this part of the evaluation we can draw the following conclusions:

i) Kriging models are robust: They maintain an acceptable accuracy also when we use them to predict real monitored signals that may be noisy. This is true also if we

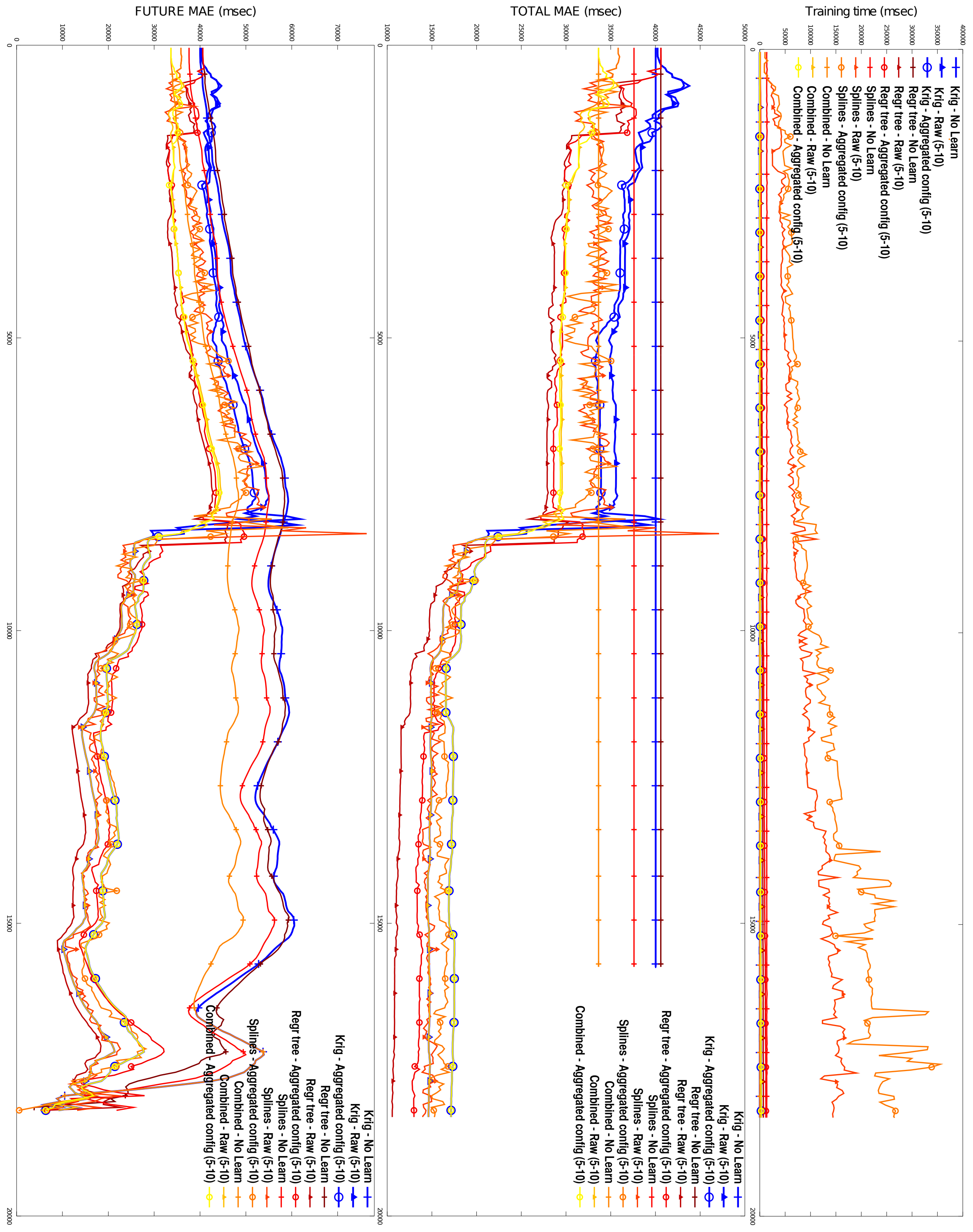

Figure 5.26. Adaptation of other models with emerging system behavior

use slow actuators with variable actuation times that introduce further inaccuracy in the data.

ii) Kriging models can leverage `sig` and seamlessly switch to other models when predictions are supposed to be not accurate. Combined models are a working implementation of this method, and we show that they usually outperform the other models.

iii) Kriging are timely: They can be queried in milliseconds and retrained in less than a couple of seconds, therefore they can be used and adapted on-line.

iv) Kriging models, when retrained on-line, improve their accuracy if the underlying system behavior does not change, and adapt to it otherwise. In case of heterogeneous workloads, they show a continuous improvement without oscillations.

v) Compared to the other models, Kriging have comparable accuracy, but demand different effort for making the predictions and during the training. Furthermore, their adaptation behavior is different from the one of the other models. In particular, Kriging models scale better than both regression trees and multivariate adaptive regression splines, while are comparable to the combined model.

In conclusion, Kriging models and combined model that derive form them are suitable to be used as core components of self-adaptive controllers. This is especially true in case the controllers need frequent adaptations, fast reactions and informed decision about the quality of model predictions.

It must be noted that the validity of these results may be threatened by the fact that we performed a deep investigation only by considering the Sun grid engine case study, while we performed only limited investigations with other case studies.

5.4 Evaluation of Kriging-based Self-adaptive Controllers

With the previous validation efforts we show that Kriging models can be employed as core components of self-adaptive model-based controllers for elastic applications running in the Cloud. That is, they fulfill the requirements about accuracy, adaptation and timing for on-line usage.

In this section, we show the performance of controllers based on Kriging models, and finally address the main research hypothesis of this thesis:

> **Research Hypothesis**: *Kriging models can be used as the core element of efficient and effective autonomic controllers.*

We adopt the following procedure to evaluate Kriging based controllers: We deploy a case study application on the Cloud infrastructure, we run our prototype Kriging based controllers to control it, and we generate a synthetic workload. The controller monitors the application KPIs, plans the control actions, and implements them on the platform. During the experiment we monitor application performances to check for

SLA violations and we monitor the Cloud infrastructure to measure the amount of resources allocated to the application.

To evaluate the generality of the controllers we repeat the same experiment under different settings. In particular we change the controller settings, the Cloud infrastructure settings, and the synthetic workloads. Finally, to contextualize the performances of Kriging based controllers, we compare them against other controllers inspired by the state-of-the-art of Cloud computing. We force pure Kriging based controllers to ignore their `sig` values, thus we reduced Kriging models to common black-box models. In this way, we can implement a fair comparison. On the other hand, we keep the `sig` value inside combined models as we have no alternative means to implement model switching.

To compare controllers we combine three metrics that measure their performances:

$\overline{VM}$ The average consumption of resources measures how many running virtual machines are used, in average, during the run of an experiment. $\overline{VM}$ therefore accounts for the cost of running system under a given controller. Smaller values of $\overline{VM}$ indicates more efficient controllers.

$\mathbf{p}_{wdp}$ The percentage of recorded violations measures how many violations are monitored during the run with respect to all the monitored values. Its value is computed according to the following formula:

$$\mathrm{p}_{wdp} = \frac{1}{n}[n - \sum_{i=1}^{n}(\mathbb{I}_{SLA})] \tag{5.5}$$

where $\mathbb{I}_{SLA}$ is the Borel set defined by the SLA, i.e., a function that equals to 1 if the value of the output (y_i) is within the threshold defined by the SLA (y_{SLA}) and to 0 if the SLA is violated. This metric measure the efficacy of controllers in an absolute sense, and ideally its value should be equal to 0.

$\mathbf{e}_{wdp}$ Average intensity of violations measures the average distance of the output signal (y_i) that exceeds the threshold defined by the SLA (y_{SLA}). Its values is computed according to the following formula:

$$\mathrm{e}_{wdp} = \frac{1}{n}[\sum_{i=1}^{n}\mathbb{I}_{\neg SLA}|y_i - y_{SLA}|] \tag{5.6}$$

where $\mathbb{I}_{\neg SLA}$ equals to 1 if y_i violates the SLA and 0 otherwise. This metric measure how (in)effective is a controller, and its ideal value should be equal to 0.

We borrow p_{wdp} and e_{wdp} from standard control theory where they are used to measure how well controllers track reference signals (see for example the work by Maggio and coauthors [66]), and we adapt their formulation to our specific case.

We combine these three metrics in an aggregate cost metric named *Total Cost*, identified as TC, and computed according to the following formula:

$$\mathrm{TC} = c_{\overline{VM}} \cdot \overline{VM} + c_{\neg SLA} \cdot (\mathrm{p}_{wdp} \cdot \mathrm{e}_{wdp}) \tag{5.7}$$

where the first part of the equation accounts for the costs due to the resources usage by means of the parameter $c_{\overline{VM}}$, and the second part accounts for costs that are due to SLA violations by means of the parameter $c_{\neg SLA}$. In our evaluation, we investigate how the total cost of run varies as we change $c_{\overline{VM}}$ and $c_{\neg SLA}$. Once we fix these the costs parameters, TC enables us to compare directly the performances of controllers: Controllers with smaller values of the metric are better.

In our evaluation, we compare Kriging based controllers against the following solutions:

i) Statically configured systems. These no-op controllers form the baseline for the comparison. For each experiment, we empirically find two boundary configurations: The smallest configuration that leads to no SLA violations and the largest configuration that causes at least one violation of the SLA.
ii) Rule-based controllers. These controllers are commonly found in the domain of Cloud computing. For example, in the experiment with the Sun grid engine, we configure the rule based controller according to the guidelines provided by our industrial partners (Sun at that time) in the context of the Reservoir EU Project [93; 102].
iii) Alternative model-based controllers. We design our prototype adopting a plug-in style and we leverage this to implement several controllers just by plugging-in different models. This will leave the original control logic intact and leads us directly compare the effects of different modeling options. In the evaluation with the SGE we run model based controllers with: Analytical models, regression tree and combined models.

For each controller, when applicable, we evaluate both the reactive and the proactive control strategy, as well as, the different adaptation strategies introduced in the previous section, i.e., `Raw` and `Aggregated config`. We also must notice that during the experiment we set the retain parameter of all the models equals to 2. In this way, we do not overwhelm these models that do not scale well in the size of the training sample. This however comes at the cost of reducing the overall quality of the models.

We run controllers to manage the Sun grid engine application under a variegated set of working conditions:

Exp1 Slow varying workloads and fast actuators. We stress the application with a slowly varying workload and we let the platform implement fast changes in the system configuration. This run is the closest one to the idealized situation that we can implement. In this case, the load is not extreme and the inaccuracy introduced by the actuators is the smallest possible in our testbed.

Exp2 Fast varying workloads and fast actuators. We stress the application with a more intense workload that we generated by compressing the previous workload in half of the time. The application now is stressed with faster variations of the workload. This configuration may be commonly found in private or hybrid Cloud infrastructures that runs applications with a changing number of clients and hot replicas of virtual machines. Lately, similar configurations can be found also in public Clouds. For example, Amazon offers Elastic Block Stores[6] that maintain virtual machine in a stop state without releasing their physical resources, thus avoiding the time spent in deploying the virtual machines.

Exp3 Fast varying workloads and slow actuators. In this experiment, we maintain the previous workload, but we use slow actuators. This case is more close to public Clouds, such as Amazon, where virtual machines are cloned, deployed and restarted from scratch every time. These settings are critical from a controller designers point of view, because the actuation time may change between different deployments, making the planning activities even more challenging.

The intent of this process is to show how controllers' performance degrade from idealized settings to real ones.

We present results case by case, and we conclude with a summary of all the runs. For each experiment, we discuss results obtained with Kriging models and we compare them against the ones obtained with other controllers.

Table 5.7 reports the results from the first experiment and is organized as follows: The first four columns identify the controllers according to their configurable parameters, i.e., the model used for planning and analysis if any, the style of control that is either reactive or proactive, the model adaptation if implemented, and the aggregation of monitoring data about the system configuration if implemented. The next three columns report the monitored metrics about resources usage and SLA violations. The last three columns report the aggregate cost metric for different value assignments of the cost weights: $^1/_2$ identifies the case where the average cost of violating the SLA is half of the cost of running a VM, $=$ identifies the case where the values of the two are equal, while 2 identifies the case where SLA violations cost is double the VM cost.

We see from these results that all the model based controllers but the one based on the analytic model causes SLA violations when they adopt the reactive style. This happens because reactive controllers assume perfect actuators that result in immediate effects on the system, but this assumption is not met even using fast actuators. In this sense, fast actuators are not fast enough to implement effective reactive control. Model (in)accuracy makes this situation worse by increasing the values taken by p_{wdp} and e_{wdp}. We can see this for example by comparing the performance of controllers based on combined models against the one that use Kriging and regression trees.

[6]`http://aws.amazon.com/ebs/`

Table 5.7. Performances of controllers in the first experiment

Controller				**Experiment**			**Total Cost**		
model	*style*	*adapt*	*aggr*	$\overline{VM}$	p_{wdp}	e_{wdp}	$^1/_2$	$=$	2
no	fixed	no	no	5.00	0.26	14.68	6.91	8.82	12.63
no	fixed	no	no	6.00	0.00	0.00	6.00	6.00	6.00
no	rules	no	no	4.36	0.00	0.00	4.36	4.36	4.36
analytic	reactive	no	no	3.25	0.00	0.00	**3.25**	**3.25**	**3.25**
analytic	proactive	no	no	3.60	0.00	0.00	3.60	3.60	3.60
combined	reactive	no	no	3.07	0.33	2.23	3.44	3.81	4.54
combined	reactive	yes	no	3.19	0.34	7.87	4.53	5.87	8.54
combined	reactive	yes	yes	3.13	0.57	14.10	7.15	11.17	19.20
combined	proactive	no	no	3.69	0.00	0.00	3.69	3.69	3.69
combined	proactive	yes	no	4.47	0.00	0.00	4.47	4.47	4.47
combined	proactive	yes	yes	3.38	0.00	0.00	3.38	3.38	3.38
kriging	reactive	no	no	4.27	0.69	26.40	13.38	22.49	40.70
kriging	reactive	yes	no	3.17	0.29	7.57	4.27	5.37	7.56
kriging	reactive	yes	yes	3.12	0.60	54.75	19.55	35.97	68.82
kriging	proactive	no	no	3.67	0.00	0.00	3.67	3.67	3.67
kriging	proactive	yes	no	3.83	0.00	0.00	3.83	3.83	3.83
kriging	proactive	yes	yes	5.05	0.00	0.00	5.05	5.05	5.05
m5rt	reactive	no	no	3.18	0.15	3.30	3.43	3.68	4.17
m5rt	reactive	yes	no	3.15	0.27	10.50	4.57	5.98	8.82
m5rt	reactive	yes	yes	3.13	0.54	45.75	15.48	27.84	52.54
m5rt	proactive	no	no	4.56	0.00	0.00	4.56	4.56	4.56
m5rt	proactive	yes	no	3.49	0.00	0.00	3.49	3.49	3.49
m5rt	proactive	yes	yes	3.93	0.00	0.00	3.93	3.93	3.93

The reactive controller based on the analytic model instead does not cause any violation because it uses an over conservative model that returns pessimistic predictions about the system response time. In fact, with a conservative model the controller is more sensible to variations that may lead to violations of SLA and reacts sooner than the other controllers, reducing the negative effects of the actuators.

Proactive controllers instead account for this “non-ideality" of the actuators, therefore they are able to avoid more violations of SLA. Also the rule base controller avoids SLA violations.

We notice small differences in the average usage of resources: Almost all the controllers used less than five virtual machines in average, with the majority of them oscillating between three and four. If we compare this result against statically configured systems we can see that the smallest fixed configuration that for this workload is able to avoid violation is also the largest one with a size of six virtual machines. This gives an intuition on the advantages of using dynamically adjusted systems over static ones.

If we compare controllers along their adaptation strategies we notice that adapting the model to the system behavior does not always result in an improvement of the controllers performance. This could be caused by the fact that we retain only two samples for configuration and also because these controllers implements generally more control actions that the one we used to collect the validation traces in the previous experimental validation. In particular, we notice that Kriging (and combined) models are sensitive to inaccuracies introduced by averaging the system configurations over the training period. For reference, we plot in Figure 5.27 the result of some of the runs that are reported in the table.

Table 5.8 reports the results from the second experiment.

Table 5.8. Performance of controllers in the second experiment

Controller				**Experiment**			**Total Cost**		
model	*style*	*adapt*	*aggr*	$\overline{VM}$	p_{wdp}	e_{wdp}	$^1/_2$	=	2
no	fixed	no	no	10.00	0.23	8.55	10.98	11.97	13.93
no	fixed	no	no	11.00	0.00	0.00	11.00	11.00	11.00
no	rules	no	no	7.24	0.00	0.00	7.24	7.24	7.24
analytic	reactive	no	no	6.71	0.02	0.05	**6.71**	**6.71**	**6.71**
analytic	proactive	no	no	7.67	0.00	0.00	7.67	7.67	7.67
combined	reactive	no	no	7.02	0.28	6.25	7.89	8.77	10.52
combined	reactive	yes	no	6.40	0.40	15.64	9.53	12.66	18.91
combined	reactive	yes	yes	6.62	0.53	24.64	13.15	19.68	32.74
combined	proactive	no	no	8.16	0.00	0.00	8.16	8.16	8.16
combined	proactive	yes	no	7.87	0.00	0.00	7.87	7.87	7.87
combined	proactive	yes	yes	7.93	0.00	0.00	7.93	7.93	7.93
kriging	reactive	no	no	6.82	0.28	6.13	7.68	8.54	10.25
kriging	reactive	yes	no	7.27	0.51	39.19	17.26	27.26	47.24
kriging	reactive	yes	yes	6.84	0.59	30.19	15.75	24.65	42.46
kriging	proactive	no	no	8.14	0.11	3.06	8.31	8.48	8.81
kriging	proactive	yes	no	7.53	0.00	0.00	7.53	7.53	7.53
kriging	proactive	yes	yes	7.68	0.00	0.00	7.68	7.68	7.68
m5rt	reactive	no	no	6.52	0.39	12.95	9.05	11.57	16.62
m5rt	reactive	yes	no	6.79	0.39	11.17	8.97	11.15	15.50
m5rt	reactive	yes	yes	6.82	0.63	50.30	22.66	38.51	70.20
m5rt	proactive	no	no	6.80	0.26	9.20	8.00	9.19	11.58
m5rt	proactive	yes	no	6.76	0.24	9.28	7.87	8.99	11.21
m5rt	proactive	yes	yes	7.63	0.03	0.10	7.63	7.63	7.64

In this experiment, we stressed the system with a workload that is more challenging than the one we used before, and we notice lower performances for all the controllers. As a consequence, their total cost increased. Looking at resource consumption, we see

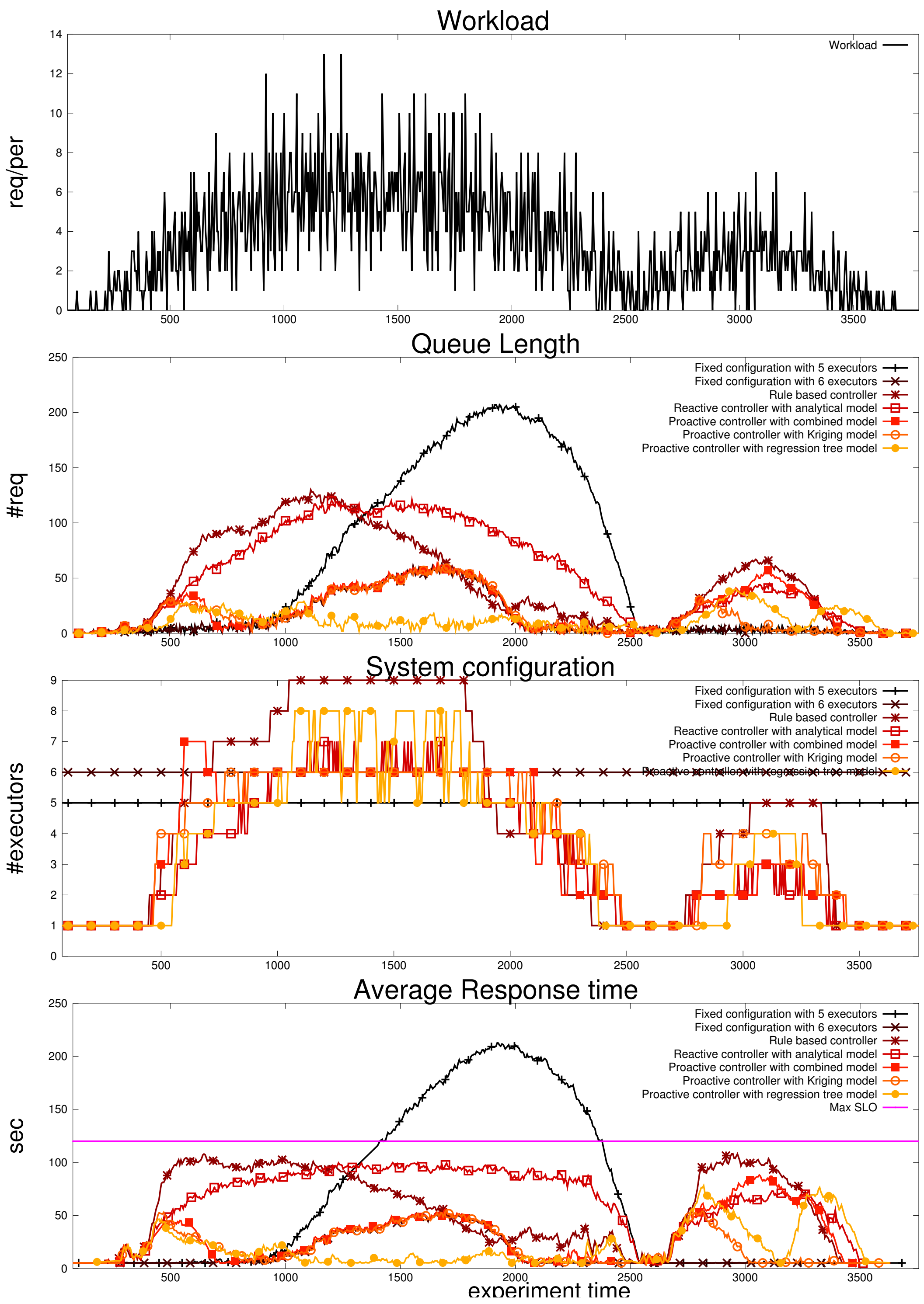

Figure 5.27. Best models for Experiment 1

that controllers almost double the virtual machines used during the run, but compared to the minimal system configuration that avoid violations, autonomic controllers result in stronger resource savings.

Looking at the violations metrics, we see again the general trends that we discovered in the previous run: Reactive controllers perform worse that proactive, and learning the system behavior from data is not always beneficial to the quality of controllers. We observe that also the reactive controller based on the analytical model fails to avoid SLA violations, thus its pessimistic attitude is not always able to trigger timely system reconfigurations anymore.

We also observe that several proactive controllers causes SLA violations: For example, all the controllers based on regression trees causes violations, with a difference between controllers that adapt using aggregated information and the others. And we notice violations also for the proactive controller based on Kriging model that does not implement adaptation. The rule based controller instead is able to sustain the workload with no violations, as does the proactive controllers based on combined models. For reference, we plot in Figure 5.28 the result of a subset of the runs that are reported in the table.

Table 5.9 reports the results from the third experiment. This experiment is more challenging from a controllers' point of view because we use slow actuators to change the system configurations, while we maintained the fast varying workload. This enables us to direct compare the results between these last two experiments.

We notice that some controllers use more resources than before, partially reducing the advantages with respect to statically configured systems. We see again how proactive controllers outperform their reactive counterparts.

Compared against the previous experiment, we see a general degradation of the performances for all the controllers except rare cases. In particular, collected data show that controllers based on regression trees have a small improvement; nevertheless they show consistent violations of the SLA. Noticeably, also the rule based controller and the proactive controller based on the combined model fail to completely avoid violations. In detail, the model based controller performs slightly better than the one based on rules, but it is also self-adaptable, easier to configure, and scales well to complex systems. For reference, we plot in Figure 5.29 the result of a subset of the runs that are reported in the table.

Summary of the results

From this last evaluation we can draw the following conclusions:

i) Reactive style of control, unless coupled with conservative models, is not so efficient nor effective. This results from the assumption of a perfect system being violated in reality, even if case of fast actuators.

ii) Proactive style of control, which does not rely on the perfect system assumption, shows better performance and effectively avoids SLA violations or reduces their

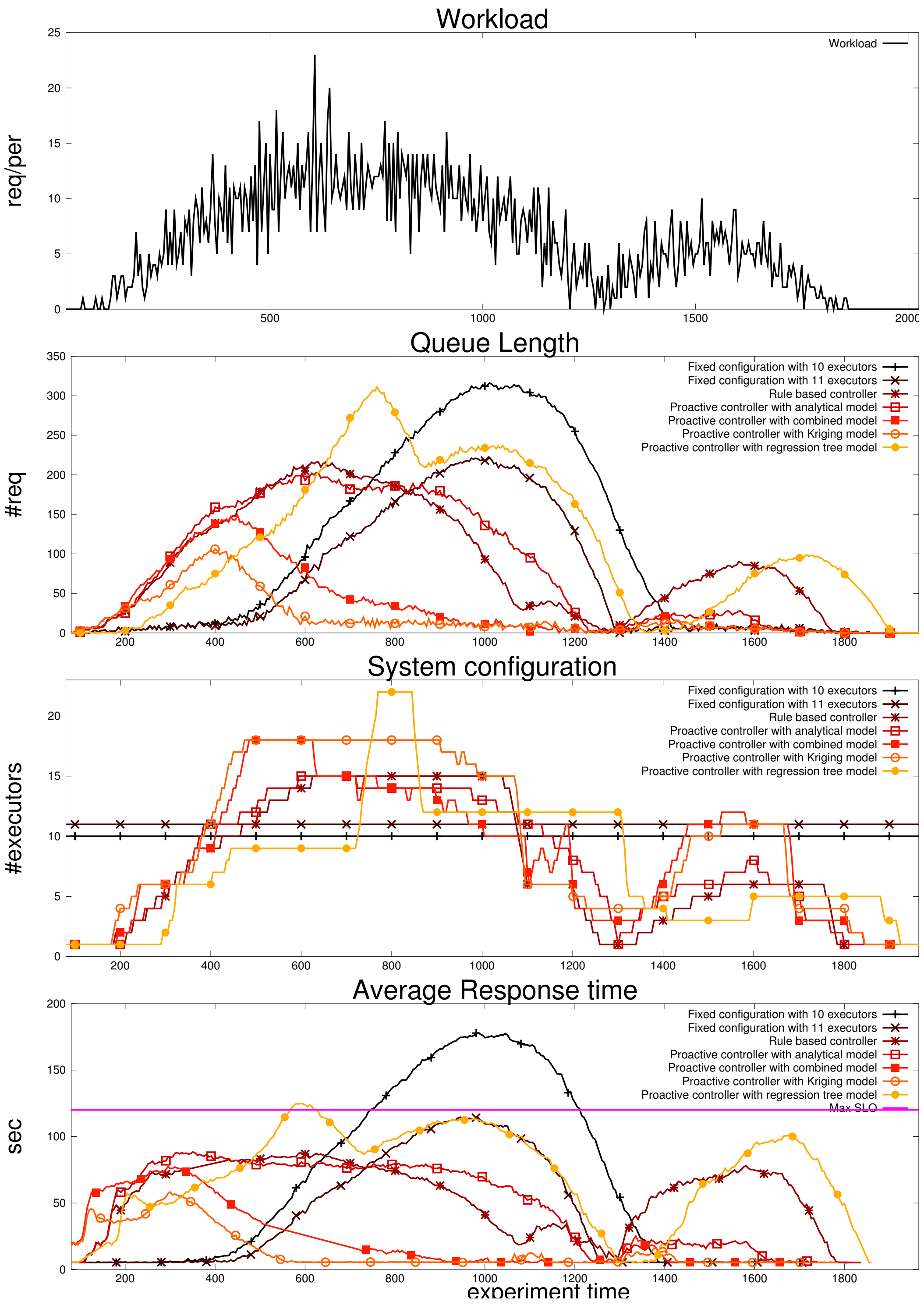

Figure 5.28. Best models for Experiment 2

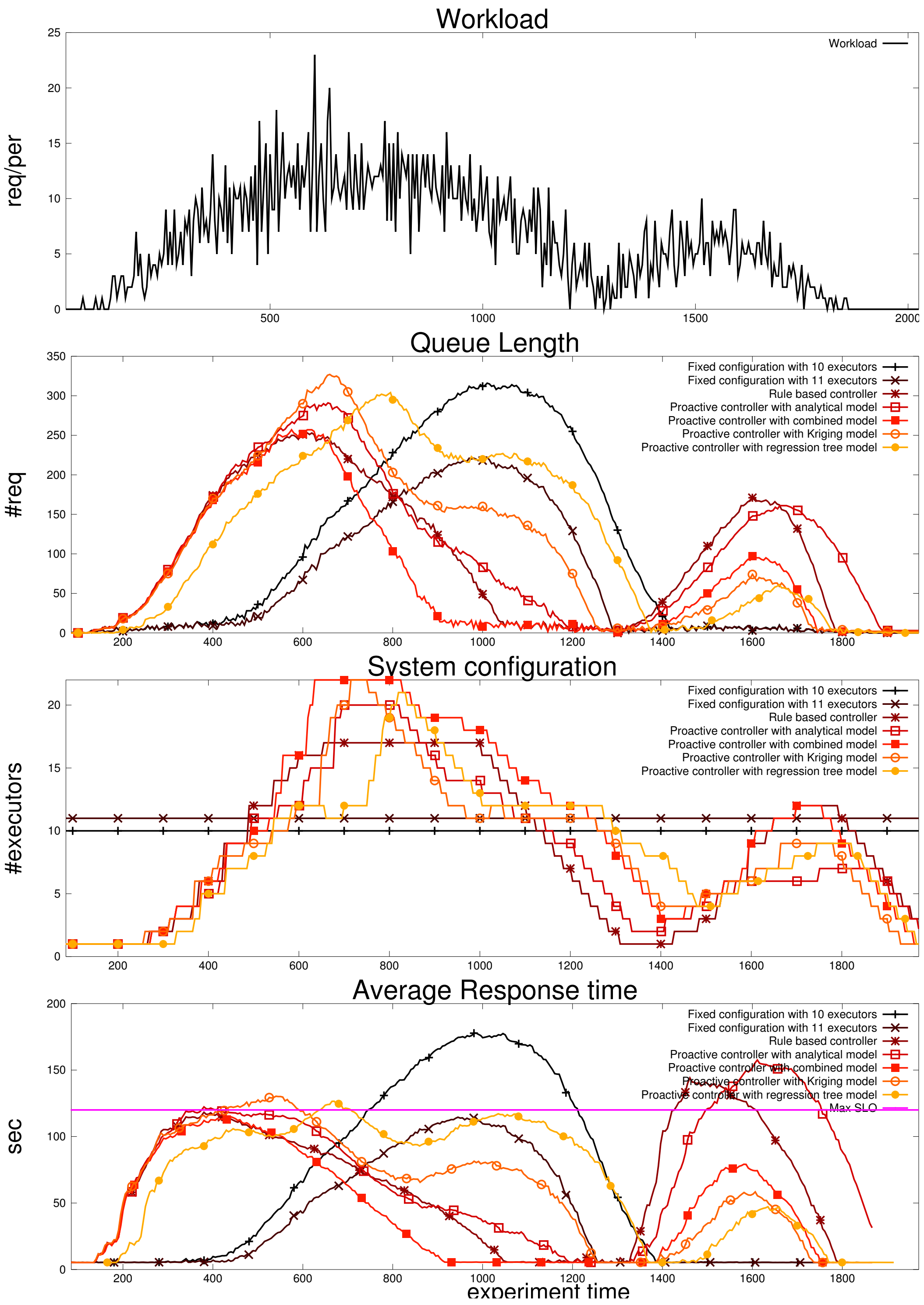

Figure 5.29. Best models for Experiment 3

Table 5.9. Performance of controllers in the third experiment

Controller				**Experiment**			**Total Cost**		
model	*style*	*adapt*	*aggr*	$\overline{VM}$	p_{wdp}	e_{wdp}	$^1/_2$	$=$	2
no	fixed	no	no	10.00	0.23	8.55	10.98	11.97	13.93
no	fixed	no	no	11.00	0.00	0.00	11.00	11.00	11.00
no	rules	no	no	8.17	0.09	1.16	8.22	8.27	8.38
analytic	reactive	no	no	6.76	0.26	3.52	**7.22**	**7.68**	8.59
analytic	proactive	no	no	7.71	0.15	2.79	7.92	8.13	8.55
combined	reactive	no	no	8.48	0.12	2.07	8.60	8.73	8.98
combined	reactive	yes	no	7.35	0.50	5.58	8.74	10.14	12.93
combined	reactive	yes	yes	8.81	0.08	0.65	8.84	8.86	8.91
combined	proactive	no	no	8.56	0.12	2.09	8.69	8.81	9.06
combined	proactive	yes	no	9.63	0.00	0.00	9.63	9.63	9.63
combined	proactive	yes	yes	7.96	0.02	0.02	7.96	7.96	**7.96**
kriging	reactive	no	no	7.60	0.53	22.36	13.53	19.45	31.30
kriging	reactive	yes	no	8.09	0.44	32.99	15.35	22.61	37.12
kriging	reactive	yes	yes	8.22	0.45	35.25	16.15	24.08	39.95
kriging	proactive	no	no	8.70	0.16	4.71	9.08	9.45	10.21
kriging	proactive	yes	no	8.84	0.12	1.03	8.90	8.96	9.09
kriging	proactive	yes	yes	7.99	0.08	0.45	8.01	8.03	8.06
m5rt	reactive	no	no	7.20	0.76	20.95	15.16	23.12	39.04
m5rt	reactive	yes	no	9.98	0.44	13.24	12.89	15.81	21.63
m5rt	reactive	yes	yes	13.49	0.48	34.44	21.76	30.02	46.55
m5rt	proactive	no	no	7.80	0.29	8.35	9.01	10.22	12.64
m5rt	proactive	yes	no	8.45	0.20	4.45	8.89	9.34	10.23
m5rt	proactive	yes	yes	8.22	0.04	0.11	8.22	8.22	8.23

occurrence to a minimum.

iii) In the explored configurations, controllers that adapts the models do not always improve over the configurations with no adaptation. Furthermore, their improvement depends on the type of model, the quantity of retained data, and their quality.

iv) If we do not resort to `sig` to evaluate the quality of predictions, Kriging based controllers perform comparably to the other controllers. However, if we do use `sig`, like we did inside the combined model, the quality of control increases notably.

In summary, we built self-adaptive model-based controllers that use Kriging models, and their derivates like the combine model, and we showed how they control a real application to avoid violations of the SLA under different workloads regimes and infrastructure settings. In particular, we showed that Kriging and combined models are suitable core components of self-adaptive controllers. We compared several controllers across different experiments and we showed the limitation of each design decision in increasingly realistic (and challenging) experimental settings. Compared to all the other controllers, we see that self-adaptive, proactive controllers based on combined models perform better.

Chapter 6

Conclusions

This thesis investigated the use of self-adaptive controllers based on surrogate models for dynamic resource provisioning. These controllers are suitable to manage on-line the allocation of resources to systems that have to maintain a specified level of service in the face of changing working conditions while minimizing their running costs. An important class of such systems are elastic applications running in Cloud infrastructures under the IaaS paradigm. In this context, controllers can add or remove virtual machines to scale the applications in the face of changing workloads to maintain applications behavior within the constraints specified by means of service level agreements.

Model based self-adaptive controllers are the alternative of commonly adopted approaches based on rules that may be difficult to set up, do not scale well with the system complexity, and may not be very effective in the presence of emerging behaviors or unexpected working conditions. We adopted black-box models, also known as surrogate models, as central element of the controllers, because black-box models are based on data collected from the running systems, do not require an extensive knowledge of system internals, and are more generally applicable than white-box models.

This thesis proposed to exploit Kriging models as core components of self-adaptive controllers. Kriging models belong to a particular family of surrogate models that were originally proposed in the domain of geo-statistics and that lately were adopted in the domains of computer-based simulation and machine learning. We chose Kriging models because they are as accurate as, easier to configure, and faster to train than other surrogate models studied so far in the same context. Furthermore, Kriging models scale well with respect to application complexity, and natively offer a measure of confidence that is associated to any prediction they make, thus enabling a more informed control strategy when used inside self-adaptive controllers.

We demonstrated the feasibility, efficiency and efficacy of self-adaptive controllers based on Kriging models in the context of Cloud computing. We implemented, adapted and ran different elastic applications and we collected a set of monitoring data that we used to initially train Kriging models. We showed the ability of Kriging models (i) to accurately capture the behavior of these elastic systems, i.e., their performance; (ii) to provide reasonably good predictions also with noise and inaccuracies in the data that may be introduced by real actuators and monitoring systems; (iii) to generate predictions in a timely manner; (iv) to be retrained within seconds also with big training sets; (v) to improve over time if the system behavior is stable, and to adapt to it otherwise. We compared Kriging models with alternative surrogate and analytic models, and we discussed relative advantages and limitations. To overcome them, we designed a model switching technique to combine Kriging and other models, in particular analytical models, and we obtained a combined model that outperforms all the other models.

We designed and implemented a generic controller for Cloud computing that leverages the plugin architectural style and can be extended to implement several kinds of controller ranging from traditional rule-based controllers to self-adaptive model-based controllers. We implemented plugins inspired by state of the art controllers, and we compared the effectiveness of the resulting instances by running them to control the allocation of resources of a case study application under different working conditions. We showed that proactive controllers based on Kriging models augmented with analytical models and that implement our model switching technique over perform the other considered controllers.

6.1 Contributions

The major contribution of this thesis is the identification of Kriging models as suitable core components for self-adaptive model-based controllers for dynamic resource allocation. Although Kriging models are well known in other domains this is the first time that they are used as core element of self-adaptive controllers; furthermore, we are the first to use Kriging models to capture the performances of real (and elastic) systems. The next major contributions are the model switching technique to combine Kriging and other models, and the proactive controller based on the accuracy measure (i.e., `sig`) provided by the Kriging models.

This thesis makes several other contributions that we summarize in the following:

Model comparison for Cloud computing To contextualize the performance on Kriging models that are accuracy, robustness, query and training time, we compared Kriging models with several surrogate models and analytical models. Our analysis of the results of such comparison identified common behaviors across surrogate models, and highlighted interesting differences that may help designers in the choice of the right modeling tool.

Generic controller architecture We designed and implemented a generic controller that can be used to implement advanced self-adaptive controllers both in the context of Cloud IaaS and outside of it. Different aspects of the implementation are easily configurable: For example, one can change (i) the adaptation strategy to retrain models at runtime, (ii) the model that the controller queries during the analysis of monitoring data and during the planning of control actions, and (iii) the basic strategy to chose the system target configuration.

Adaptation strategies We designed and implemented three adaptation strategies: `No Learn`, `Raw`, and `Aggregated config`. We empirically compared them across different workloads, controllers, and Cloud infrastructures. The results of this comparison may help designers in choosing the right option while developing their self-adaptive controllers.

Controllers We implemented several plugins to customize the generic controller architecture, and we obtained several controller instances. For example, we provided implementation of models such as Kriging models, combined models, regression tree models, multi adaptive regression spline models, analytical models, and multiple linear regression models. We also provided implementation of configurable rule based controllers (limited to the considered case studies), and controllers that implement statically configured systems.

Comparison of controllers for Cloud IaaS We compared the performance of the different controller instances under different working conditions. The results of this comparison may be beneficial for designers in choosing the right controller and in configure it during the design of self-adaptive controllers for the Cloud.

Benchmark applications To empirically validate our work, we modified legacy applications to became elastic, and we designed new elastic applications from scratch. These applications are able to automatically configure themselves, and come packaged in single, standard, virtual machines that can run on many the IaaS platform without requiring modifications. For example, we ran them on the Reservoir, Eucalyptus and Amazon Ec2 platforms. When released, these applications could be used as benchmarks to compare controllers for dynamic resource management in the Cloud.

Other components We developed a set of additional components, mainly infrastructural components, to perform our empirical validation. These components are generic and can be reused. In particular, we extended the Lattice monitoring framework originally developed by Clayman et al. [18], we provided a Web based GUI to live-monitor the status of the system, and we extended the open source Apache JMeter workload generator.

6.2 Limitations

We make assumptions on system behavior stability, smoothness, continuity and *homoscedasticity*[1] that may limit the applicability of our approach or its validity, and our results shall be revised when dealing with systems that do not satisfy these properties. Among them the most critical is the assumption of finite and constant variance that in the case of performance modeling may not always hold: For example, systems under heavy loads that have the utilization factor (ρ) close to or greater than the unity may not have finite variance for the average response time. Nevertheless, we empirically showed that common systems running on Clouds most of the time do not invalidate these assumptions. For example, their evolution has a slow pace, they are approximated well with continuous dimensions, and in general, have a smooth behavior.

Regarding the adaptability of the model, our approach can deal with all the systems' evolutions that do not require changing the input space of the models. For example, the approach can automatically capture improvements in the system performance due to code optimization, and reflect them in the models. However, the approach cannot automatically deal with structural changes in the system architecture. For example, if designers introduce an additional elastic tier, both controllers and Kriging models must be manually reconfigured to account for the added input feature, i.e, the amount of VMs allocated to the new tier.

6.3 Open Research

This thesis opens the road for new research in several domain:

Advanced model-based controllers This thesis can be the starting point for designing advanced solutions that employ new surrogate models, combine models in different ways, and employ time varying parameters in the models. These advanced controllers find their application in problems like dynamic resources allocation, autonomic design optimization, and more generally self-optimizing and self-managing system.

Experiment automation and automatic surrogate model training Designer can leverage the ability of Kriging models to highlight system configurations that are not optimally explored, and automatically plan for experiments to run. This is in line with the traditional research on models and simulations that commonly involve Kriging models for design optimization. Designers can also adopt similar techniques to plan staging tests with the aim of collecting data to produce good quality models, that is, experiments to optimally train surrogate models.

[1]Homoscedasticity is the property of having equal (and finite) statistical variances.

Model based testing and validation of dynamically varying systems Surrogate models and self-adaptive systems can be used to test dynamically varying or other self-adaptive systems. Designers can leverage the powerful modeling capabilities of surrogate models to capture the behavior of the self-adaptive systems and other dynamically varying systems, like elastic applications, and use the models to test or validate them.

Appendices

Appendix A

Use of `sig`

In this appendix, we report more details about the use of the confidence measure, i.e., `sig`, in designing robust control policies and for the implementation of the Combined model.

The importance of `sig`

`sig` is the uncertainty measure that the Kriging models pair with each prediction. The importance of `sig` relies in the fact that we can correlate it with the prediction error: If the value of `sig` is high then we might expect an high prediction error, while if the value of `sig` is low we expect a small prediction error.

It must be noted that “high” and “low” may takes very different values depending on the data set and application considered. If the dataset contains noisy data then the absolute value of the prediction error, and `sig` will be bigger than the ones obtained by not noisy data.

To evaluate the correlation between the prediction error and `sig` we follow a procedure similar to K fold cross validation: We split the original dataset in K subsets; we remove one of such subset, and we build a Kriging model with the remaining data; we predict the removed subset with the model just trained. We repeat this process for all the K subsets.

At each iteration, we compute the correlation between the `sig` and the absolute prediction error using the Spearman’s coefficients (usually denoted using the greek letter ρ). We repeat the same process ten times with $K = 20$, and we average the values of all the correlation coefficients.

The sign of the Spearman correlation indicates the direction of association between the two variables, and it increases in magnitude as they become closer to being perfect monotone functions of each other. If variables under study are perfectly monotonically related, the ρ becomes 1, at the contrary, if its value is -1 the variables have opposite signs, thus opposite directions.

We use the dataset of the Sun grid engine case study, and we show the result of the experiment in Figure A.1. The upper plot shows the actual and predicted values for all the K runs, the plot in the middle shows the corresponding absolute prediction error, and the plot at the bottom shows the corresponding `sig`. To ease the explanation we sort the data according to the absolute values of the predictions.

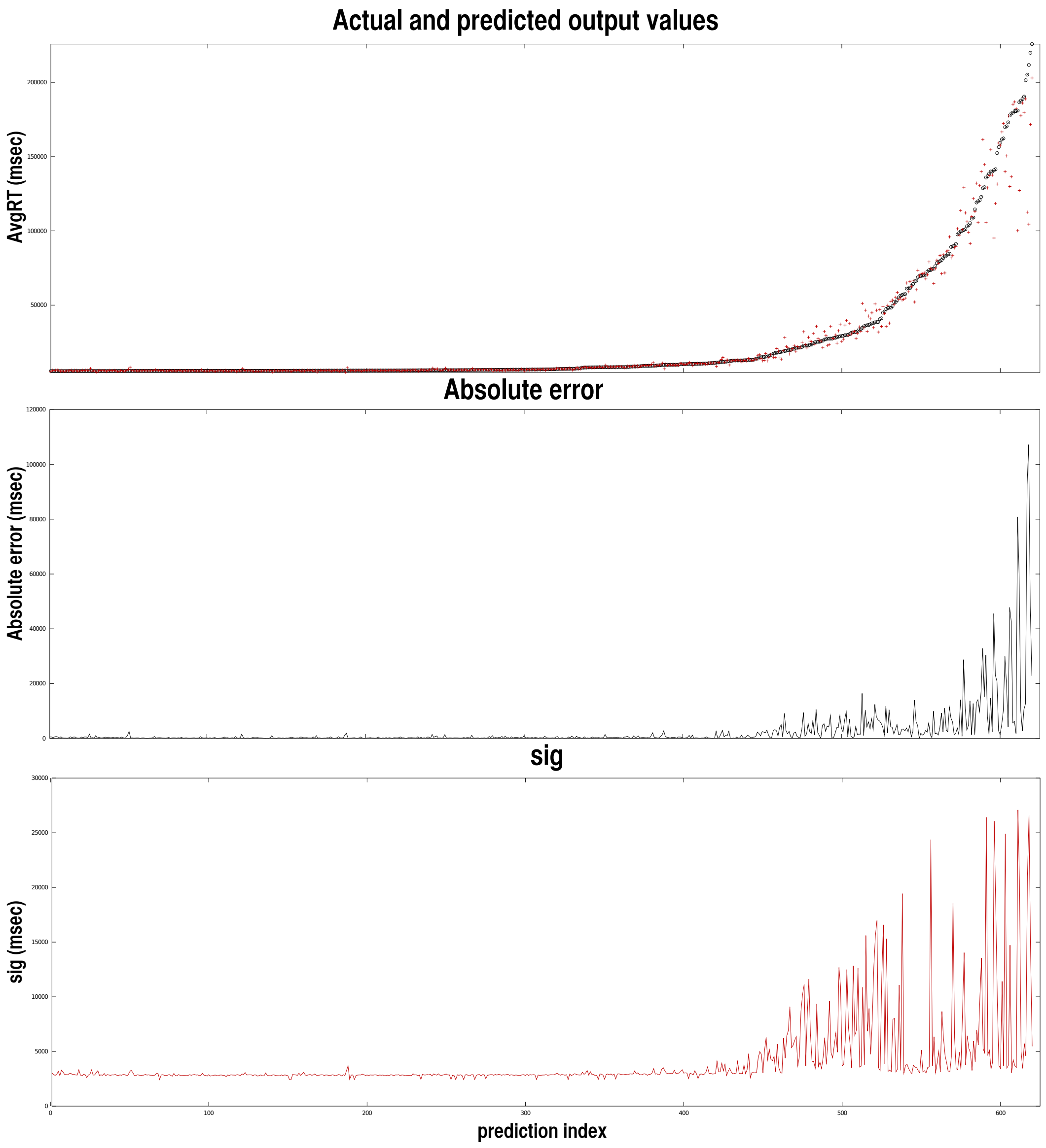

Figure A.1. Correlation of `sig` and absolute error

We obtain an average correlation coefficient of 0.68404 that means a good correlation between `sig` and the absolute prediction error.

The use of `sig` in robust control

We leverage the value of `sig` to decide whenever a prediction of the Kriging model is expected to be accurate enough and resort to another model otherwise, for example the analytic model presented in Chapter 5. To decide on the accuracy of a prediction we define a threshold on the ration between the `sig` value paired to the prediction and the maximum value of `sig` as given by the Kriging model. In case the resulting value is above the threshold we discard the prediction, otherwise we keep it.

We design a set of experiment to prove that combining Kriging model even with the simple analytical model and using `sig` to switch between them gives more accurate results than using the Kriging model alone. We run the K fold cross-validation and we compare the resulting value of the cross validation error (CVE). For the experiment, we set the threshold value to 0.4 (i.e., 40% of the maximum value of `sig` for the model).

Figure A.2 summarizes the results of the K-fold cross validation as bar plot: Each color corresponds to a different model and bars are grouped according to the value of K.

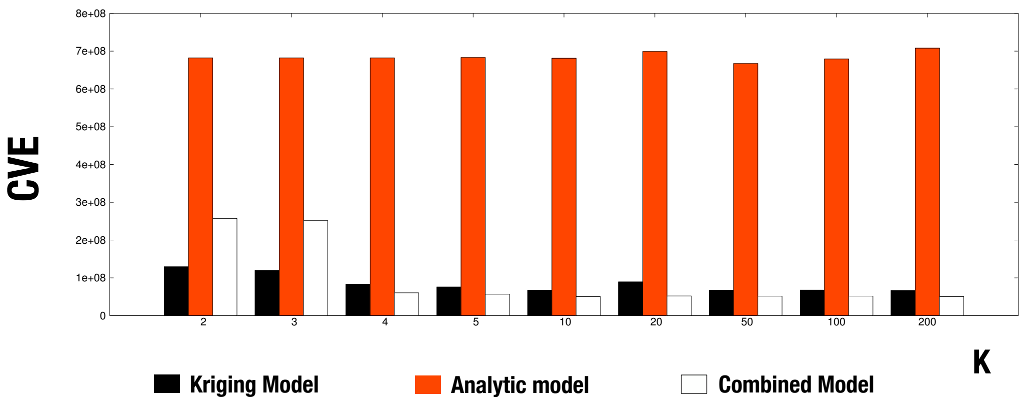

Figure A.2. Cross validation error comparison of standard Kriging, analytic model, and the combined model

From the height of the bars we can see that except for very small values of K the Combined model outperform both analytic model and the standard Kriging model. As more data are used for the training, the difference between the Combined and the Kriging models fades out.

The previous experiment showed that an aggregate value of the accuracy has an improvement by using the combined model. This however does not highlight clearly the benefits of the combination that appear by considering the locality of the prediction. To stress this point with train several Kriging models by partitioning the original dataset for the Sun grid engine in the following way: We set the value of K to define the size of each validation subset, but differently from the standard K-fold cross validation that takes random data points to form the validation set, we pick elements that are geometrically close in the input space, and we create a subset of data for each system configuration. This procedure enables us to create **on purpose** “holes” in the training sets that corresponds to entire system configurations. In the missing regions, we expect that the Kriging model will perform less good than the combined model because of the missing data points.

It must be noted that this situation may appear in real life: For example, this is usual when designers have not enough resources for collecting training data for all the possible system configurations.

For this experiment, we divided the possible configurations of the Sun grid engine in the following four subsets that we label with the minimum and maximum number of executors allocated: {2-5}, {6-10}, {11-15},{16-21}. We always keep the minimal, i.e., one executor, and the maximal, i.e., twenty-two executors, configurations in the training set to account for the boundaries of the systems and the interpolating nature of Kriging models.

Table A.1 reports the results of running this experiment using the original dataset for the Sun grid engine. The first column refers to the validation subset used to collect the data, in particular it identifies which configurations were removed from the training data. The second and third columns report the mean absolute error and the root square mean error for the standard Kriging models, while the next two report the same figures for the combined model.

Table A.1. Comparison of the prediction ability of the Kriging model and the Combined model

	Kriging		Combined	
Validation set	**MAE** msec	**RSME** msec	**MAE** msec	**RSME** msec
{2-5}	38960.38	46462.30	29916.24	39639.23
{6-10}	22946.66	38263.52	14318.20	18461.97
{11-15}	2125.05	3125.35	2127.23	2949.60
{16-21}	4239.01	5042.64	4166.53	4850.23

From the table we can see that almost all the time the combined model has a lower error compared to the standard Kriging model.

Appendix B

Evaluation

In this appendix, we report additional details about the evaluation process that we did not include in Chapter 5. In particular, the appendix reports the results that we obtained with different case studies for the evaluation of Kriging model accuracy and promptness, and more extensive results on the Sun grid engine.

Kriging models for prediction

Table B.1 summarizes the results we obtained using Kriging models to predict the average response time and throughput of the Doodle Web service and the DoReMap system. These results were originally presented in [31] and [103].

We collect data for training and validation through batch experiments and we average the data over the run to simulate the system steady state. In the table, the first column indicates the case study application, the second column indicates the performance index that is modeled, either average response time (Avg RT) or average throughput (Avg TX), the third column reports the size of the training set, while the fourth reports the size of the validation set, the last two columns show the error metrics.

Table B.1. Prediction ability of Kriging models for different applications and performance indexes

Application	**SLO**	**T Set**	**V Set**	**MAE**	**RMSE**
Doodle Web service	Avg RT	30	18	52 (msec)	91 (msec)
DoReMap	Avg TX	100	20	5.2788 (req/s)	8.4500 (req/s)

Kriging models for control

Robustness of the models

This section presents additional details about the robustness for the Combined model, regression tree and multivariate adaptive regression splines.

Figures B.1, B.2 and B.3 plots the predictions and the prediction error that we obtained with these models against the first dataset present in Section 5.3.

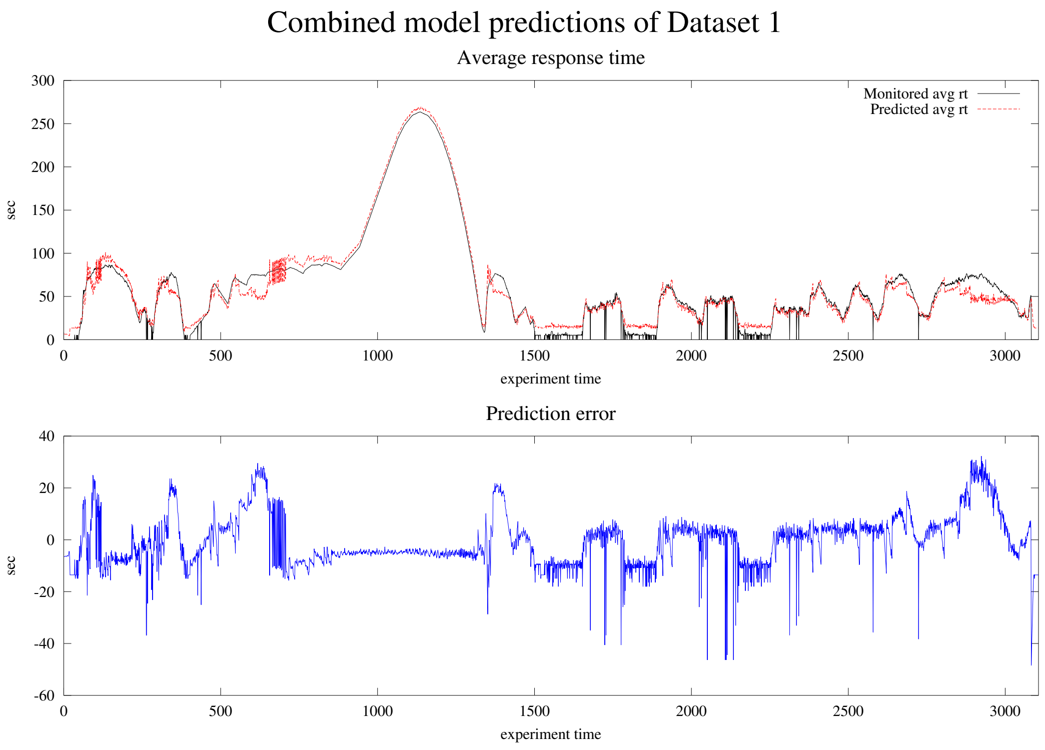

Figure B.1. Combined model predictions and prediction error on Dataset 1

Figures B.4, B.5 and B.6 show the results obtained with the second dataset.

Promptness of the models

Table B.2 reports results about promptness of the Kriging models with the Doodle We service and the DoReMap system. The tables has the same structure of Table **??** but we obtained the results using different settings. In particular, we use a single-vCPU VM running a Debian GNU/Linux 5.0 and Octave 3.0.1 on a dual-core Dell desktop to obtain the results in Table B.2.

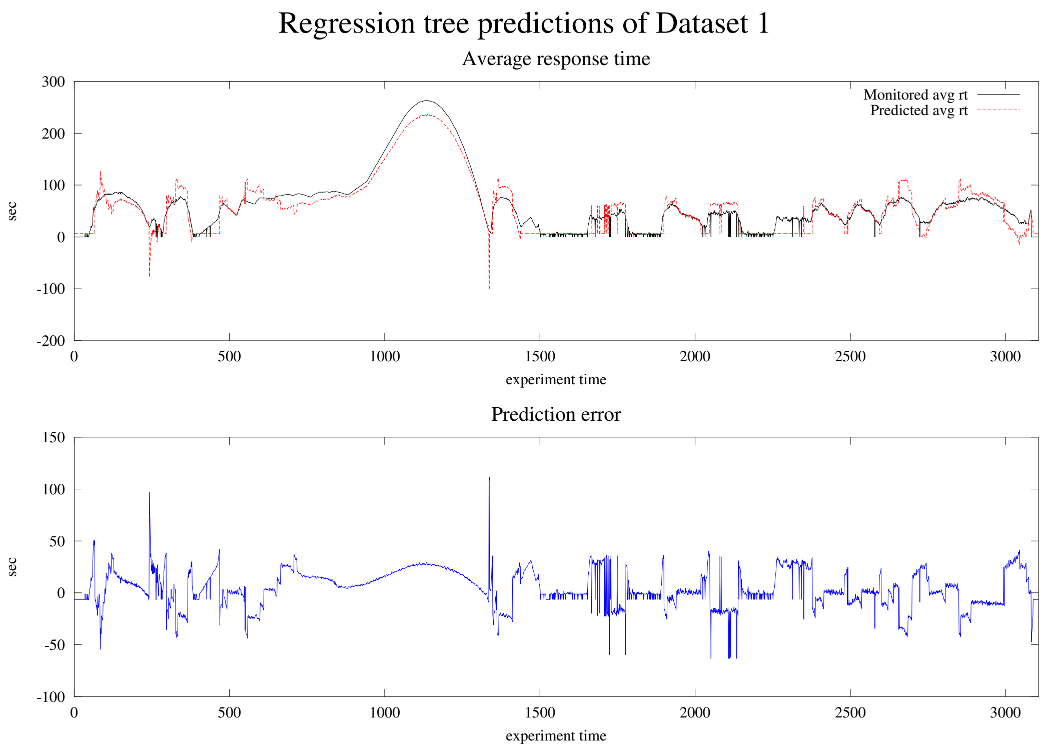

Figure B.2. Regression tree predictions and prediction error on Dataset 1

Table B.2. Kriging models training time

Case study	**Sample size**	**Training time (msec)**
Doodle Web service	6	18
Doodle Web service	12	18
Doodle Web service	18	15
Doodle Web service	30	146
Doodle Web service	48	112
DoReMap	50	24.20
DoReMap	75	48.15
DoReMap	100	87.45

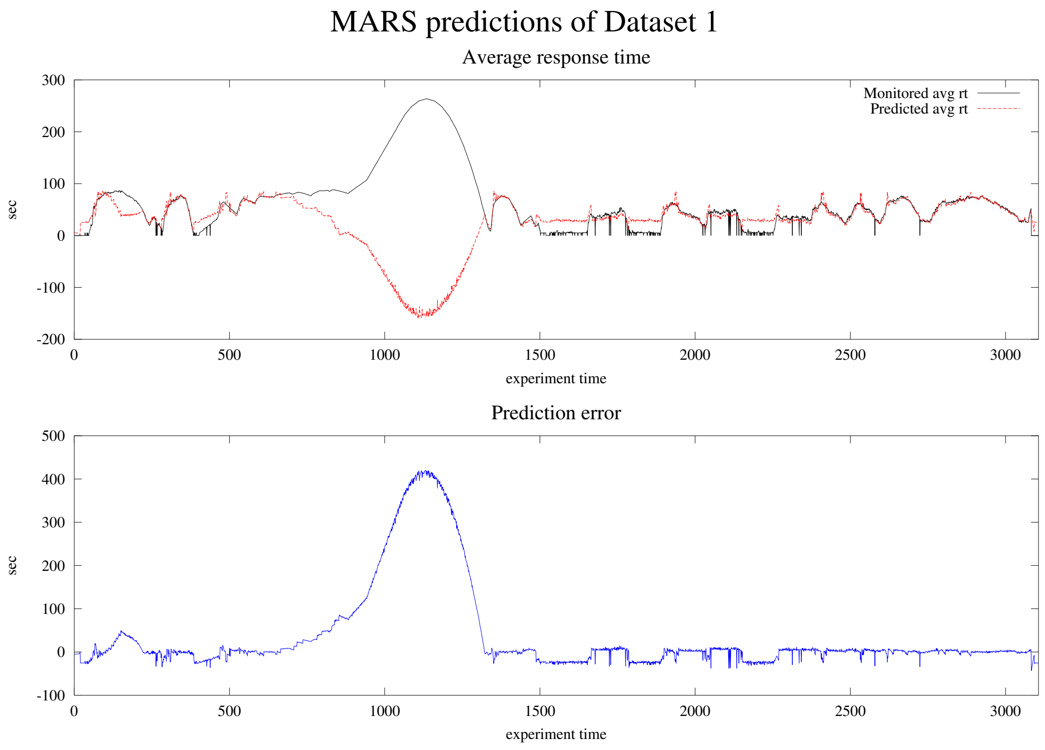

Figure B.3. Multivariate Adaptive Regression Splines predictions and prediction error on Dataset 1

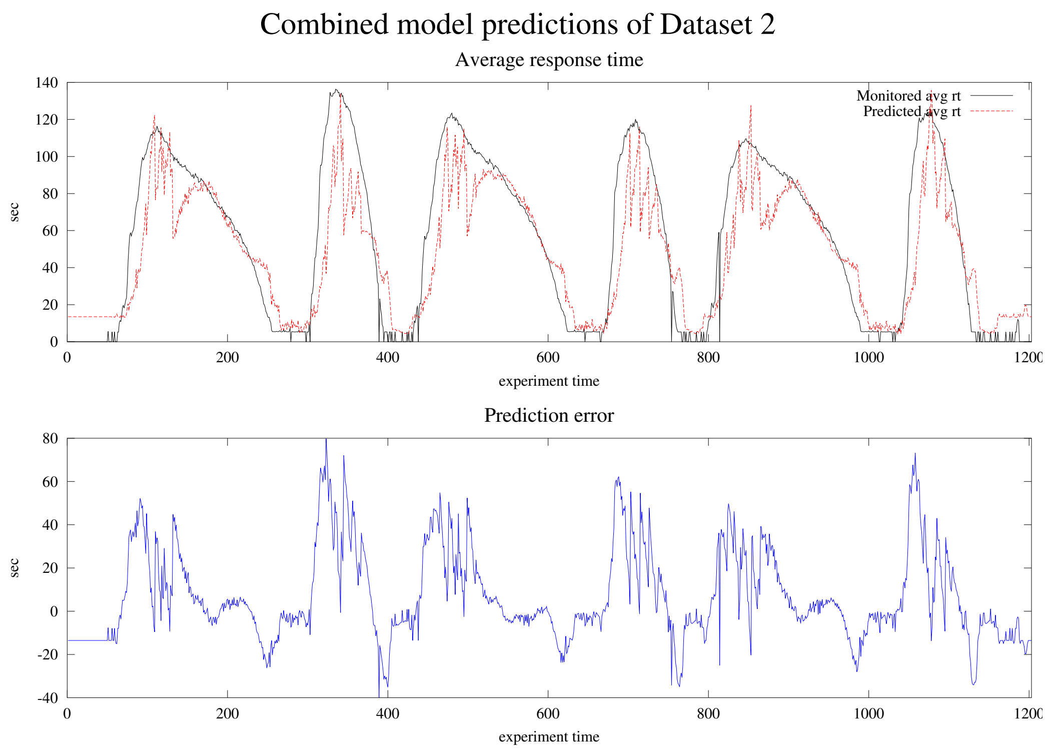

Figure B.4. Combined model predictions and prediction error on Dataset 2

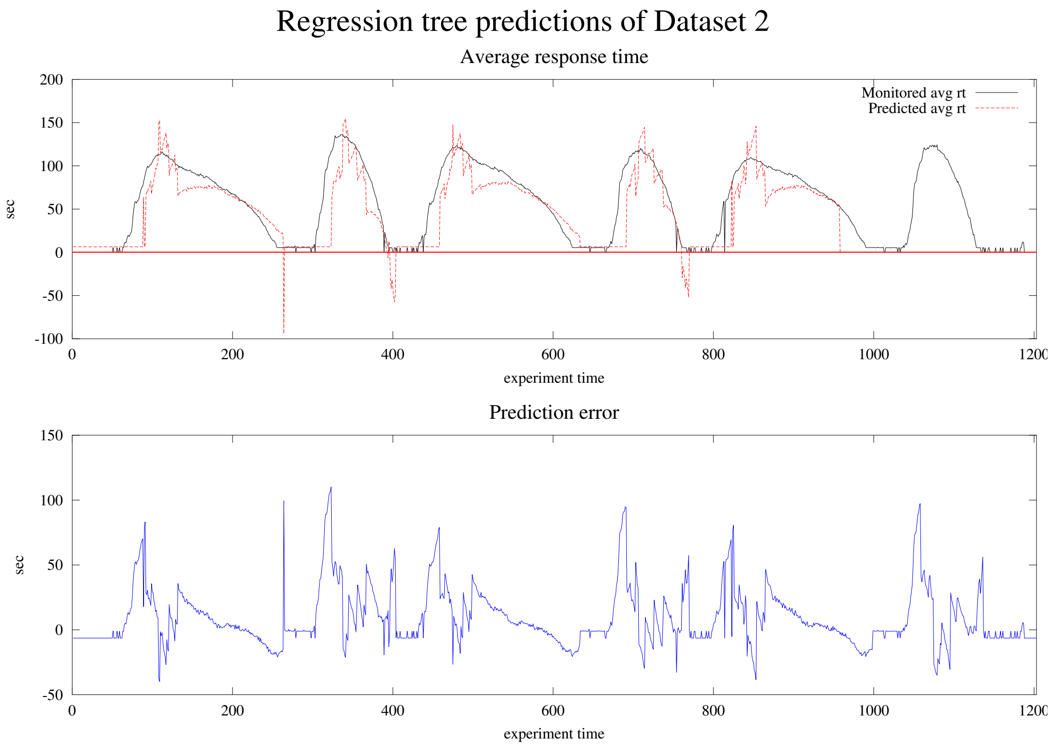

Figure B.5. Regression tree predictions and prediction error on Dataset 2

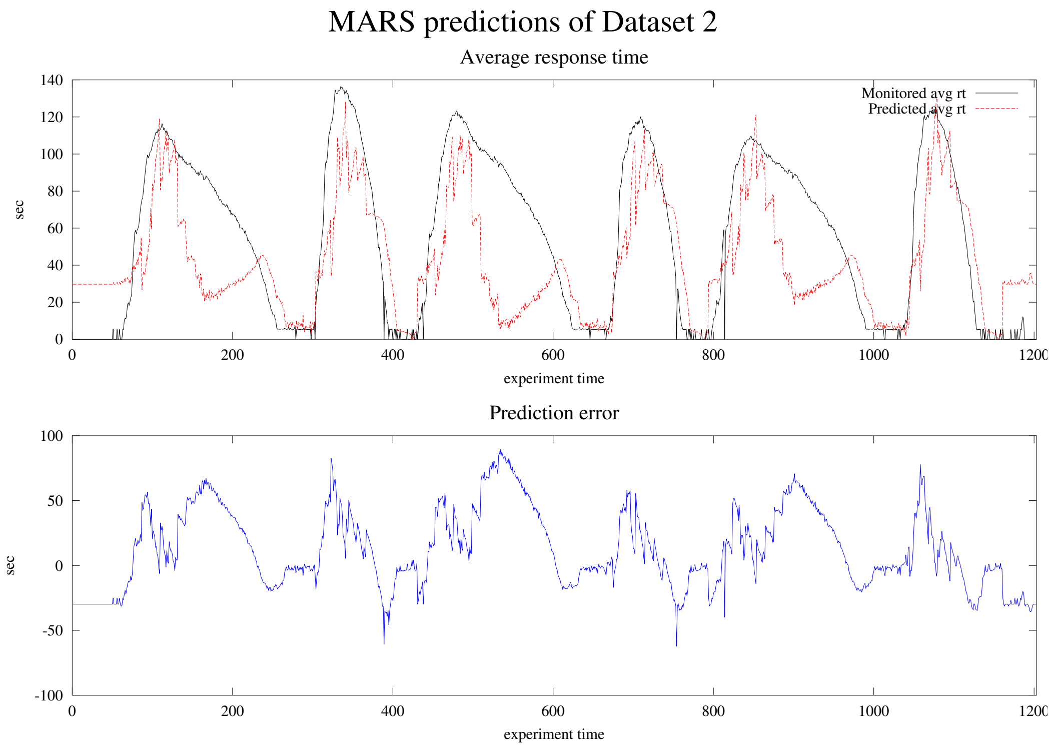

Figure B.6. Multivariate Adaptive Regression Splines model predictions and prediction error on Dataset 2

Bibliography

[1] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Proceedings of the International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 449–460, 2011.

[2] Virgilio A.F. Almeida and Daniel A. Menascè. Capacity planning: An essential tool for managing web services. *IT Professional*, 4(4):33–38, 2002.

[3] Mauro Andreolini, Sara Casolari, and Michele Colajanni. Autonomic request management algorithms for geographically distributed internet-based systems. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO'08, 2008.

[4] B. Ankenman, B. L. Nelson, and J. Staum. Stochastic kriging for simulation metamodeling. *Operational Research*, 58(2):371–382, 2010.

[5] Danilo Ardagna, Carlo Ghezzi, and Raffaela Mirandola. Model driven qoS analyses of composed web services. In *Proceedings of European Conference on Towards a Service-Based Internet (ServiceWave'08)*, pages 299–311, 2008.

[6] M. Armbrust, A. Fox, D. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-independent storage for social computing applications. In *Proceeding of Conference on Innovative Data Systems Research*, CIDR'09, 2009.

[7] Michael Armbrust, Armando fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Arie Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkley view of Cloud Computing. Technical Report No. UCB/EECS-2009-28, University of California at Berkley, 2009.

[8] Michael Armbrust, Armando fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Arie Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, and Rolf Neugebauer. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, 2003.

[10] Novella Bartolini, Gian Carlo Bongiovanni, and Simone Silvestri. Self-* overload control for distributed web systems. In *Proceedings of International Workshop on Quality of Service (IWQoS'08)*, 2008.

[11] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, Jan 2009.

[12] Fran Berman, Geoffrey Fox, and Anthony J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley &amp; Sons, Inc., New York, NY, USA, 2003.

[13] Jing Bi, Zhiliang Zhu, Ruixiong Tian, and Qingbo Wang. Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center. In *Proceedings of International Conference on Cloud Computing*, CLOUD'10, pages 370–377, July 2010.

[14] Peter Bodik, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the Conference on Hot topics in cloud computing*, HotCloud'09, pages 75–80, 2009.

[15] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M. Kienle, Marin Litoiu, Hausi A. Müller, Mauro Pezzè, and Mary Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*. Springer Verlag, 2009.

[16] Nicolo Maria Calcavecchia, Ofer Biran, Erez Hadad, and Yosef Moatti. Vm placement strategies for cloud scenarios. In *Proocedings of the International Conference on Cloud Computing*, 2012.

[17] Ludmila Cherkasova, Lei Lu, Vittoria de Nitto Personé, Ningfang Mi, and Evgenia Smirni. AWAIT: Efficient overload management for busy multi-tier web services under bursty workloads. In *Proceedings of International Conference on Web Engineering (ICWE'10)*, 2010.

[18] S. Clayman, A. Galis, and L. Mamatas. Monitoring virtual networks with lattice. In *IEEE/IFIP Network Operations and Management Symposium Workshops*, NOMS, pages 239–246, April 2010.

[19] Sand Correa and Renato Cerqueira. Statistical approaches to predicting and diagnosing performance problems in component-based distributed systems: An experimental evaluation. In *Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '10, pages 21–30, 2010.

[20] M. Deisenroth, R. Turner, M. Huber, U. Hanebeck, and C. Rasmussen. Robust filtering and smoothing with gaussian processes. *IEEE Transactions on Automatic Control*, 2011.

[21] Marc P. Deisenroth, Florian Weissel, Toshiyuki Ohtsuka, and Uwe D. Hanebeck. Online-computation approach to optimal control of noise-affected nonlinear systems with continuous state and control spaces. In *Proceedings of European Control Conference*, ECC'07, 2007.

[22] M.P. Deisenroth. *Efficient Reinforcement Learning using Gaussian Processes*. PhD thesis, Karlsruhe Institute of Technology, 2010, Revisited 2012.

[23] M.P. Deisenroth, C.E. Rasmussen, and J. Peters. Gaussian process dynamic programming. *Neurocomputing*, 72(7-9):1508–1524, 2009.

[24] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck. From data center resource allocation to control theory and back. In *Proceedings of International Conference on Cloud Computing*, CLOUD'10, pages 410–417, July 2010.

[25] Ernst & Young. Ready for takeoff: Preparing for your journey into the cloud. Business briefing, Ernst & Young, April 2012.

[26] Jacqueline Floch, Svein O. Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[27] A.I.J. Forrester, A.J. Keane, and N.W. Bressloff. Design and analysis of noisy computer experiments. *AIAA*, 44(10):2331–2339, 2006.

[28] Alexander I.J. Forrester and Andy J. Keane. Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*, 45(1–3):50 – 79, 2009.

[29] Jekabsons G. *M5PrimeLab: M5' regression tree and model tree toolbox for Matlab/Octave*, 2010.

[30] Jekabsons G. *ARESLab: Adaptive Regression Splines toolbox for Matlab/Octave*, 2011.

[31] Alessio Gambi, Mauro Pezzè, and Giovanni Toffetti Carughi. Protecting SLA with surrogate models. In *Proceedings of International Workshop on Principles of Engineering Service-Oriented Systems (PESOS'10)*, 2010.

[32] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 592–603, 2009.

[33] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10), 2004.

[34] Gartner, Inc. Paas road map: A continent emerging, 2011. Report ID Number: G00209751.

[35] Robert B. Gramacy, Herbert K. H. Lee, and William G. Macready. Parameter space exploration with gaussian process trees. In *Proceedings of the international conference on Machine Learning (ML'04)*, pages 353–360, 2004.

[36] Jens Happe, Dennis Westermann, Kai Sachs, and Lucia Kapová. Statistical inference of software performance models for parametric performance completions. In *Proceedings of International Conference on the Quality of Software Architectures*, volume 6093 of *QOSA'10*, pages 20–35, 2010.

[37] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley, September 2004.

[38] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Research, 2001.

[39] Nikolaus Huber, Fabian Brosig, and Samuel Kounev. Model-based Self-Adaptive Resource Allocation in Virtualized Environments. In *SEAMS'11: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Honolulu, HI, USA*, 2011. Acceptance Rate (Full Paper): 27% (21/76).

[40] IBM. An architectural blueprint for autonomic computing. Technical report, IBM Research, June 2006.

[41] IEEE. Ieee cloud computing initiative.

[42] Sadeka Islam, Jaky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the Cloud. *Future Generation Computer Systems*, 28:155–162, 2012.

[43] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of the International Conference on Performance Engineering*, ICPE '12, pages 85–96, 2012.

[44] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, WWW'10, pages 471–480, 2010.

[45] D.R. Jones, M. Schonlau, and W.J. Welch. Efficient global optimization of expensive black-box functions. *Global optimization*, 13(4):455–492, 1998.

[46] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 1–20, 2009.

[47] Gueyoung Jung, K.R. Joshi, M.A. Hiltunen, R.D. Schlichting, and C. Pu. Generating adaptation policies for multi-tier applications in consolidated server environments. In *Proocedings of the International Conference on Autonomic Computing and Communications (ICAC'08)*, pages 23–32, June 2008.

[48] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *Proocedings of the International Conference on Autonomic Computing and Communications*, ICAC'09, pages 117–126, June 2009.

[49] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[50] Bithika Khargharia, Salim Hariri, and Mazin S. Yousif. Autonomic power and performance management for computing systems. *Cluster Computing*, 11(2):167–181, 2008.

[51] J. Kocijan, R. Murray-Smith, C.E. Rasmussen, and B. Likar. Predictive control with gaussian process models. In *Proceedings of International Conference on Computer as a tool*, volume 1 of *EUROCON'03*, pages 352–356, Sept 2003.

[52] Samuel Kounev, Ramon Nou, and Jordi Torres. Autonomic qos-aware resource management in grid computing using online performance models. In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, ValueTools '07, pages 48:1–48:10, 2007.

[53] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Proceedings of Future of Software Engineering (FOSE'07)*, pages 259–268, 2007.

[54] D.G Krige. *A statistical approach to some mine valuations and allied problems at the Witwatersrand*. PhD thesis, University of Witwatersrand, 1951.

[55] Dara Kusic and Nagarajan Kandasamy. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. *Cluster Computing*, 10:395–408, December 2007.

[56] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. In *Proceedings of the 2008 International Conference on Autonomic Computing*, ICAC'08, pages 3–12, 2008.

[57] Philipp Leitner, Branimir Wetzstein, Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, and Frank Leymann. Runtime prediction of service level agreement violations for composite services. In *Proceedings of the Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*, 2009.

[58] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *Proceedings of the ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 23–31, 2009.

[59] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 1–14, 2010.

[60] Han Li and Srikumar Venugopal. Using reinforcement learning for controlling an elastic web application hosting platform. In *Proceedings of the international conference on Autonomic computing*, ICAC'11, pages 205–208, 2011.

[61] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceeding of the International Conference on Autonomic Computing*, ICAC'10, pages 1–10, 2010.

[62] Harold C. Lim, Shivnath Babu, Jeffrey S. Chase, and Sujay S. Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the workshop on Automated control for datacenters and clouds*, ACDC'09, pages 13–18, 2009.

[63] Greg Linden. Make data useful, November 2006.

[64] Marin Litoiu, Murray Woodside, Johnny Wong, Joanna Ng, and Gabriel Iszlai. A business driven cloud optimization architecture. In *Proceedings of the ACM Symposium on Applied Computing*, SAC '10, pages 380–385, 2010.

[65] Xue Liu, Jin Heo, Lui Sha, and Xiaoyun Zhu. Queueing-model-based adaptive control of multi-tiered web applications. *IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT*, 5(3):157–167, Sept 2008.

[66] Martina Maggio, Henry Hoffmann, Alessandro Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision making strategies for self-optimization in autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, pages 1–33, To appear.

[67] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. A comparison of autonomic decision making techniques. Technical report, Massachusetts Institute of Technology (MIT), April 2011.

[68] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Decision making in autonomic computing systems: comparison of approaches and techniques. In *Proceedings of the International Conference on Autonomic Computing*, ICAC'11, pages 201–204, 2011.

[69] Simon Malkowski, Markus Hedwig, and Calton Pu. Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'09, pages 118–127, 2009.

[70] Simon J. Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC'11, pages 131–140, June 2011.

[71] Giovanni Mariani, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Meta-model assisted optimization for design space exploration of multiprocessor systems-on-chip. In *Proceedings of Euromicro Conference on Digital System Design (DSD'07)*, pages 383–389, 2009.

[72] G Matheron. Principles of geostatistics. *Economic Geology*, 58:1246–1266, 1963.

[73] Daniel A. Menascè and Mohamed N. Bennani. On the use of performance models to design self-managing computer systems. In *Proceedings of Conference on Computer Measurement Group*, pages 1–9, Dec 2003.

[74] Adina D. Mosincat and Walter Binder. Automated maintenance of service compositions with sla violation detection and dynamic binding. *International Journal on Software Tools for Technology Transfer*, 13(2):167–179, 2011.

[75] Adina D. Mosincat, Walter Binder, and Mehdi Jazayeri. Runtime adaptability through automated model evolution. In *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference*, EDOC'10, pages 217–226, 2010.

[76] Mohamed N. Bennani and Daniel A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the International Conference on Automatic Computing*, ICAC'05, pages 229–240, 2005.

[77] D. Nguyen-Tuong and J. Peters. Local gaussian process regression for real-time model-based robot control. In *Proceeding of International Conference on Intelligent Robots and Systems*, IROS, pages 380–385, 2008.

[78] Takayuki Osogami and Sei Kato. Optimizing system configurations quickly by guessing at the performance. In *Proceedings of the International conference on Measurement and modeling of computer systems*, SIGMETRICS'07, pages 145–156. ACM, June 2007.

[79] Pradeep Padala, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kang Shin. What does control theory bring to systems research? *ACM SIGOPS Operating Systems Review*, 43(1):62–69, 2009.

[80] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(2):223–255, 2008.

[81] K. Parent. Server consolidation improves it's capacity utilization. Technical report, Court Square Group, 2005.

[82] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. A multimodel framework to implement self-managing control systems for qos management. In *Proceeding of the international symposium on Software engineering for adaptive and self-managing systems*, SEAMS '11, pages 218–227, 2011.

[83] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. Survey on the design of self-adaptive software systems using control engineering approaches. In *Proceeding of the international symposium on Software engineering for adaptive and self-managing systems*, SEAMS'12, 2012.

[84] C. Pautasso and T. Heinis. Automatic configuration of an autonomic controller: An experimental study with zero-configuration policies. In *Proocedings of the International Conference on Autonomic Computing and Communications (ICAC'08)*, pages 67–76, 2008.

[85] Cesare Pautasso. Composing RESTful services with JOpera. In *Proceedings of the International Conference of Software Composition*, volume 5634 of *LNCS*, pages 142–159, 2009.

[86] Cesare Pautasso and Gustavo Alonso. The JOpera visual composition language. *Journal of Visual Language and Computing*, 16(1-2):119–152, 2005.

[87] Avenade Perspective. Server virtualization: A step toward cost efficiency and business agility. Technical report, Avenade, 2009.

[88] D. Petelin and J. Kocijan. Control system with evolving gaussian process models. In *Proceedings of IEEE Workshop on Evolving and Adaptive Intelligent Systems*, EAIS ’11, pages 178–184, April 2011.

[89] Andres Quiroz, Hyunjoo Kim, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *Proceedings of the IEEE/ACM International Conference on Grid Computing*, GRID’09, pages 50–57, 2009.

[90] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[91] George Reese. *Cloud Application Architectures. Building Applications and Infrastructure in the Cloud*. O’Reilly, 2009.

[92] Ivan Rodero, Eun Kyung Lee, Dario Pompili, Manish Parashar, Marc Gamell, and Renato J. Figueiredo. Towards energy-efficient reactive thermal management in instrumented datacenters. In *Proceedings of IEEE/ACM International Conference on Grid Computing*, pages 321–328, 2010.

[93] Luis Rodero-Merino, Luis Miguel Vaquero Gonzalez, Victor Gil, Fermín Galán, Javier Fontán, Rubén S. Montero, and Ignacio Martín Llorente. From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, 26(8):1226–1240, 2010.

[94] J. Sacks, W.J. Welch, T.J. Mitchell, and H.P. Wynn. Design and analysis of computer experiments. *Statistical science*, 4(4):409–423, 1989.

[95] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Proceedings of International Conference on Distributed Computing Systems*, ICDCS’11, pages 559–570, June 2011.

[96] M.B. Sheikh, U.F. Minhas, O.Z. Khan, A. Aboulnaga, P. Poupart, and D.J. Taylor. A bayesian approach to online performance modeling for database appliances using gaussian models. In *Proceedings of the International conference on Autonomic computing*, ICAC’11, pages 121–130, 2011.

[97] Piyush Shivam, Varun Marupadi, Jeff Chase, Thileepan Subramaniam, and Shivnath Babu. Cutting corners: workbench automation for server benchmarking. In *Proceedings of the USENIX Annual Technical Conference*, USENIX-ATC’08, pages 241–254. USENIX Association, 2008.

[98] Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceeding of the International Conference on Autonomic Computing*, ICAC'10, pages 21–30, 2010.

[99] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server virtualization in autonomic management of heterogeneous workloads. In *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM'07)*, pages 139–148, 2007.

[100] Gabriel Tamura, Norha M. Villegas, Hausi Muller, Joao P. Sousa, Basil Becker, Gabor Karsai, Serge Mankovskii, Mauro Pezze, Wilhelm Schafer, Ladan Tahvildari, and Kenny Wong'. Towards practical run-time verification and validation of self-adaptive software systems. In Rogerio de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems*, Dagstuhl Seminar Proceedings, 2011.

[101] Gerald Tesauro, Nicholas K. Jong, Rajarshi Das, and Mohamed N. Bennani. On the use of hybrid reinforcement learning for autonomic resource allocation. *Cluster Computing*, 10(3):287–299, 2007.

[102] The Reservoir Seed Team. High level architectural specification. Technical Report 2, Reservoir FP7, December 2009.

[103] Giovanni Toffetti, Alessio Gambi, Cesare Pautasso, and Mauro Pezzè. Engineering autonomic controllers for virtualized web applications. In *Proceedings of International Conference on Web Engineering (ICWE)*, LNCS, 2010.

[104] Beth Trushkowsky, Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: scaling a distributed storage system under stringent performance requirements. In *Proceedings of the USENIX conference on File and stroage technologies*, FAST'11, pages 12–26, 2011.

[105] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the International Conference on Automatic Computing*, ICAC'05, pages 217–228, 2005.

[106] Bhuvan Urgaonkar, Prashant J. Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1):1–39, March 2008.

[107] W. van Beers and J.P.C. Kleijnen. Kriging interpolation in simulation: a survey. In *Proceedings of the Winter Simulation Conference*, WSC'10, pages 113–121. Institute of Electrical and Electronics Engineers, University of Michigan, 2004.

[108] Nedeljko Vasic, Dejan Novakovic, Svetozar Miucin, Dejan Kostic, and Ricardo Bianchini. DejaVu: Accelerating resource allocation in virtualized environments. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2012, page 13, March 2012.

[109] W. Vogels. A head in the cloud – the power of infrastructure as a service. In *Proocedings of the First workshop on Cloud Computing and Applications*, CCA'08, 2008.

[110] Gary G. Wang and S. Shan. Review of metamodeling techniques in support of engineering design optimization. *Mechanical Design*, 129(4):370–380, 2007.

[111] Dennis Westermann, Jens Happe, Michael Hauck, and Christian Heupel. The performance cockpit approach: A framework for systematic performance evaluations. In *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA'10, pages 31–38, 2010.

[112] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of International Conference On Automated Software Engineering*, ASE'12, pages 190–199, Sept 2012.

[113] Dennis Westermann, Rouven Krebs, and Jens Happe. Efficient experiment selection in automated software performance evaluations. In *Proceedings of the European conference on Computer Performance Engineering*, EPEW'11, pages 325–339, 2011.

[114] Dennis Westermann and Christof Momm. Using software performance curves for dependable and cost-efficient service hosting. In *Proceedings of the International Workshop on the Quality of Service-Oriented Software Systems*, QUASOSS'10, pages 1–6, 2010.

[115] Danny Weyns, Robrecht Haesevoets, Bart Van Eylen, Alexander Helleboogh, Tom Holvoet, and Wouter Joosen. Endogenous versus exogenous self-management. In *Proceedings of International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08)*, pages 41–48, 2008.

[116] C. Murray Woodside and Marin Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering*, 34:391–406, May 2008.

[117] M. Woodside, Tao Zheng, and M. Litoiu. Service system resource management based on a tracked layered performance model. In *Proocedings of the International Conference on Autonomic Computing and Communications*, ICAC'06, pages 175–184, 2006.

[118] Jing Xu, Ming Zhao, Jose Fortes, Robert Carpenter, and Mazin Yousif. On the use of fuzzy modeling in virtualized data center management. In *Proceeding of the International Conference on Autonomic Computing*, ICAC'07, pages 25–35, 2007.

[119] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the International Conference on Autonomic Computing*, ICAC'07, pages 27–36, 2007.

[120] Rui Zhang and Alan J. Bivens. Comparing the use of bayesian networks and neural networks in response time modeling for service-oriented systems. In *Proceedings of the Workshop on Service-oriented computing performance: aspects, issues, and approaches*, SOCP'07, pages 67–74, 2007.