Neural Adaptive Admission Control Framework: SLA–Driven Action Termination for Real–Time Application Service Management

Tomasz D. Sikora[a] and George D. Magoulas[b]

[a] `sikora.t@gmail.com`, [b] `gmagoulas@dcs.bbk.ac.uk`
Department of Computer Science and Information Systems,
Birkbeck Colledge, University of London, Malet Street, London WC1E 7HX, UK

ARTICLE HISTORY
Compiled February 28, 2019

ABSTRACT

Although most modern cloud–based enterprise systems, or operating systems, do not commonly allow configurable/automatic termination of processes, tasks or actions, it is common practice for systems administrators to manually terminate, or stop, tasks or actions at any level of the system. The paper investigates the potential of automatic adaptive control with action termination as a method for adapting the system to more appropriate conditions in environments with established goals for both system's performance and economics. A machine–learning driven control mechanism, employing neural networks, is derived and applied within data–intensive systems. Control policies that have been designed following this approach are evaluated under different load patterns and service level requirements. The experimental results demonstrate performance characteristics and benefits as well as implications of termination control when applied to different action types with distinct run–time characteristics. An automatic termination approach may be eminently suitable for systems with harsh execution time Service Level Agreements, or systems running under conditions of hard pressure on power supply or other constraints. The proposed control mechanisms can be combined with other available toolkits to support deployment of autonomous controllers in high–dimensional enterprise information systems.

KEYWORDS
Neural Autonomous Control, Application Service Management, Cloud Computing, Service Modelling, Service Level Agreement.

1. Introduction

Currently, data centres employ much more computational power, machines and energy than it is really needed to provide the required level of service and ensure customer satisfaction. This is a result of holding extra servers in reserve, ready to be brought into operation in case of high demand, and is a consequence of the effect of high switching on/off inertia of the reserve systems (Mao and Humphrey 2012). Theoretically, this strategy would appear as inefficient but in practice it is considered as an effective approach to secure some strongly desired non-functional requirements, such as resiliency, scalability and reliability (Zhang and Lin 2010).

The Cloud Computing paradigm tackles this problem distributing computational power more efficiently through the allocation and reuse of resources, hardware and services to several clients. Cloud auto–scaling based on resources availability has been researched thoroughly, including considerations about deadlines and budget constraints, both from practical and theoretical perspectives (Mao, Li, and Humphrey 2010; Dutreilh et al. 2010). In an attempt to provide solutions that would optimise energy usage and available equipment utilisation under dynamic workload changes, and at the same time they would offer the flexibility of a computation framework, industry has developed a range of cloud computing approaches ranging from Infrastructure as a Service (IaaS), Software as a Service (SaaS), Platform as a Service (PaaS), to serverless and Function as a Service (FaaS) computing. Other relevant areas that have provided additional benefits are advances in flexible containerisation, micro–services, and distributed computing (García-Valls, Cucinotta, and Lu 2014). All these cloud computing and distributed computing developments aim at the efficiency of power and infrastructure use, and ultimately seek to fully utilise the entire computational power available. However, the closer we are to that goal, the probability of facing risks related to fulfilled capacity and overused infrastructure rises. This can lead to operating scenarios that generate much harder problems for task scheduling, more difficult control situations, much longer processing times, and in effect can have an impact on Service Level Agreements (SLA). Any violated SLAs may cause high fines to be paid as a result of contractual penalties. Whatever has not been optimised in a SLA negotiation phase on a market level (Sim 2013) will have to be dealt in later operation. Application Service Management (ASM) tackles control aspects of the operating environment by attempting to balance cost and service levels objectives. This paper considers that these objectives can be reduced to monetization, where all resources, usage profiles (services calls and load profile) and SLAs can be aggregated down to cost and revenue calculations under consideration in SLA definitions.

This also relates to advances in Power–Aware Computing, focusing to maximise the computation efficiency as a function of energy consumed (Fan, Weber, and Barroso 2007; Kim, Beloglazov, and Buyya 2009; Berl et al. 2010; Kim, Beloglazov, and Buyya 2011) relying on dynamic resources allocation (Beloglazov, Abawajy, and Buyya 2012; Lee and Zomaya 2012) or virtual machines placement (Beloglazov and Buyya 2010; Moreno et al. 2013; Duolikun, Enokido, and Takizawa 2017), but may also be facilitated by application action level control (Sikora and Magoulas 2015). Moreover, this approach can be beneficial in the context of Energy Proportional Computing, where the main issue is to deal with the high static power that is caused by the fact that a computer consumes significant energy when it is idle but also to control the power to operate under load, which is not linear (Barroso and Hölzle 2007). Low dynamic range and poor efficiency at low to medium CPU utilisation (low energy proportionality) is generated when there is a high static power relative to the maximum loaded power (Barroso, Clidaras, and Hölzle 2013). This makes natural for cloud providers to consider upper resources utilisation ranges, unavoidably reaching saturation levels; that is an area where termination control, discussed further in the paper, can provide a good countermeasure to methods relying on dynamic resources allocation.

Currently, the cloud computing industry does not offer effective hard real-time guarantees as the emphasis is on resources provision, e.g. existing hypervisors provide no guarantee on latency, and there is lack of SLAs on latency. However, these types of requirements are going to be more and more present in the near future, as the demands for latency sensitive applications increase, e.g. cloud gaming, communication, streaming, making simple cloud outsourced resources allocation, with calculated headroom, is not adequate (Liu, Quan, and Ren 2010; Duy, Sato, and Inoguchi 2010; Zhu et al. 2014).

In this context, the paper investigates how embedding neural-based admission control into the application run-time, terminating actions before execution on a server but also during their processing in the application, can help managing different types of real-world load patterns and at the same time can meet requirements of profit/penalty oriented SLAs.

The rest of the paper is organised as follows. Section 2 reviews literature that relates to this research. Section 3 introduces the formulation of the problem used in this work. Section 4 presents the proposed control framework and strategy. Section 5 introduces the reader to aspects of the testbed design, and describes architectural aspects and components. Section 6 presents test cases and experiments conducted to validate the proposed adaptive control strategy. Section 7 discusses results and potential applications of the service management approach to cloud computing. Section 8 ends the paper by providing concluding remarks and insights about aspects that deserve further investigation.

2. Related Work

Scheduling mechanisms for tasks and requests, resource allocation, load–balancing, auto-scaling and admission control on the cloud have been the topics of considerable research attention in recent years. Most of the research done so far has tackled the control problem using a combination of components working in an inter-dependent manner, often operating in a sequence where admission control is done first, and then scheduling and resource allocation are utilised. For example, such an approach can be found in the work of Urgaonkar and Shenoy (2004) that aims to manage CPU and network bandwidth in shared clusters. Also, Yu et al. (2008) used admission control in the context of control of virtual network embedding, while Sharifian, Motamedi, and Akbari (2008) applied admission control for a cluster of web servers together with load balancing, and Almeida et al. (2010) proposed joint admission control and resource allocation for virtualized servers. Lastly, Ferrer et al. (2012) introduced a framework, called OPTIMIS, for cloud service provisioning where admission control is a key component.

More recently, there has been a lot of interest in cost–aware control. For example, Malawski et al. (2015) proposed methods for cost– and deadline–constrained resources provisioning in IaaS using a priority–based scheduling algorithm. Also, Yuan et al. (2016) built workload scheduling integrated with admission control for distributed cloud data centres, whilst Bi et al. (2017) test application–aware dynamic resource provisioning in cloud data centres. Whilst Ranaldo and Zimeo (2016) applied more proactive measures and propose capacity–driven utility model for SLA bilateral negotiation to optimise the utility for cloud service providers, costs and penalties prices, Messina et al. (2014, 2016) discuss an agent based negotiation protocol for cloud SLA.

There have also been approaches that exploit learning algorithms working in supervised or unsupervised model. For example, Muppala, Chen, and Zhou (2014) applied a model independent reinforcement learning with cascade neural networks technique for load proportional auto-configuration of virtual machines and session based admission control. Other researchers have proposed to steer admission control with neural controller and support vector machines (Mohana and Thangaraj 2013), and use scheduling algorithms based on meta–heuristic optimisation Hoang et al. (2016). Database access with profit–oriented control has been researched by Xiong et al. (2011), while requests preemption in admission control context was studied by (Salehi, Javadi, and Buyya

2012).

Another part of research has concentrated on requests termination or actions cancellation. Although work in this area has been limited, Cherkasova and Phaal (2002) proposed an admission control mechanism based on server CPU utilisation using four different strategies, and Leontiou, Dechouniotis, and Denazis (2010) used Kalman filtering and ARMAX models to predict changes in incoming load in order to support stable adaptive admission control of distributed cloud services. Zheng and Sakellariou (2013) built admission control with work–flow heuristic to evaluate schedule plans that meet budget-deadline constrains, while He et al. (2014) employ network calculus to perform admission control on the cloud tackling large number of parallel service requests.

Work in this area has been summarised in a number of survey papers, which cover general cloud control aspects (Buyya et al. 2009), present a network embedding perspective (Fischer et al. 2013), focus on auto–scaling techniques for elastic applications (Lorido-Botran, Miguel-Alonso, and Lozano 2014), or review machine learning approaches for energy–efficient resource management in cloud computing environments (Demirci 2015). Although, as mentioned above, there have been several control approaches and experimental platforms, which have been deployed in both model–based and real–world systems with the use of synthetic and benchmark workloads, there is still no clear established standard to enable an easy performance comparison.

Despite the massive progress in pre–call request termination as the main admission control strategy, we are not aware of any documented research that focuses on action termination, tackling pre– and during– request execution inclusively. In our view, it is imperative to explore the potential of these control approaches in isolation without utilising scalability and resource allocation methods, which could introduce obfuscation to cumulative results.

In this context, the paper contributes a framework that allows embedding the neural admission control into the application run–time. It builds on a previously introduced learning controller framework for adaptive control in ASM environments (Sikora and Magoulas 2013). This framework is extended, it is equipped with re–training capabilities and it is incorporated into a model–based testbed, where the collection of results is more efficient allowing the execution of long test runs involving real–systems experimentation. Emphasis is placed on more complex, business–oriented scenarios, where SLAs are defined as functions of not only call counts, as in our previous work, but also execution time, and are converted directly into revenue, as per the defined monetization model of SaaS, or PaaS systems.

The enhanced framework enables to terminate actions before execution on a server but also during the processing of the requests in the application. It also supports building synthetic workload profiles that are able to reproduce behaviours of real–world load patterns, where priority is set as part of profit/penalty– oriented SLAs with termination supporting real-time systems and harsh time–wise conditions. The new control approach is validated through synthetic and real–world scenarios, built from several system and workload profiles, which have been used by other researchers in the literature. The proposed action termination solution is further evaluated using four different control types, aiming at assessing both performance and cost–effectiveness.

The next section introduces the main concepts behind our approach and presents a theoretical formulation of the problem.

3. SLA–Driven Services Management

In our formulation, financial performance is tightly associated with the ability to offer the best available service under a contracted SLA. Financial performance P is defined as a composition of revenue R and costs C, so that:

$$P = \sum (R - C) \sim f_{SLA};\ R, C \in \mathbb{R}_+ \cup \{0\}\ . \tag{1}$$

Hence, the main objective is to optimise the service of the system, i.e. maximise the effectiveness in the background of load, resources usage, and performance characteristics of the service by reducing the costs defined in the SLA. All those aspects are time–variant; thus, adaptivity is a key requirement in a control system. The controller should be capable of readjusting the characteristics of actions' execution time at run–time using only termination actions and without changing other functionalities.

To this end, we define the scope of possible actions as undertaking revenue/costs related decisions, in line with what the parties have agreed to do when the conditions are met. This allows to transform the SLA, defined as a set of service–level objectives represented as *if...then* structures, into a function of actions execution time and incorporate it into a more complex function, where costs and revenues of the same argument are considered (see Equation 1, and Equation 2, below).

3.1. *Defining the Control System*

Suppose an enterprise system receives incoming requests to process action a in order to generate a particular service outcome. This is achieved by employing resources $r(t)$, according to an effectiveness criterion defined in an SLA function f_{SLA}. Hence, the general performance of the system can be considered as a dynamic process of all activities $\mathbf{a}$ and resources $\mathbf{r}$, and is produced under some conditions, as a result of running the system code and providing some functionality on a given hardware/software configuration.

The output of the system is fed back through a monitoring facility, which employs sensor measurements, and is compared with a reference value that has been estimated from a collection of historic data (measurements/metrics). The controller considers the difference between the reference point and the current output (actions execution times and resources in our case) to change the inputs to the system under control. This type of controller design follows the paradigm of the closed–loop feedback control (Hellerstein et al. 2004), and the controller is suitable for a multi–input multi–output (MIMO) enterprise system, defined in a high–dimensionality space of $\mathbb{R}^{n+1} \rightarrow \mathbb{R}^{n+1}, n = k + m$ in time domain, where k is the number of measurable system actions and m is the number of system resources (Sikora and Magoulas 2013).

The system performance is represented by the SLA function values and is defined as

$$P \sim f_{SLA_a} = f_R(a_e) - f_C(a_{et}) - f_C(a_{term}) - f_C(r)\ , \tag{2}$$

where $f_R(a_e)$ denotes revenue, a function of action's executions that is the main factor exposed to the client as a price for a particular service, $f_C(a_{et})$ represents the cost of SLA violations, which is a function of action's execution time, $f_C(a_{term})$ represents the cost of penalties for terminated actions, and lastly, $f_C(r)$ denotes the cost of resources, e.g. infrastructure provisioning. Due to the fact that, in this context, most of the computation cost can be specifically defined for a given action, one can derive the cost of resources needed to support a specific computation. Thus, $f_C(r)$ considers mainly the infrastructure required to support the service. Effectively, infrastructure costs are less dependent on computing fluctuations, and in the control approach adopted in this work we have assumed that this function is constant. So we consider that the infrastructure is not subjected to control; therefore, any optimization does not consider this dimension, which can be set as a parameter that takes a constant value.

A system state S at time t is defined as a vector of collected metrics about resources utilisation, $r(t)$, but also system input load to actions $a(t)$, and outputs, such as execution times and SLA values $f_{SLA_a}(t)$. During operation, the evolution of the system states forms a trajectory in the state space $\mathbf{S}$, where the search for the best available solution takes place:

$$S(t_c) = [\mathbf{a}, \mathbf{r}],\ S(t_c) \in \mathbf{S} \ , \tag{3}$$

where $\mathbf{a}, \mathbf{r}$ are vectors of actions and resources collected:

$$\begin{aligned} &\mathbf{a} = [a_0(t_c), f_{SLA_{a0}}(t_c), ..., a_n(t_c), f_{SLA_{an}}(t_c)] \ , \\ &\mathbf{r} = [r_0(t_c), ..., r_m(t_c)] \end{aligned} \tag{4}$$

at a specific time instance t_c.

In order to maximise the productivity of the system, P (Eq. 2), the controller evaluates system states, S, attempting to reconcile service workload economics, defined by incoming activity, $f_R(a_e)$, with utilised resources, $f_C(r)$, on one hand, and other limiting costs of SLA violations, $f_C(a_{et})$, and actions terminations, $f_C(a_{term})$, on the other hand. Consequently, the control process aims to generate a sequence of system states that would lead to $\max f_{SLA}(t) \in \mathbf{S}$. Thus, a key performance metric of the control system is the sum of SLA values, in time, for all action types, which can be considered as a cumulative indicator of productivity of the service provider that shapes the landscape of profitability from service provisioning perspective:

$$T_{SLA}(t) = \sum_i f_{SLA_i}(t) \ , \tag{5}$$

where $f_{SLA_i}(t)$ is a sequence of collected values in time t (i.e. a time–series) that incorporates costs and revenue for an action type SLA, f_{SLA_a}.

3.2. *Action Termination Control*

In certain situations, mainly when there is excessive utilisation of saturated resources, it is economically viable to terminate incoming calls[1] so that the bottleneck resource can be released. This will allow other actions to execute which can potentially bring

[1]Similar to calls termination is a concept of throttling, or the introduction of time-outs, that would cancel the execution of a request to prevent overloading.

more profit. This is a type of control that will be investigated further in our experimental study. In this context, there are several challenges involved, such as the number of action types and the different economic contracts defined in their SLAs, the impact of unknown run–time characteristics, and the nature of interference caused by others action types accessing the same shared resources.

In practice, real–time processing fails if execution is not completed within a specified time frame, or by a specified deadline, and this depends on the action type. Deadlines must always be met, regardless of system load or functionality. Attempting to optimise resources allocation in a system experiencing higher load is a typical control scenario (Buyya, Garg, and Calheiros 2011; Al-Dahoud et al. 2016; Hwang et al. 2016; Tran et al. 2017).

Optimising resources consumption in this environment requires balancing the goals of service providers and clients, e.g. the service provider may want to allocate many different types of activities, or even many tenants on the same resources, whilst the client may not be able to deliver a solid implementation. Although all functionalities following this regime must finish on time, some may end before the deadline producing a named error, which, then, should be interpreted by the client. This is softening the definition of *real–time* but it may resolve the problem of optimising resources consumption, and offer more flexibility to both parties. This is a desirable feature and it can be implemented in the framework at the API level, especially in serverless ecosystems where audit and control are possible.

Since typically real–time services must guarantee response within specified time limits, the service provider can avoid contract violations by terminating the execution before a hard deadline, just before processing the requested function but also during execution. This scenario is the main focus of this paper, where the assumption is that already allocated resources are not scaling up or down due to control decisions but can be released by terminating incoming actions requests. This is a technique widely used in practice by human administrators. Nevertheless, it has not attracted considerable attention in cloud computing, cloud–based enterprise systems or operating systems implementations yet.

Although both resources allocation and actions termination scenarios can be used in conjunction, this paper concentrates on termination only. Following a scaling–up strategy, e.g. adding extra resources, requires higher investment and introduces inertia, which may limit flexibility to deal with high instantaneous (spike) usage, and ultimately increase SLA costs. In contrast, termination control applied either before or during execution offers an attractive alternative, as it will be demonstrated in our experiments presented below.

4. A Framework for Action Termination Control

The control framework implementation sets the base for an autonomous agent that monitors the complex enterprise environment through sensors and acts upon the current run–time situation using actuators. Through appropriate control actions, it directs the system activity towards goal states that fulfil SLA requirements. Figure 1 shows the agent-based control blocks, which are able to control processes on application level, deployed into a cloud environment.

The adoption of an agent–based implementation allows the controller to collect instances of earlier control decisions and use a learning mechanism in order to refine the acquired knowledge and achieve the desired goals. Integrated into the framework are a component, named Evaluator, containing history of system states **S**, and a Control Block API, supporting actuator features, to adapt the current run–time dynamics in a way that makes the service more profitable for the service provider. The control block incorporates neural agents that are able to work independently and execute concurrently, utilising a model built based on data generated by the system under control.

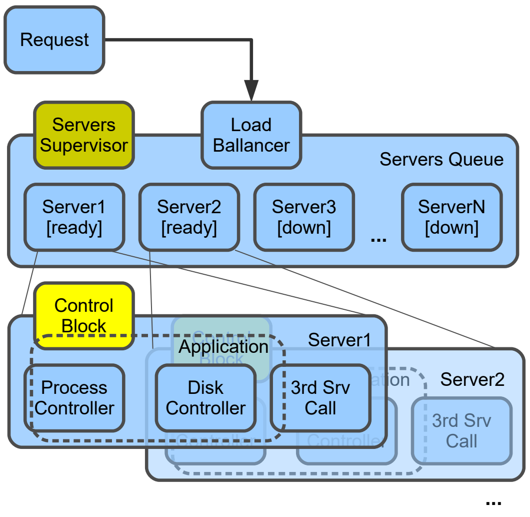


Figure 1. Deployment architecture of control blocks into application running on a cloud–based system.

This type of control introduces benefits especially in soft real–time systems, where the expectation is that the system responds within a given period of time, and this is explicitly defined as a service–level objective in the SLA definition. Examples of such functions, which better illustrate the concept, are provided in Section 6.3 below.

Special consideration is needed for termination errors, which may have appeared suddenly due to a termination actuation, so as to avoid instabilities to the rest of the client's integrated systems. The control framework enables termination errors to be handled on the client side at a price of maintaining a tighter conversation – more frequent responses on requests – with the client systems, and in this way supports higher availability across a variety of applications. This strategy may be very effective especially in micro–services architectures, where many weakly–coupled components are present. In such systems, instabilities caused by network issues, or over–utilised services, must be considered by design in the handling framework. Thus, the cost of extending error management with termination mechanisms is fairly low. Furthermore, such an approach enforces integration of time–constraints, since an error message caused by a termination event is in essence part of the contract that must be followed between interconnected services. This is consistent with event–driven distributed services deployed on serverless (Baldini et al. 2017) workers, or micro–services (Dmitry and Manfred 2014), that are getting more and more attention amongst programmers (García-Valls, Cucinotta, and Lu 2014).

Termination control, whether before or during execution, provides better potential for serverless computing to support real–time systems constraints. For example, it can provide an additional level of flexibility in the case of soft real–time systems, as mentioned above, where the code is deployed to PaaS services, or in situations where processes have types of actions with long execution, e.g. asynchronous messaging, batch systems, etc, that are of lower importance. This is further discussed below in Section 6.5.

5. Testbed Design

In this work, model–driven experimentation is used as the main approach to analyse the framework and its components, and evaluate its performance. The testbed constructed as part of the research allows profitability analysis of SaaS, FaaS or PaaS for software and enterprise cloud providers, respectively, interpreting flexible SLAs measures, actions count and SLA functions evaluations in the background of complex system dynamics, which are represented by actions, a, performance, and resources, r, consumption. Experiments for various scenarios are conducted using the testbed. The scenarios target functional and serverless computing, where an action (or a function) is an atomic portion of requested computing service that a cloud service provider manages. The cost/revenue model incorporates costs incurred for starting, or stopping, virtual machines with workers to serve requests and execute an action by measuring resources required to satisfy the call. The testbed framework contains an Experiments Runner that is responsible for presentation but also collection of data gathered in earlier experiments; the testbed design is illustrated in Figure 2. More details about the various independent components[2] and their function are provided in Sections 5.3, 5.4 and 5.5.

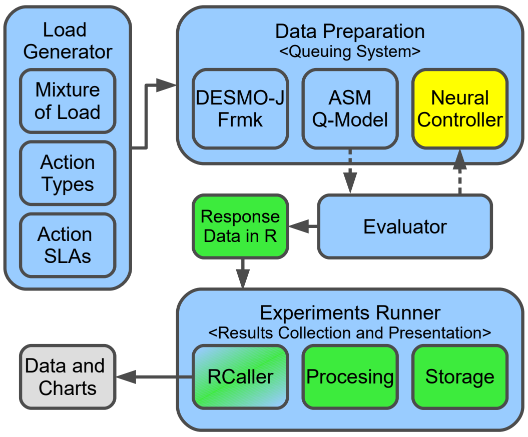
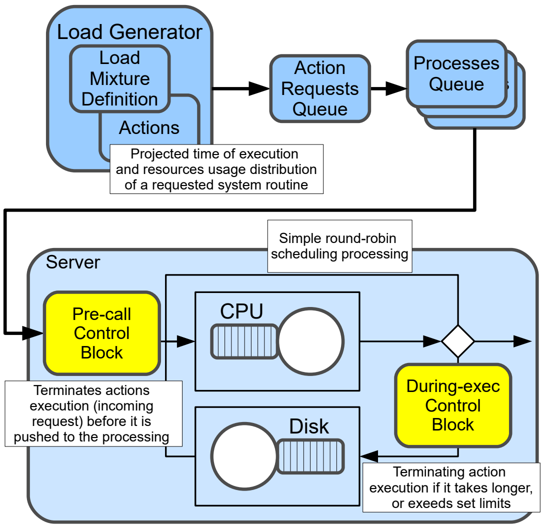


Figure 2. Testbed design contains model components, experiment preparation framework, evaluator with direct connection to controller, and data collection for visualisation and debugging.

5.1. *Discrete Event Simulation and Software Framework*

In order to model computer systems we have used a discrete–event simulation approach using the Java library DESMO–j (Page, Kreutzer, and Gehlsen 2005; Göbel et al. 2013; Tim Lechler 2014)[3]. The library has been extended with a framework wrapping the APIs, and implementing specific features of such computing systems, replicating queuing models for server, process, disk controller, action requests, and action types. We have found that lower–level discrete event modelling of the machines and scheduling approaches with DESMO–j gives better control and extensibility than other comprehensive cloud modelling frameworks like CloudSim (Calheiros et al. 2011)[4]. Moreover, the focus of this research is more on application dynamics, and therefore,



Load Generator
Load Mixture Definition
Actions
Projected time of execution and resources usage distribution of a requested system routine
Action Requests Queue
Processes Queue
Simple round-robin scheduling processing
Server
Pre-call Control Block
CPU
Disk
During-exec Control Block
Terminates actions execution (incoming request) before it is pushed to the processing
Terminating action execution if it takes longer, or exeeds set limits

Figure 3. Architecture of the ASM control system applied to a computer system model. Two different control blocks are deployed– one is responsible for pre–call and the other for during–call termination.

[2]The proposed framework allows experimentation and testing of the model, which is equipped with a neural controller directly applied to SaaS Cloud Service provisioning, in the light of revenue and cost of service usage.

[3]DESMO–j is freely available on Apache Licence version 2.0 (Apache 2004)

[4]CloudSim is able to simulate an entire cloud computational centre as well as federation of centres, offering a complete cloud model with provision of hosts, network topology, virtual–machines, and resources utilisation of CPU, memory, disks and bandwidth.

there is a stronger requirement for detailed definition of action types that reflect execution of the application functionalities, convoluted load profiles, and SLA definitions. All that has to give stable and rigorous ground for adaptive control weaved into the model. Thus, the DESMO–j library, which offers a high level of flexibility and insight into the low–level operations, appears more suitable for this task.

The framework uses the Not–weighted Round Robin Scheduling (Stallings 2014). Each process is given a fixed time to execute, called a quantum. Once a process is executed for a specified time period, it is preempted and another process executes for a given time period (Jensen, Locke, and Tokuda 1985). Each action request is transformed into a process that is decomposed into smaller chunks, which are served by resource controllers, according to the distribution set of the action type definition. Context switching is used to save states of preempted processes [5].

In this paper, two types of resources have been considered into the model: processors and disk controllers (see Figure 3). This reflects the real systems context and allows to get insight into the execution of complex action workloads. In such environments, other resources, such as network, multiple servers or memory, as well as software components, such as message queues or databases, may be interesting for implementing more sophisticated models of modern enterprise distributed systems but are not considered as essential for investigating termination control in server–based or serverless environments, which is the focus of this paper. This issue is discussed further in Section 5.8.

5.2. *Load Generator and SLA Contracts*

The load for each of the action types can be generated according to a statistical profile considering: (a) the probability distribution of arrival requests; (b) the load pattern evolution in time for the particular experiment with repetition (allowing to loop a pattern so that it repeats itself); (c) the execution time in relation to resources usage distribution. Load profiles will be explored further below, whilst some examples are provided in Section 6.1.

An important consideration under load conditions is maintaining the stability of the service, as this has implications on the performance of the systems and significant impact on usability. Thus, in practice, providers secure additional computational resources than effectively needed in order to ensure stability. Naturally, this strategy affects the costs of the provided service. Therefore, we expect that the use of reward/penalty–driven SLAs will become more widely adopted. Such SLA contracts specify precisely up to what level of execution time, the provided service is acceptable to the client, and, at the same time, profitable to the service provider. Of course execution time is a function of the required resources and of the algorithm that is being executed. The easiest way to reduce resources consumption is to optimise the application code adapting it to the specific conditions, or to the requirements of the platform on which it is running. However, since the application, software, or routine, is complex, or may have been deployed by the customer (as per SaaS on service provider system), this may not be possible. The introduction of SLA contracts gives to clients a clearer understanding as to what the highest–acceptable level of execution time is, and to service providers guidance on what resources have to be allocated in a given

[5]This is a fairly simple yet efficient scheduling algorithm. Round Robin compared to other standard approaches, like Shortest Job Next, Priority Based Scheduling, Shortest Remaining Time, or Multiple-Level Queues Scheduling, performs particularly well in conditions of overloaded systems, when jobs characteristics are unknown (Arpaci-Dusseau and Arpaci-Dusseau 2015)– an area of special consideration for this research.

point in time.

5.3. *Evaluator*

A key component of the control framework is the so–called Evaluator. Its role is to interpret run–time situations presented through monitored activity at the input, read gathered data about resources consumption, and generate appropriate output states in order to create an appropriate trajectory in the system state space. This type of outputs form an actuating signal that can be used in order to execute or terminate a service. Following the fail first, learn, adapt, and succeed feedback–loop approach[6], which is an essence of agent–environment type of interaction with delayed reward, like in reinforcement learning (Sutton 1992; Sutton and Barto 1998), the Evaluator selects systems states that formulate the training set of the Decision Block. This process selects states in operating regions where minimum and maximum SLA function boundaries are evaluated and generates the training set. Then a control action is executed and its impact is evaluated once more before the next training round. The process is iterative, so that the latest control actions can be evaluated together with results of previous actuations. Figure 4 illustrates the software components that support the data flow and the control feedback–loop, whilst Figure 6, described in the next section, provides an example of actions sequence, illustrating iterative evaluation of actions with subsequent control phases for one of our experiments. Previous control states are evaluated according to a rule–based strategy, as listed in Table 1. These rules can be seen as different strategies for selecting suitable data to train the Neural Networks–based Decision Block. The rules can form groups that define different control schemes or policies, which helps dealing with the exploration–exploitation trade–off. Hence, in our implementation the Evaluator can derive training sets effectively supporting four control schemes– these include rules combinations as shown in Table 2. Further decision making or states conflict resolution strategy is not included in the evaluation phase but is covered by the neural controller and the training process.

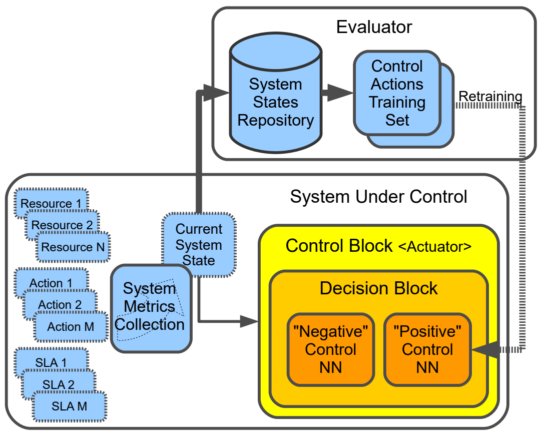


Figure 4. Architecture of the neural networks-based decision block. It combines two multilayer perceptron networks and is weaved into the system code, identifying run–time situations in system's states and executing appropriate control actions.

Both "positive" and "negative" operating conditions are important and should be "learned" by the controller. Considering "positive" control states are the main area of concern of the Evaluator. "Negative" control states operate as an additional protective measure, making sure that no control, or termination control objectives, are applied to states that were selected by mistake, producing negative outcomes. This could happen for example in a very rapidly changing environment, where observed "disturbances" may have an impact on the stability of the control. Consequently, sets of "positive" and "negative" states create operating regions in the Neural Networks-based Decision Block of the control system, as explained in Section 5.4.

[6]Similar approach to fail first, learn, adapt, and succeed is also used in Test–Driven Development.

Table 1. Rule-based Strategies

Rule	Description
Rule 1	Search for "low" –enough and "high" –enough SLA states based on total of all SLAs, T_{SLA}, see Equation 5, and label control states that are below a predefined threshold of low–mark–percentage or above a high–mark–percentage (by default 5% and 95%), as "positive" or "negative", respectively. This is similar to the strategy used in (Sikora and Magoulas 2013), which presented a different control framework that is used as a baseline.
Rule 2	Evaluate "positive" and "negative" outcome of previous control actions based on a comparison with gathered actuation decisions. Select all system states where executed termination actions had led to an SLA decrease, as measured within a given time window. In all these cases, the system's reaction on the actuation had a positive outcome so all those states are labelled as "positive". These marked systems states are passed for training the decision block of the controller, unless there is a similar system state that generated the opposite effect on the SLA values. This is an important element of decision making for conflict resolution, which is an internal part of the evaluation process. In the opposite situation, when termination control brought an increase of SLA value, and this is confirmed in other observed system states under control, these states are labelled as "negative" because the effect is counter–productive and the controller should avoid actuation in those system states.
Rule 3	Introduce stronger exploration strategy with epsilon–greedy policy (Sutton and Barto 1998; Scheffler and Young 2002) randomly selecting states, where the control actions will be applied and evaluated in the next evaluation round. System states selection is arbitrary, i.e. regardless of SLA values or reactions to control actions. By default 10% of all the collected system state points, amongst those not already selected, are added to both "positive" and "negative" sets.
Rule 4	This rule considers specifics of actions types scenarios, allowing to run independent evaluations and generate training sets per action type. Such an approach creates a multi–agent system (Busoniu, Babuska, and De Schutter 2008), where control blocks operate independently. Each evaluation cycle is guided by reward of SLA values linked to particular action types rather than the overall total of all SLAs– this will be discussed in detail in Section 5.6.

5.4. *Neural Controller*

The controller is a software component deployed into the system. It contains a decision block that is responsible for holding the generalised knowledge about earlier system states and a model of control actions/actuating signals. It is equipped with actuating logic that terminates the execution when the current system state maps to particular operating regions in the model and enforces the effective control over the incoming system requests. It is applied to the system code API (pointed to selected methods, batches, user interfaces, etc.), either weaved in with the use of Aspect Oriented Programming (AOP), or coded explicitly as per the control framework exposed to the application run–time. The controller decision block is equipped with two Neural Net-

Table 2. Control Schemes

Type	Termination On/Off	Rule	Description
0	Off	No Rule	No automatic control is applied. This type is used in simulations as a baseline for comparison purposes.
1	On	Rule 1	Search states of "low" and "high" SLA value within a given band, e.g. 5%, and allocate states for "negative" and "positive" control regions to be used for training the neural networks-based control block.
2	On	Rule 1+2	Select "low" and "high" SLA states. Evaluate marked "positive" and "negative" control states based on comparison with earlier control actions, so that recent control decisions are validated.
3	On	Rule 1+2+3	Select states of SLA extrema, evaluate earlier control decisions, and add random control points. Introduce "curiosity" effect by randomly selecting states with uniform distribution where control actions will be enforced in the next runs.
4	On	Rule 1+2+4	Select states of SLA extrema and evaluate earlier control independently per action type, considering dedicated SLA values, and execute termination targeted per each action type separately.

works (NNs), multilayer perceptrons trained with backpropagation with momentum, one working as an “expert” on “positive” and one operating on “negative” control states, as evaluated during earlier system runs.

The controller processes the current vector of the system state S and makes decisions about termination. For each received request, the system state is evaluated by the “negative” control neural network first, in order to confirm that the potential control action is not going to harm system’s behaviour. Then, if found promising, the second step is to validate this state against the mapping learned by the “positive” control neural network to make sure that it lies within control regions that introduce positive response increasing the SLA. Figure 3 illustrates the deployment of the controller on the system infrastructure. Figure 4 presents in more detail the architecture of the neural-networks based controller.

Every iteration of evaluation creates a new updated model instance, exploiting information from recently executed controller termination actions through retraining. This approach allows to apply more precise control in the next cycle, and to adapt the controller decisions to changing run–time or load characteristics of the system; this is further discussed in Section 6.

5.5. *Action Termination Actuator*

Popular approaches to implement action termination are: (a) the use of AOP– in the simplest case when the neural controller considering the current system state takes a decision to terminate a run–time process, an exception is thrown from the object that has been woven–in the code under the instrumentation. This may be a very good solution for all internal APIs, User Interface facades, and batch job actions; (b) the use of framework API integration by adding APIs in a serverless architecture (Kiran et al. 2015), and event–driven function–oriented service provision (Baldini et al. 2017).

Although in practice the controller and the termination actuator would use some part of CPU, for simplicity, the testbed model does not consider this part in the resources utilization. The typical execution time for decision–making by the neural controller is $7-9ms$, and then less than $1ms$ is needed for action termination (actuating the decision). Figure 3 illustrates the deployment and the queuing considerations of the modelled, or real, resources.

5.6. *Concurrent Multi Actions Control*

In the previous discussion, the main measure driving the focus of the controller on “positive” or “negative” operating regions in the system’s state space was the total of SLA values, T_{SLA}. That cumulative measure allows to simplify the formulation of the problem by considering it in terms of the cost and financial performance of the service provision, without the need to implement more complex Multi Objective Optimization (MOO) techniques (Zhu et al. 2016).

Intuitively, controlling each action type independently may produce better results, mainly because different actions can have far different load, performance, resources usage and most importantly SLA characteristics. Furthermore, the evaluation of earlier control actions, per action type, allows to consider specific SLA values and execute targeted termination for each action type separately. For instance, action type A can execute longer using a significant portion of resources, whilst action type B is rather shorter, called occasionally, but contracted with high SLA penalties for longer execution times. In this scenario it is desirable to terminate action A at times when there is a higher demand for many type B actions. Of course certain functionalities are not consuming much processor time nor disk but they may still have long action execution time due to their code structure. In such cases, although running multiple actions on the environment may not have direct impact on key resources consumption, some of these actions may still influence the execution of other actions. The best way to establish the effective run–time characteristics and interdependencies of the various action types is to allow them to execute, collect data representing the situation under a given load, and try to apply control so as to assess whether the anticipated operational changes are providing the expected benefits.

Figure 5 illustrates an example of load activity related to functionalities of five different actions, shown using a range of red and brown shades. It includes a case where the SLA of one of the actions and the total SLA value move to opposite directions. This is representative of situations where part of the system may be perfectly productive, and, thus, no termination control is required, although the SLAs values for most of the other actions drive the controller to take termination actions. A control scheme that has been trained using a Type–4 policy (cf. Table 2) would focus on the termination of the actions that cause negative SLA values, leaving the most productive action (in red) running.

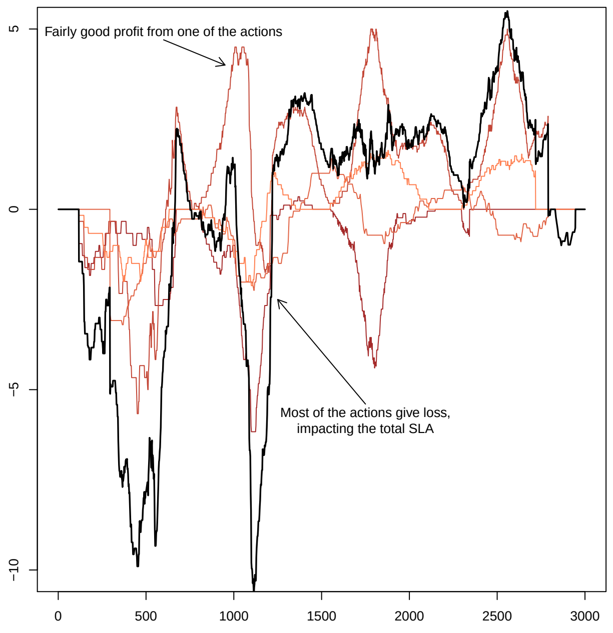


Figure 5. Sequence of SLA values for different action types depicted using a range of red and brown shades. Total SLA, line in black, is the main cumulative measure driving the control schemes described in Table 2, apart from type 4.

It is important to note that the Evaluator must review the low and high SLA values independently for each of the actions (see time–series in the bottom row of Figure 6). The direct measure of the total of all SLAs, T_{SLA}, is used for simpler control but still the importance of influence, or strength, of termination control per type of action needs to be proportionate to the total SLA measured in a given system state. This allows to weaken termination control on types of actions that do not substantially impact the total SLA[7], and vice versa, i.e. apply termination control to actions whose SLA values are matching the total SLA profile [8]. To resolve a potential conflict between the total of all SLAs, T_{SLA}, and the SLA of a particular action, the algorithm considers states with "low" and "high" SLA values for all selected types of actions scoring them based on total SLA. Subsequently, a search is applied across all action types using a specified threshold– by default this is 30% of the worst action points. As a result, control is more likely to be formulated for an action type, whose SLA values are the lowest in

[7]Weak-influencer actions types– there is no point in terminating actions whose SLA values do not cause big losses, i.e. where SLA values are not very low compared to others.

[8]For example, terminating actions with worst SLA values whilst saving well performing actions.

the system's state subspace that has been defined by the selected "low" states.

There are two potential implementations for the controller's decision block: (a) training a single neural network that can be used for all action types but when in operation, the network decodes a control decision depending on the given execution, or (b) training separate, dedicated neural networks to form the decision block (see Figure 4).

In this article, the second approach is used; hence, independent neural networks (NNs) are trained based on the same training data source. Such an approach offers more flexibility compared to implementation (a) mentioned above; for example, it allows to create an array of neural networks dedicated to different action types, if necessary. In addition, our research shows that this controller design approach usually produces better control system performance, although there are associated costs for designing different neural networks-based decision blocks, running the training procedure and an independent evaluation process. The performance of such a control approach is promising especially in cases when there is a significant difference in the run–time characteristics of the various types of actions. It is worth noticing that for scenarios where far different action types are used, i.e. batches and user interface functions, Type–1, 2 and 3 control schemes (cf. Table 2) can be used separately, so that there are isolated instances of the same framework used. Both implementation strategies can be easily applied to SaaS and PaaS models, but also to internal on–premises enterprise systems with ASM controllers.

5.7. *Terminations During Action Execution*

The previous sections considered the termination of an action when the requests are being received by the server, i.e. before any specific computation for a particular action type has been done (see Section 5.5). In this context, the controller takes action termination decisions based on expectation, as derived through training from response and usage patterns of previously observed execution instances. This aggressive approach is effective as long as the action execution times are predictable, and enforces the control system to maximise the saving of resources used by the action. Although this can be especially attractive in soft real-time systems where the code is deployed to PaaS/SaaS services, in scenarios tested where the execution times are quite constant under non–overloaded system conditions, little overall benefit was observed (see Fig 11). However, the more unpredictable the execution time is, the more this method is expected to outperform simple termination control working before action execution only.

It is worth mentioning that the longer an action runs, the more resources are consumed, so terminating the action at the end provides lower value since computational power has been consumed for processing and execution already. Thus, often, it is much more beneficial to decide not to produce a response in a given time line than to produce it with delay. This is especially true for functions that are directly exposed to users or human operators, and may cause stress to the user, such as when users make successive clicks on the same button. For example, if a service is slower than usual, users tend to confirm the requested action by clicking a button again and again. In most cases, the first request has been processed correctly but there are just not enough resources to execute it, or the performance might have been altered. Consequently, right after the second request comes to the server, there is no point in executing the first one[9].

[9]A potential solution to this problem in practice is to block the function from being clicked again, or

PaaS/SaaS/FaaS systems may be equipped with a control API specifically tackling both action termination schemes following the application code structure, where a specific control block type is weaved into different types of actions for best performance. For example, short and predictable calls, like user–interface or web–services, controlled by pre–execution control, and longer less predicable ones, like batch processes or asynchronous messaging, controlled during action execution by terminator actuators.

In engineering practice, instrumentation required by this approach can be difficult to implement or may be using significant resources due to the repeatable additional checks that should be performed during action execution. In many cases, this technique can be simply too intrusive to be effective. However, it can be appropriate for batch based systems or for selected longer actions. Further extensions could allow the controller to detect execution times or distribution, predictably, and choose which control scheme is best for a given action type under specific run–time conditions.

5.8. *Simplifications and limitations*

Enterprise systems are complex in nature so in order to simplify the experimental data collection and analysis two types of resources are considered in this paper, namely CPU and Disk. Of course, other types such as network, memory, virtualization layer aspects, or calls to other services (introducing idle time and latencies) could be added in a similar way to the already implemented resources. Moreover the computational effort related to the collection of the metrics, storage, evaluation and neural–control block training were not included in the model. There are two reasons for that: (a) the monitoring facilities and the Evaluator can be isolated from the system under control, having minimal effect on the system performance; (b) investigating engineering details of the neural network training was not part of the research[10].

Although extending the framework to deal with many servers would give simulation results closer to real–world data centres, it is not considered essential for action termination control, which involves single server run–time, node instance or container and interconnections with other nodes do not impact the core observations. Connectivity to other servers, applications or systems, with waiting effects and networks utilisation considerations were also left for future work in order to keep the range of the experiments more focused on the potential of action termination. Therefore in this study a single CPU and Disk queue per server has been chosen in order to simplify the set up, and aspects, such as multi–threading, virtualization overhead, multi–node coordination that could introduce further complexity and obfuscate the observations were not considered. In the future, the model could be easily extended with more CPUs. Hhowever, it is expected that any change of characteristics would be close to linear due to the used scheduling algorithms.

As discussed in Section 3, the execution time is very important for profitability of PaaS/SaaS/FaaS running in tight SLA contracts. However, there are also factors such as cost of equipment and energy usage that can be factored into the aggregate cost values, which are added to the SLA functions associated with action types. The paper

simply generate the fastest action possible. Ideally, the entire end–user–performance execution time should be less than 1 second. That is considered to be the limit for the user's flow of thought to stay uninterrupted, while 10 seconds is the limit for keeping user's attention focused on a human–machine dialogue (Jakob 1993; Miller 1968).

[10]It takes around 1–5 seconds to train a neural–control block per action type in a 2000–time steps experiment on a single i7-6500 CPU@2.59GHz with use of around 11000 system metrics collected.

does not focus on those dimensions although the framework can incorporate those values as constants in the SLA function definitions.

6. Experiments

A set of experiments is presented in this section to evaluate the proposed control framework under various conditions and demonstrate its adaptiveness and effectiveness to generate control actions that optimise financial performance in ASM environments. Each of the experiments is executed in the context of a list of action types with specific load patterns, distributions of resources usage and SLA function definitions. All these elements create a complex environment even in a fairly simple server–disk control scenario.

Section 6.1 elaborates more on the mixture of components used to define the load profiles and the load parametrisation. The discussion of the experiments conducted starts from simpler cases and then progresses to more complex and elaborate scenarios. The first experiment in Section 6.2 demonstrates operation in a simple scenario, which is first executed without control and then with control applied on the system in order to compare changes in resources usage and SLA function optimisation between the two configurations. Then, in Sections 6.3 and 6.4 the simulations exhibit more adaptivity and illustrate financial performance details providing a comparative evaluation for different types of control schemes (cf. Table 2) under various load profiles.

6.1. *Load Parametrization*

Different types of actions are realised by application functionalities that utilise CPU or Disk. The load is specified by three factors: (a) load intensity pattern, multiplied by (b) the quantity of incoming requests to perform the action, generated by a probability density function, and (c) an execution time distribution.

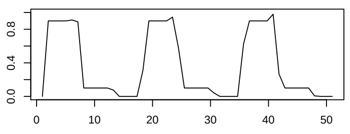
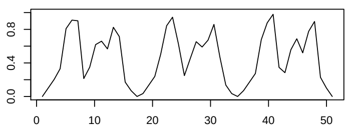
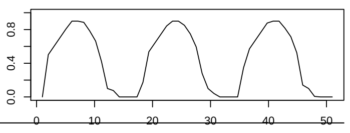
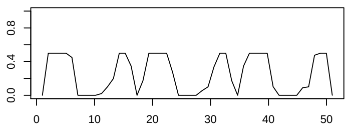
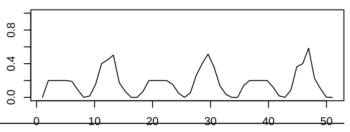
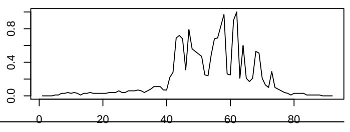
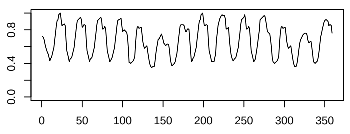
For example, action type A3IO2 definition, in Tables 3 and 4, provides the relevant statistics for a functionality that is rather Disk–bound (90%), utilizing CPU only 10% of the execution time, and has the following execution distribution parameters: exponential distribution with arrival rate $Exp(\lambda = 10)$; action execution time defined by a normal distribution $\mathcal{N}(\mu = 5, \sigma^2 = 0.1)$; a variable load intensity set as a request probability pattern repeated 3 times within the time frame of a single experiment[11].

```
ActionType actionA3IO2 =
  new ActionType("A3IO2",
    new LoadPattern(
      // arrival density distribution
      new LoadDistExponential("DExp", 10),
        // load pattern
        new double[]{  // intensity
            0.0, 0.2, 0.2, 0.2, 0.2, 0.2,
            0.1, 0.0, 0.0, 0.1, 0.4, 0.4,
            0.6, 0.2, 0.1, 0.0, 0.0},
            3),          // LM repetition
    // execution time distribution
    new LoadDistNormal("DNorm", 1, 0.1),
    // resources utilization distribution
    new ResourcesUsage[]{
      new ResourcesUsage(
          ProcessingType.CPU, 0.1),
      new ResourcesUsage(
          ProcessingType.Disk, 0.9) },
    new SLA());
```

Table 3. Example of action type: Java code snippet of action A3IO2, its load profile, and load distributions

[11]The load profile offers a precise way of configuring variability in the expected intensity of the frequency of incoming requests for a particular action. It helps defining different test scenarios, highlighting cases such as actions interference when using the same resource, higher load applied temporarily to observe the effects of spikes in resources consumption, or short reoccurring load changes to analyse the impact of a "delay" factor on the strength of ASM signal deconvolution (Sikora and Magoulas 2014, 2015).

Table 4. Action Types used in the experiments

Action Type	Quantity of incoming calls[a]	Execution time distribution[b]	Resources Usage CPU, Disk(IO)	Load Intensity Pattern[c]
A1	$\mathcal{N}(\mu = l, \sigma^2 = 0.1)$	$\mathcal{N}(\mu = 4.2, \sigma^2 = 0.1)$	CPU=100%, Disk=0%	
	This action type imitates a business driven function that utilizes only CPU. Load pattern of request calls and execution time are following a normal distribution with average interval l and μ equal to 4.2 .			
A2B	$\mathcal{N}(\mu = l, \sigma^2 = 0.1)$	$\mathcal{N}(\mu = 2.0, \sigma^2 = 0.1)$	CPU=80%, Disk=20%	
	This action type models another business related function, where request calls frequency is constant and execution time follows a normal distribution with average set to 2.0. This action uses 20% of Disk and 80% of CPU to compute the results.			
AOS	$\mathcal{N}(\mu = 10, \sigma^2 = 0.1)$	$\mathcal{N}(\mu = 0.5, \sigma^2 = 0.1)$	CPU=90%, Disk=10%	
	This action type was introduced to mimic load coming from operating system activities. Actions are very short and it is rather CPU driven.			
A3IO	$\exp(\lambda = 5)$	$\mathcal{N}(\mu = 3.0, \sigma^2 = 0.1)$	CPU=1%, Disk=99%	
	This action type computes responses by mainly using disk resources– only 1% of execution time is used by CPU. The frequency of calls is defined by an exponential distribution, so it is less predictable than other actions shown above.			
A3IO2	$\exp(\lambda = 10)$	$\mathcal{N}(\mu = 1.0, \sigma^2 = 0.1)$	CPU=10%, Disk=90%	
	Similar to A3IO but calls are 3x shorter, two times less frequent, and is using less disk time. The code snippet of Table 3 provides implementation details.			
WC1, ..., WC12	$\mathcal{N}(\mu = l, \sigma^2 = 0.1)$	$\mathcal{N}(\mu = 4.2, \sigma^2 = 0.1)$	CPU=90%, Disk=10%	
	This set of action types imitates a business driven function, which utilizes mainly CPU. Request calls and execution time are the same as in a sequence of A1, A2B, AOS.			
W.En, W.De, W.Ja, W.Es	$\mathcal{N}(\mu = l, \sigma^2 = 0.1)$	$\mathcal{N}(\mu = 4.2, \sigma^2 = 0.1)$	CPU=90%, Disk=10%	
	This action type imitates a business driven function, which utilizes mainly CPU. Request calls and execution time are same as in A1.			

[a]Quantity of incoming calls defines load produced by requests following a probability distribution: normal, $\mathcal{N}(\mu, \sigma^2)$, or exponential $\exp(\lambda)$.

[b]Execution time of a single action call defined by normal distribution.

[c]Load intensity patterns depicted in the table have been applied for Load Multiplier (LM) equal to 3.

Examples of SLA functions are provided in Table 5. The SLA function is formulated according to an agreement made between parties; for example, it can be a function of service quality provided that depends on execution times, for instance: "I as a service provider expect payment (positive: revenue/profit) for short executions, and pay back penalties for the longer executions and potentially for termination actions".

Additional examples of action types that are used in the experiments are shown in Table 4, while Table 6 describes experimental scenarios and defines SLA functions. There is a mixture of synthetic– and real– workloads. The later represent loads captured on the 1989 World Cup website and on Wikipedia in October 2017, and have been used in the past by adaptive admission control researchers Xiong et al. (2011) and Ferrer et al. (2012).

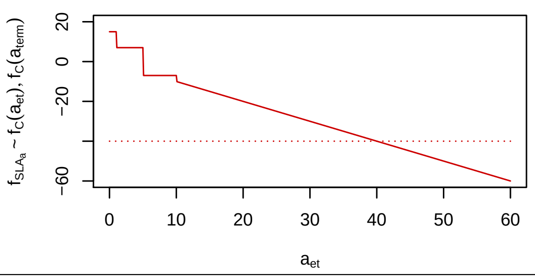
Table 5. Example of execution time SLA cost functions; code and visualisation

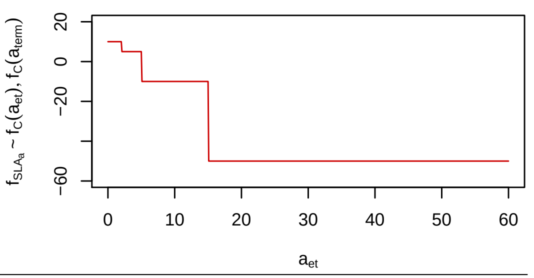
SLA function with termination penalty

```
// such SLA costs are common
// for Soft Real-Time Systems
ExecutionTimeCostsSLA sla
  = new ExecutionTimeWithTermCostsSLA(){
    public double price() {
      // linear (not constant) penalty
      if  (getServiceTime() >  10)
        return -1*getServiceTime();
      else if (getServiceTime() >  5)
        return -7;
      // sub-second call, good price
      else if (getServiceTime() <= 1)
        return 15;
      // normal positive price
      else if (getServiceTime() <= 5)
        return 7;
      else return 1;
    }
    public double terminationPenalty() {
      return -40;
    }
  };
```

SLA function without termination penalty

```
//
//
ExecutionTimeCostsSLA sla2 =
  new ExecutionTimeCostsSLA() {
    public double price() {
      // high penalty
      if (getServiceTime() >  10)
        return -50;
      else if (getServiceTime() >  5)
        return -10;
      // sub-second call
      else if (getServiceTime() <= 1)
        return 10;
      // normal positive prize
      else if (getServiceTime() <= 5)
        return 5;
      else return 1;
    }
    // no termination penalty
  };
```





6.2. *Experiment 1: Comparing operation with and without the Neural Control Block*

This experiment aims to give an overview of the operation of the controller and demonstrate its adaptivity and effectiveness, starting from the system operating without control and then, as the load profile and SLA function remain the same, the controller is activated. The control system performance is evaluated and gradually optimised through controller retraining.

Figure 6 presents time sequences for input activity, impact on resources and financial performance, in terms of SLA values for an action type. The same load profile is applied and the system operates without control (on the left) and with termination control (on the right). Actions executions are presented in shades of red, whilst resources utilizations are in blue and black. The bottom row presents the sequence of the total SLA values for the experiment. The total SLA includes the costs of overrunning actions/SLA violations, termination penalties, but also the revenue/rewards for actions serviced in a normative time. Thus, the higher these SLA values are, the more profitable the scenario is. Termination control is executed after the first evaluation that was called at the 500th time slot. Later, the controller cancels the selected action requests to free up resources for another action and in effect the total SLA is improved. Count of terminations is shown in the sequence on the 6th row.The response diagrams on the 4th and 5th row illustrate utilisation of disk and process queue (CPU usage by actions computations). On the right hand side plots, the resources are clearly less utilised due to the lower number of serviced actions, as shown by the sequences on the 2nd and the 3rd row.

The mean arrival time and quantity of different action types used in this experiment expose the system to load, which is around 4 times bigger than the system is able to handle; that is around 4 times above the saturation threshold. Although the system load is generally more CPU–bound, disk is significantly overused as well. Thus the controller attempts to reduce the load by terminating some of the actions. Naturally, both the termination penalty and the penalty for longer execution times are considered (see the example of SLA function definition in Table 5). We can see that the controller drives the system into "profitable" states just after the first training cycle, where SLA function values get significantly higher around the 750th time slot. The training phase is repeated every 500 time steps, and it should be noted that not all actions are terminated, whilst resources consumption is reduced soon after the first training cycle. The training set used each time consists of "positive" and "negative" states identified by the Evaluator, as described in Section 5.3. An example is shown in the scatter plot matrix of Figure 7, where positive and negative states are denoted by Control Mark "Right" and "Wrong", respectively. This figure shows all the key dimensions of the system states in pairs illustrating their relations. All system states presented have been processed by the Evaluator and some of them were labelled with "positive" and "negative" control marks. Note how the spikes in Disk Queue impact total SLA, lowering the values due to high execution times (see red points). This is simply due to

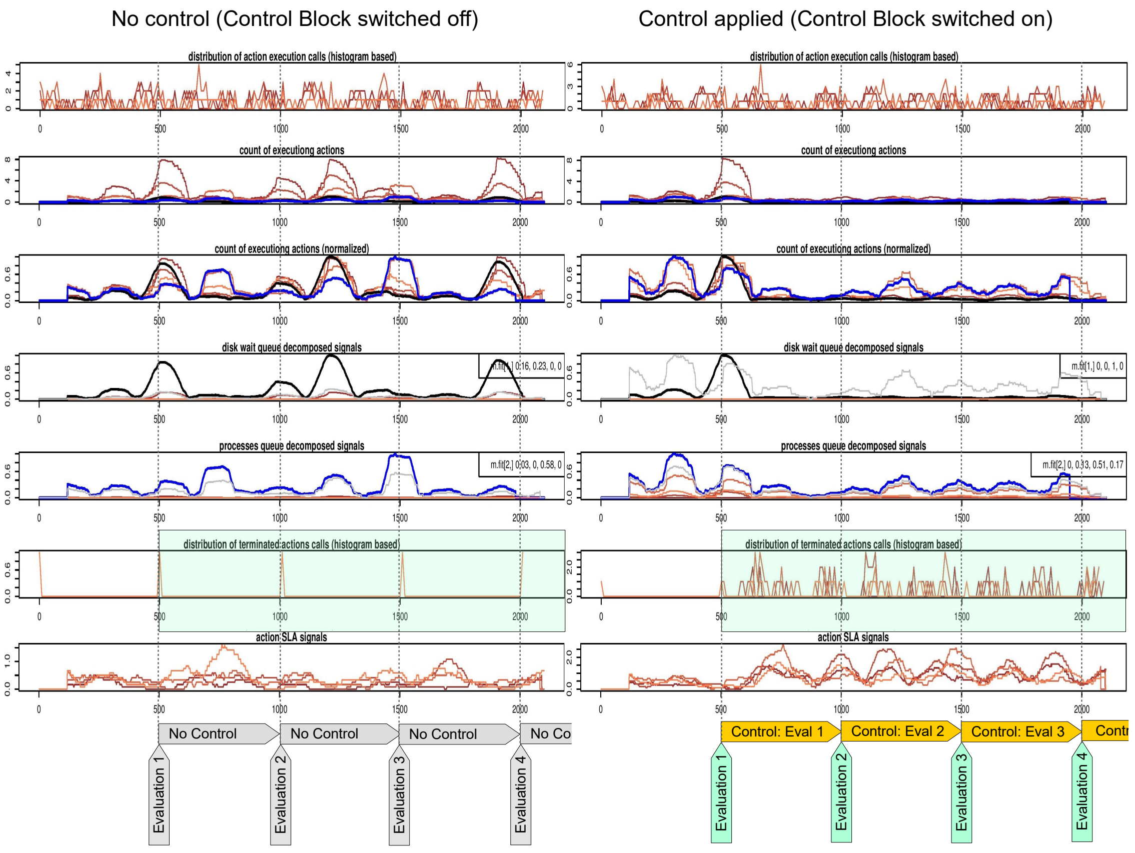


Figure 6. Metrics time sequence and system responses for operation without controller, on the left side, and with neural control block that uses Type–2 control scheme (cf. Table 2), on the right side. In both cases, the experiment was executed under the load pattern and SLA function of scenario 4 (cf. Table 6).

the fact that all actions were executing simultaneously saturating the CPU. After the 500th time slot, when the controller is engaged for the first time, resources queues are reduced substantially. High utilisation of the resources is still allowed, offering high computational power for service provisioning, but overloading the system and very long executions are avoided. The number of states with high SLA values, violet points, after 500 time steps in the scatter plot of the pair CPU utilization/Time (column 1, row 3 of the matrix) indicate revenue generation and more effective utilisation of resources, and thus a much more profitable operation. The same effect can be seen in Figure 6, where the time sequence in row 3, on the right side, shows a huge spike of concurrently executing actions between the 480th and the 600th time slot. Although the same load was applied repeatedly, i.e. 4 times, during the run, such a big accumulation of waiting actions was not repeated, which is attributed to the application of termination control. In general, after the controller takes over, financial performance, represented by the values of the total SLA, gets better, as exhibited in Figure 7 (see plot in the first column, last row of the scatter plot matrix). Due to the changing nature of the load profile, saturation issues occurring quickly and system inertia, resources utilisation is not highly correlated with SLA value changes, but directly relates to the queue length that far better corresponds with high SLA function values observed.

Dimensions representing resources queues appear to be sources of rich information for the system entering saturation. Thus, control actuators monitoring resources usage and queues are able to react to the changing conditions and execute effective control actions.

6.3. *Experiment 2: Adaptivity to Load Profile and Financial Performance*

The testbed allows to perform many experiments in batch mode exploring various problem dimensions such as: comparing types of control schemes (cf. Table 2), verifying system performance under changing load patterns and different values for the mean time of arrival, repetitions with different random generator seeds, and, lastly, testing various model parameters using a set of scenarios with different action types under specific load patterns and SLA definitions; see Tables 4 and 6 respectively.

In this section, we present experiments to test the above conditions and discuss their results. Eight different system scenarios are discussed below, as summarised in Table 6. The scenarios differ in the way their SLA contracts are defined. The scenarios corpus starts from the most aggressive SLAs– mainly driven by operations monetization perspective– where the control system has the most challenging situation to manage from both financial and load perspectives. The first three scenarios are penalty driven, and as a result the service provider will incur penalties for action executions times that are longer than agreed. In the last three scenarios no penalties are included but the reward/revenue for longer calls is zero; thus, in the case of a slow environment the service provider will be operating the requests for computation service with minimal revenue generated, which will incur internal costs for the resources utilised.

In the first round, we compare the system operating without control against the first two of the termination control schemes presented in Table 2. Tests are conducted under eight different load scenarios and SLA function definitions, as presented in Table 6. This test examines termination control before execution only, i.e., just before the request for the functionality to be executed is processed. The results of the experiment are exhibited in Figure 8. The plots illustrates the impact of the control

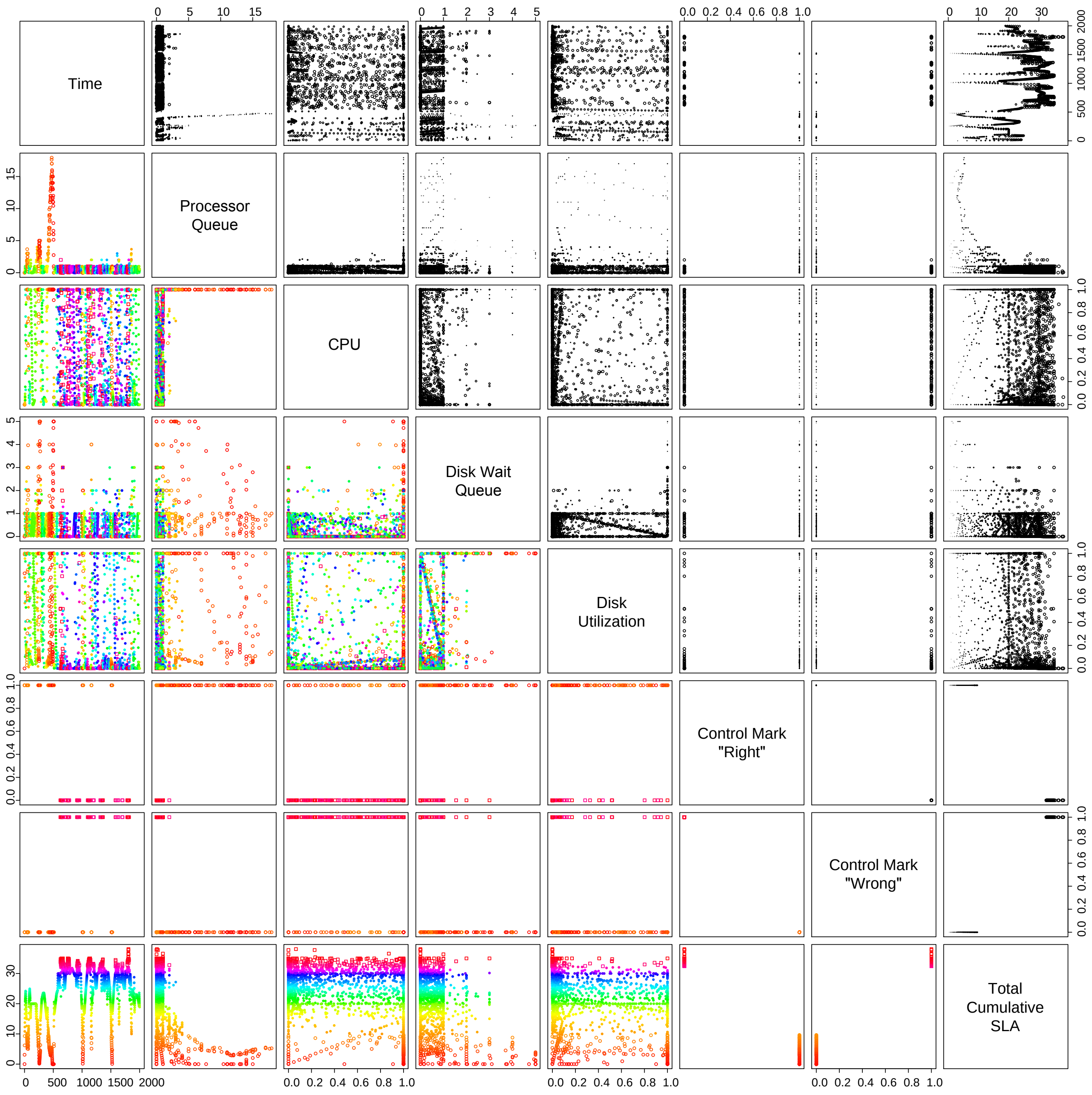
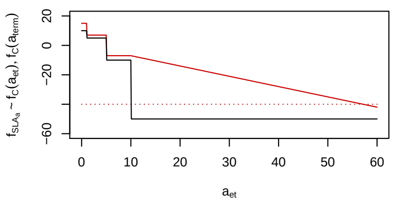
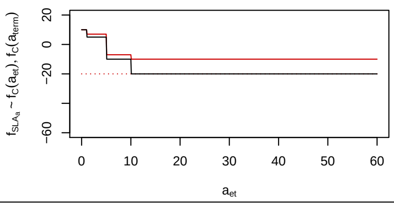
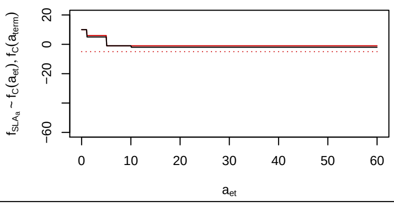
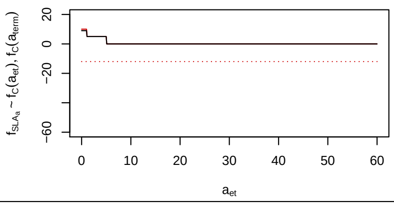
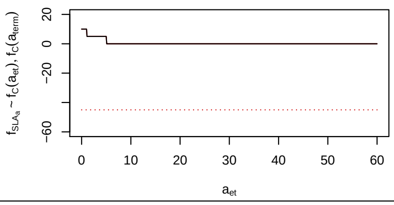
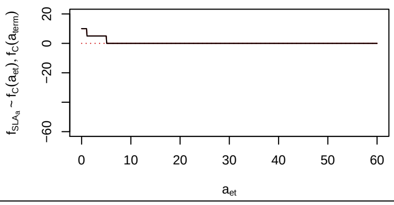
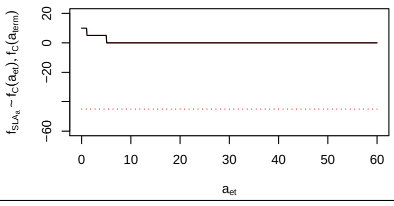
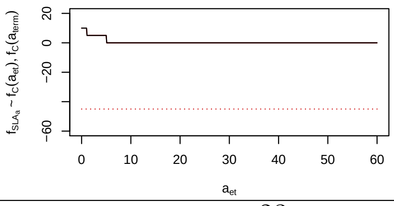


Figure 7. Scatter plot matrix of the most important dimensions of the system states collected during the last evaluation iteration in Experiment–1. The neural control block uses a Type–2 control scheme (cf. Table 2), whilst the load pattern and SLA function follow scenario 4 (cf. Table 6). The dimensions named in the diagonal, starting from top–left, are: Time, Processor Queue, Processor Utilization (CPU), Disk Queue, Disk Utilization, "Positive" and "Negative" Control States (denoted by Control Mark "Right" and "Wrong", respectively), Total Cumulative SLA. The system states have been processed by the Evaluator (cf. Section 5.3) and have been labelled with "positive" and "negative" control marks used later for neural networks training. In the plots below the diagonal, a rainbow colour scale is used, starting from red for low values and ending with pink-violet for high values. The size of a point in the scatter plots above the diagonal represent its SLA function value, so that system states with lower SLA value are smaller and less visible.

Table 6. Examples of Load Scenarios and SLA function definitions

Scenario	**SLA Function**	**Description, Actions Types (SLAs), Load Pattern, LM**[a]
Scenario 0: This is a very challenging control scenario that employs a very aggressive penalty scheme for an overloaded system.		5 action types, namely A1, A2B, AOS, A3IO, A3IO2 with very aggressive SLAs, where the system is profitable only for calls shorter than 5 time stamps. For execution times longer than 10 stamps, the penalty is linearly increasing, whilst the termination penalty remains very high, equal to -40 for each termination decision. The code for this SLA is shown in Table 5. Effectively, termination control makes sense only for very long execution times. Actions load pattern is described in Table 4 and load is applied for LM=4.
Scenario 1: A slightly less aggressive penalty scheme for an overloaded system.		4 action types, namely A1, A2B, A3IO, AOS. Execution time less than or equal to 5 time steps generates a +7 award, but longer executions produce a penalty of -7, while the ones that are longer than 10 time stamps produce a penalty of -10. The termination penalty is -20 (two times smaller than in Scenario 0). Actions load pattern is described in Table 4 and LM=4.
Scenario 2: A scenario with low penalties for long execution times and termination penalty, which is cheaper than long execution time penalty.		4 action types, A1, A2B, AOS, A3IO, are used. There are low penalties for long executions: only -1 penalty for execution time longer than 5 time steps, also reasonably low penalty, -10, for each action termination, while for AOS and A1, termination penalty is only -5. Actions load as described in Table 4 and LM=4.
Scenario 3: This scenario has no penalties for long execution calls, so that the control can focus on revenue optimization only.		4 action types are used: A1, A2B, AOS, A3IO. There are no penalties for long executions but also zero award. There is a substantial penalty, -12, for termination of A1, A3 and AOS, but also awards for shorter calls; that is +10 for calls shorter than 1 time slot and +5 for shorter that 5. Action load, see Tab. 4, LM=4.
Scenario 4: This scenario demonstrates the effect of termination control on overloaded system without penalties.		4 action types are executed: A1, A2B, AOS, A3IO. No penalties for longer executions are applied but there is substantial penalty for termination. Shorter calls are rewarded, enabling active control. This scenario uses a load pattern based on Wolfs Sunspot Numbers Hathaway, Wilson, and Reichmann (2002). Actions load is standard, except for AOS that follows a pattern of annual sunspots[b], LM=3.
Scenario 5: This is a scenario that demonstrates the effect of termination control on overloaded system without any penalties.		4 action types are executed: A1, A2B, AOS, A3IO. There is no penalty for termination nor for longer execution times (SLA function has no negative values). Still, shorter calls are rewarded which drives the evaluations and the controller optimization direction. Details of the actions load pattern are in Table 4; load is applied for LM=2.
Scenario 6: The "World Cup 1998" scenario, described by Arlitt and Jin (2000), shows a three month period of quite bursty traffic and aperiodic workload.		12 actions of type A1, A2B, AOS, A3IO are executed, representing 1% of the most popular pages of the website that received 75% of all requests (Arlitt and Jin 1998). There are no penalties for longer executions and termination, but still higher reward for shorter calls. The load pattern distinguishes this scenario from the rest, as shown in Table 4, and it is applied for LM=1.
Scenario 7: This scenario uses Wikipedia workload in October 2007 (Urdaneta, Pierre, and Van Steen 2009) demonstrating operation under cyclic traffic.		There are 4 action types, namely W.En, W.De, W.Ja, and W.Es, responsible for 63.18% of the total load to Wikipedia in October 2017. There are no penalties for longer executions and termination, but still reward for shorter calls is higher. The load pattern is distinctive, shown in Table 4. Although LM=1, the pattern is periodic.

[a]LM = Load Pattern Multiplier is a parameter that multiplies the sequence of a base entry load profile, so that repetitive sequences of system states can be tested. An LM equal to 4 correlates very well with the frequency of the Evaluator cycles (see Section 5.3), which executes every 500 time steps. In all experiments, a total of 2100 time steps is used (see Section 6.2).

[b]"Annual sunspot relative number 1936-1972" and "Wolfs Sunspot Numbers 1936-1972", source `https://datamarket.com/data/set/22nu/annual-sunspot-relative-number-1936-1972#!ds=22nu&display=line`

scheme on financial performance, represented by the Total Cumulative value of SLA units measured for each of the monitored system states T_{SLA}. As the time between calls increases, represented by the mean arrival time l that is defined according to a probability distribution for each scenario, the lower the frequency of calls gets, and, effectively, the load that the system needs to consume. Table 7 shows a comparison of financial performance increase $P \sim f_{SLA}$ in the test runs, per scenario and load l, using cumulative figures of T_{SLA} collected during these simulations.

Depending on the functionalities produced by the different actions in the scenario, the control system reacts differently to the load pattern it is exposed to taking into account the SLA function's definition. In effect, the controller adapts to the particular load conditions. In all scenarios, the controller improves the revenue generated by the provided service when the system is under higher load, as denoted by the lower mean arrival time l values. Furthermore, Type–2 control scheme performs better than Type–1 in most of the cases. In scenarios 2, 4 and 5 the control system was able to improve the situation regardless of the load that the system was exposed to. Note that even for action types, whose SLA function definition does not include penalties set explicitly (i.e., no negative SLA values for longer executions) but there is a higher reward for shorter execution times (see for example scenarios 3, 4, and 5), the controller drives the system to states that generate revenues. Also, the controller tends to terminate the longer actions, although these may be cheap in terms of associated penalty. This is because it considers that a cheaper action that is consuming resources is not economically viable to operate, even if termination incurs additional cost. This is a key finding for the monetization of computing services such as PaaS/FaaS/SaaS equipped with this type of control. The controller shapes the run–time situation in order to achieve better conditions for running many types of actions under the same shared pools of resources. It favours cheaper executing actions that generate revenue over more resource expensive actions and their revenue, taking into consideration the potential costs of termination. Intuitively, such control could be most efficient when system's utilisation is high. In that case, releasing resources required to compute competing for access to resources actions benefits high–revenue actions. Therefore, it is not surprising that the best performance of the termination control in the experiments was in the heavily loaded system cases. The highest efficiency was noted under scenario 4 and 5, for arrival time lower than 4.0. In scenario 5, Type–1 and Type–2 control schemes were able to improve the system's operation value by 30–52% and 13–33%, respectively. This is clearly due to the less restrictive SLA and the lack of penalty for termination. Still in scenario 4, where additional cost for termination is considered, the improvement is significant– in the range of 15–35% and 8–25% for Type–1 and Type–2 control, respectively. Note that in scenario 4 there is still good improvement after applying control in the lower load as well. This is possible due to the relaxed nature of SLA, and the lack of penalties for termination. Scenario 5 has quite good overall improvement that is gradually reduced– the higher the load (l value gets low), the worse the total profit is. Interestingly, the operation performance is much greater than the intuitively expected increase of the revenue generation due to higher load (more actions to account) but impeded by the overused system resources, see the distribution shown on the violin charts. Very cheap termination in scenario 3 shows fairly good improvement, 5–12%, in cases of higher load, for arrival time lower than 4. The lower the load is, the fewer the chances to reduce the usage of resources by termination actions, so the operational improvement is gradually lowering but it is still better than no–control. This behaviour is repeated even when lower load is applied, e.g. mean arrival time gets equal to 6, where the system was profitable for all system

Table 7. Financial performance increase $P \sim f_{SLA}$ between control (Types 1 and 2) and no–control (Type–0), in independent test runs per scenario and load, in terms of T_{SLA} values measured during simulations execution (cf. Figure 8)

		ΔT_{SLA} [%]							
Control Type	Scenario	3.0	3.5	4.0	4.5	5.0	5.5	6.0	Total
1 to 0	0	5.63	-18.87	-25.52	-5.14	-6.18	-22.04	-43.66	-16.54
	1	0.60	2.72	3.64	-23.69	-16.89	-16.16	-29.22	-11.28
	2	9.07	9.09	28.70	21.85	14.77	15.02	7.71	15.17
	3	12.30	5.62	0.55	-8.14	-3.31	3.18	-16.21	-0.85
	4	34.64	30.50	15.36	14.46	22.57	20.41	14.69	21.80
	5	47.03	52.26	8.09	9.67	3.64	11.35	10.50	20.36
	6	96.41	75.14	33.95	25.70	54.86	64.44	62.26	58.97
	7	146.41	58.33	38.88	34.74	55.07	56.78	59.86	64.30
2 to 0	0	-1.89	-14.48	-31.35	-2.67	5.39	-19.58	-25.46	-12.86
	1	0.52	13.87	6.84	-22.51	13.31	-21.78	-36.02	-6.54
	2	5.84	66.99	22.75	36.18	18.82	3.42	7.34	23.05
	3	12.89	3.32	-4.26	1.61	-13.41	-4.79	-13.57	-2.60
	4	8.54	24.72	13.78	10.61	23.31	12.99	11.21	15.02
	5	16.81	33.40	13.63	16.82	6.64	14.03	9.44	15.82
	6	98.12	22.87	27.51	21.45	57.38	45.18	54.27	46.68
	7	146.41	61.61	34.23	31.94	52.71	55.78	58.36	63.01

states monitored.

Performance in scenarios 0, 1, 2, where SLAs contain penalties for longer termination, is lower than in scenarios 3–5, but still there are improvements made especially for higher loads. The financial performance in these cases is mainly impacted by the fact that SLAs contain substantial penalties for longer execution times that widen the band of profitability distribution and reduce the scale of the potential improvements that can be made on the system. Of course high prices for termination additionally impact the chances to improve the operation. The most difficult scenarios, 0 and 1, show practically optimisation of the loss of a service provider that is exposed to extremely challenging contracted SLA conditions. The control system was able to improve the situation by 5% only under very high load, where mean time of arrival is 3.0.

It is important to note that in the case of difficult to control scenarios where SLAs are aggressive in terms of penalties, like in scenario 0 and 1, the lower load may carry risks such as introducing degradation of the operational performance and even greater financial losses.

6.4. *Experiment 3: Resources Utilisation and Financial Performance*

The focus of this experiment is to evaluate how the controller operates under different load conditions when the executing actions require resources to be consumed, and the impact of the control scheme on resources utilisation and cost optimization and revenue generation.

It is clear that action execution times will heavily depend on the utilization of resources; thus, high SLA values are likely to be found in the situations where the system is reaching a saturation state. From this point on, the system will be queueing the requests and SLA values will grow significantly. Figure 9 show this effect as observed in the experiments, while Figure 10 illustrates the states' SLA values. Each row in these figures presents several independent runs in one scenario, illustrating the distribution of system states for process queue, disk wait queue and total cumulative SLA for each one the three control schemes (Types 0, 1, and 2) tested under various load patterns.

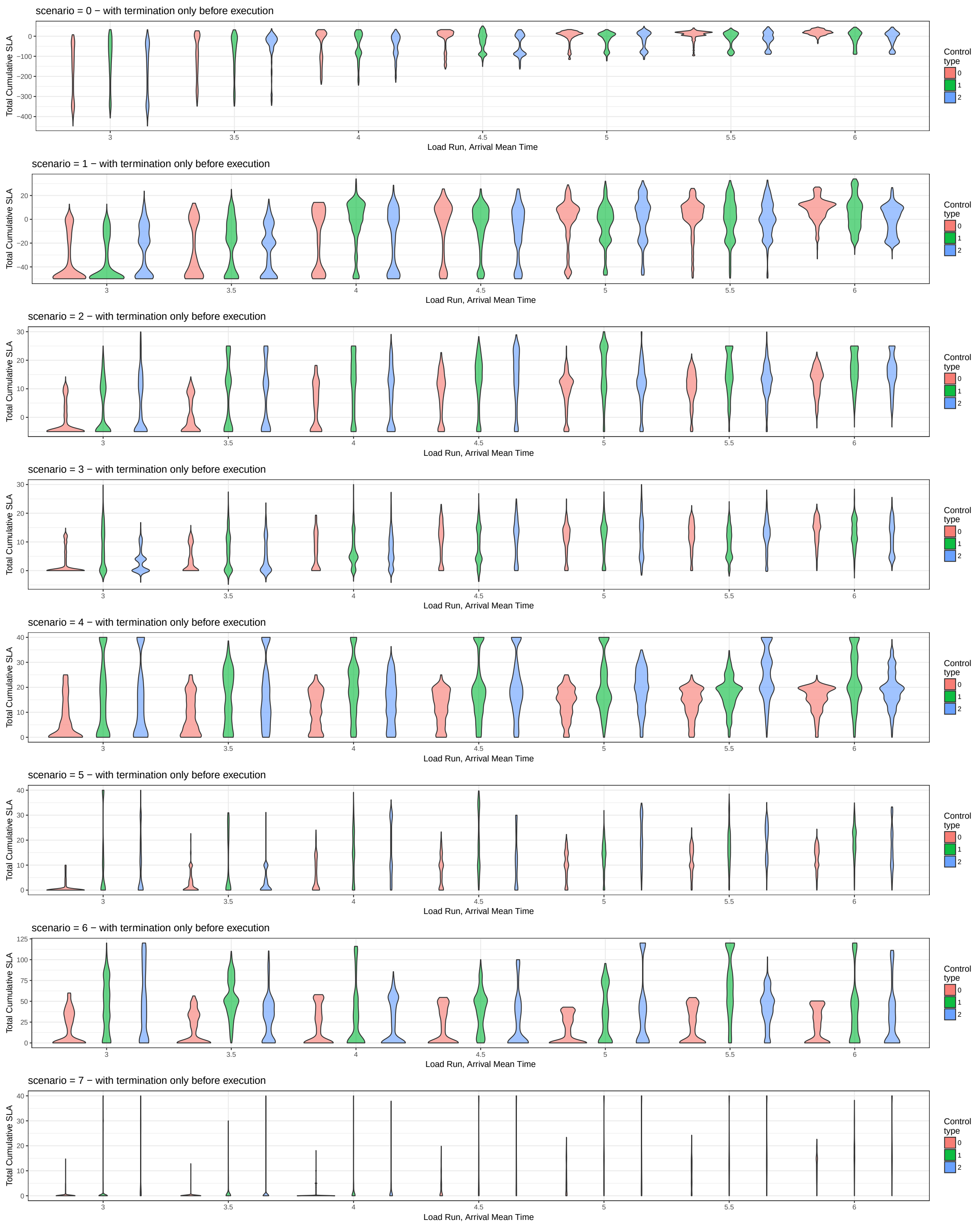


Figure 8. Comparative results for operation without control and with two of the termination control schemes, namely Type–1 and Type–2 (cf. Table 2), which apply termination control only before action execution. The diagram presents the distribution of system states as a function of load run. The focus is on financial performance, represented by the Total Cumulative value of SLA units measured for each of the monitored system states. SLA function definitions consider costs/permissions and revenue generated, and the higher the values reached in the vertical axis, the more profitable a control scheme is for the particular scenario.

Similarly to the previous tests, to simplify the results presentation[12], this one also examines termination before execution only.

One can see that in scenario 0 the CPU is massively over–utilised, e.g. there is a large process queue length with up to 80 tasks awaiting. For other scenarios, namely 1, 2, 3 and 4, where SLA functions get gradually less restrictive (e.g. especially when penalties for actions termination are lower), the controller is able to gradually free up more and more resources in order to give an improved SLA response. The only exception to this "rule" is the last scenario, number 5, where the load–pattern multiplier–LM (cf.Table 6) is too low to allow the controller to experience more repetitive sequences of systems states, and thus to reinforce the notion of "good"/"bad" control states. Effectively, there are fewer high quality control actions executed, and only the lack of penalties compensates the cumulative SLA value. Another effect observed in cases of lower LM is the much higher queues due to the longer duration of high load.

Overall, for every scenario, the controller using either Type–1 or Type–2 control scheme manages to reduce resources consumption regardless of the load used in such a way that states with better financial performance are reached.

6.5. *Experiment 4: Timing of Termination and Impact on Financial Performance*

The next experiment focuses on aspects related to the timing of the termination action: (a) just after the request has been received but before it has been processed, or (b) during execution when the action is being processed.

In Figure 11, violin plots are used to illustrate the distribution of Total SLA in system states measured during the experiments for the two approaches mentioned above. The left column shows control approach (a) whilst the right one shows approach (b). Four control schemes, Types 1, 2, 3 and 4, and no–control (Type–0), (cf. Table 2), have been tested with the two approaches. The plots focus on one load condition with mean arrival time equal to 4 time buckets, while the expected mean execution time of actions, l (cf. Table 4), was set to 4.2. When the load pattern reached around 95%, the system was already above limit and was going into saturation. Table 8 contains aggregated general performance improvement values per type of control scheme collected in simulations which involve a wide range of load conditions across all scenarios.

Termination during execution, namely control approach (b), offers substantial flexibility allowing the controller to improve the system's revenue/costs performance significantly. Although, in some cases, the added value seems low, as shown in Figure 11 and Table 8, cumulatively approach (b) is an attractive option whenever an action takes longer and/or is affecting resources consumption, deadline violation costs are substantial and/or termination penalty is not high.

Of course, performance is impacted by the fact that allowing the execution of an action causes resources consumption, and thus, it introduces costs. So after termination, the benefits of releasing these resources are lower than applying approach (a) directly. Nevertheless, the potential penalty for the termination remains the same, even though in certain cases the controller may find that it is still worth applying termination immediately after the start of an action.

In most cases, Type–4 control is the best. Only in scenario 5, simpler control schemes (Type–1, 2, or 3) are better, as there are no penalties. Although the model does not

[12]Figure 9 and 10 shows distributions of systems states collected in 336 test runs (168 experiments executed twice) that generated 2GB of trace data.

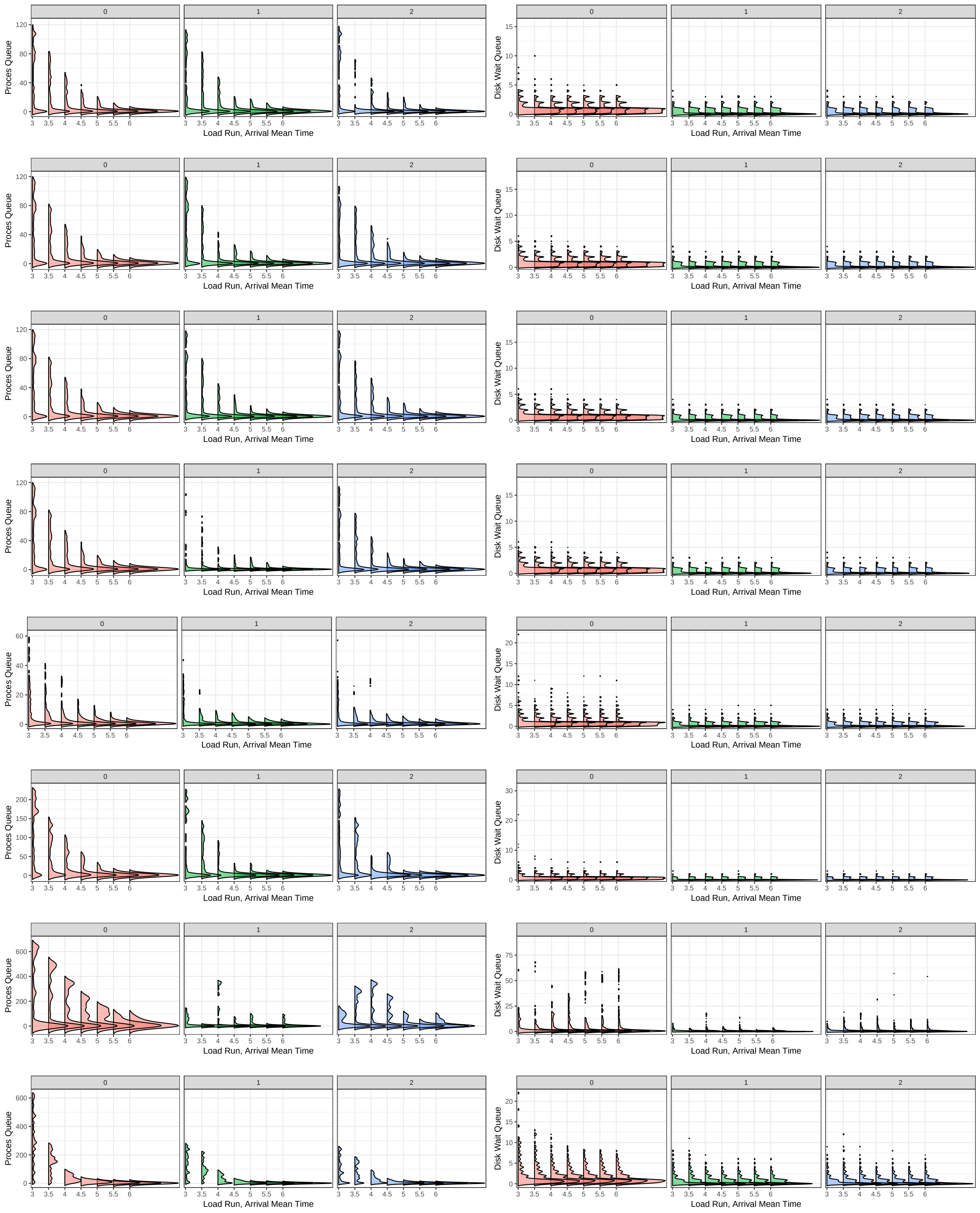


Figure 9. Experimental results for Type–0 (no–control) and Types 1 and 2 control schemes (cf. Table 2) for each of the eight scenarios (cf. Table 6). The diagram shows the distribution of system states as a function of load run. Each experiment contains around 500K states sampled during 168 tests: 3 control types, 8 scenarios, 7 load runs, across 2500 time steps. Every experiment was run twice for incremented pseudo-random number generator seed. The focus is on the effect of the control on resources utilisations, namely Processor and Disk Queues, as a function of load run.

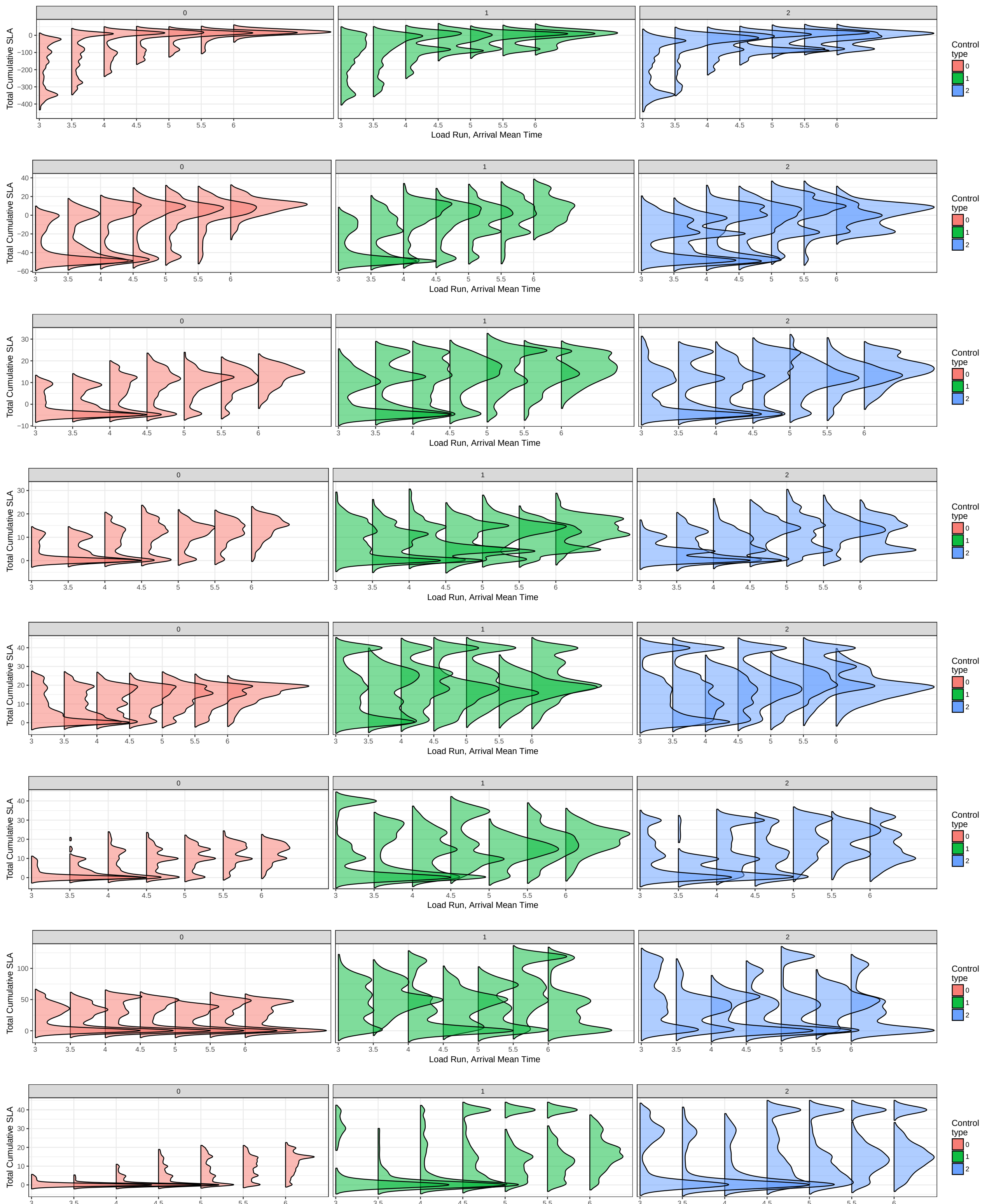


Figure 10. Effect of the control on financial performance, represented by the Total Cumulative SLA values of the system states with respect to load run. Results are for the experiments of Figure 9, i.e. around 500K states sampled during 168 tests: 3 control types, 8 scenarios, 7 load runs, across 2500 time steps.

Table 8. Financial performance increase $P \sim f_{SLA}$ between control (Types 1, 2, 3 and 4) and no–control (Type–0), in independent test runs across all scenarios (cf. Table 6), in terms of mean T_{SLA} values of the simulations (cf. Figure 11)

Control Type	Only before (a)	During (b)	Difference [%]
1 to 0	16.23	19.84	22.28
2 to 0	18.34	19.18	4.59
3 to 0	19.67	19.99	1.65
4 to 0	32.16	35.38	10.01

take into account the computation loss related to control instrumentation processing, the termination during execution approach (right column) is only slightly better. The difference is clearly visible only in scenarios 1, 3, 4. Termination approach (b), applying control during action execution, was best performing for Type–1 control scheme where an average improvement of 19% on average was noted across all scenarios. For Type–3 control the improvement was 14%, while for Type–4 the amelioration was only 4%, even thought Type–4 gives the best results over all.

7. Discussion

In enterprise information systems, high load situations and requirements for stable service at times of peak demand are typically dealt with by increasing computational power or by releasing load; for example, terminating incoming actions that appear less important to the new operating context and its activity demand pattern.

The experiments identified many benefits, qualities, but also implications of the termination control when it is applied to different action types of distinct run–time characteristics under dedicated service level objectives.

Table 9 presents a qualitative comparison of main control mechanisms for cloud–based enterprise information systems. Action termination can be an especially interesting option for cloud service providers that would like to mitigate the risk of the longer VM spin-off in situations of a sudden load increase. In such cases, they need to pay additional attention to prevent overload when auto-scaling is too costly (reached certain threshold), and responses times (execution times) are strictly defined in SLAs.

The benefits of the weaved–in termination control approach for enterprise systems in real–world applications can be substantial: (i) when it is preferable to return error before the maximum contracted execution time is reached than to calculate a response that takes much longer than normally expected. In such cases, the control framework enables termination errors to be handled on the client side for the price of maintaining a tighter conversation (of more frequent responses on requests) with the client systems supporting in this way a higher availability. This strategy may be very effective especially in micro–services architectures where many weakly–coupled components are present. In this context, instabilities caused by network issues or over-utilised services must be considered by design in the handling frameworks. Thus, the costs of extending the error management with termination mechanisms should be fairly low. Effectively, the action termination control swaps risks of execution time unpredictability on response type uncertainty but in a precisely defined time frame set in the SLA function definition. This can be applied to cloud service provider systems managing PaaS, FaaS and SaaS, but also to client systems deployed on IaaS where the control framework is weaved-into the application. (ii) When a service provider or client changes/renegotiates the SLA in run–time, this can be supported by the framework's adaptive nature.

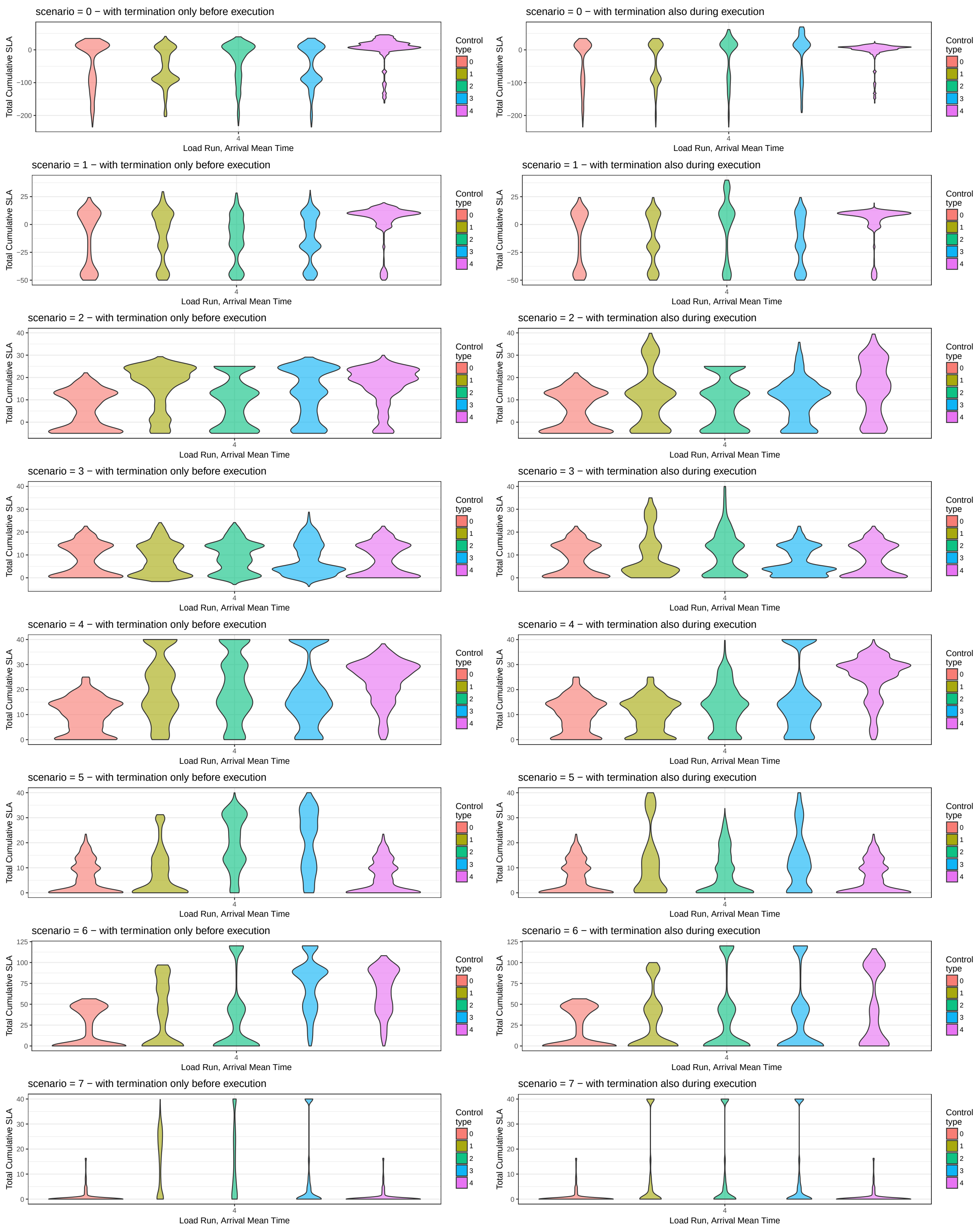


Figure 11. Results of experiments testing four control schemes (Types 1, 2, 3, and 4) and no–control (Type–0), (cf. Table 2), in all load and SLA definition scenarios (cf. Table 6) for a given mean arrival time equal to 4. The violin plots show distributions of Total Cumulative SLA values measured at each one of the monitored system states. The column on the left side shows values for states where termination was performed only before action execution, whilst the column on the right side refers to states where termination was done also during the action execution.

Table 9. Comparison of main control approaches for cloud–based enterprise information systems and their applicability to different service delivery models.

Control approach		Main cloud features support			Applicability to cloud service layer			
Name	Architecture	Load	Scalability	SLA	IaaS	PaaS	FaaS	SaaS
Server Consolidation[a]	Hypervisor[c]	Strong[d]	Strong	Weak[f]	Strong	→ (Strong, often given via IaaS)	→	→
Virtual Machines Migration[b]	Hypervisor[c]	Strong[d]	Strong[e]	Weak[f]	Strong	→ (Strong, often given via IaaS)	→	→
Conteneralization[g]	OS-level processes isolation[h]	Strong[i]	Strong[j]	Weak[f]	Medium	Strong	→ (Strong, often given via PaaS)	→
Auto-scaling	IaaS Service Manger	Strong[di]	Strong	Weak[f]	Strong (Client-driven control mechanism)	—	—	—
Adaptive micro-scheduling, Asynchronous Messaging	Application tier, Message Queue Systems	Medium[if]	Medium[k]	Medium	N/A[n]	Strong (by event-driven platform framework, Application Server, MQ)	Strong	Strong
Request Termination	Application tier or Platform.	Medium[ifl]	Weak[k]	Strong[m]	N/A[n]	Medium (by admission control framework, AOP, App. Server)	Strong	Strong

[a]Server Consolidation is provided by multiplexing physical resources over virtualized environments adapting the assignment to the current system workload (Vogels 2008). Such an optimisation can be formulated as a bin-packing problem which is NP-hard (Padala et al. 2007).

[b]Dynamic VM placement to assign/reassign virtual resources to applications on-demand (Ahmad et al. 2015). VM migration is often used to consolidate VMs residing on multiple under-utilised servers onto a single server, so that the remaining servers can be set to an energy-saving state (Zhang, Cheng, and Boutaba 2010).

[c]Amongst other most notable technologies are Xen, VMware, Libvirt.

[d]Applicable to all load scenarios, with lower efficiency for sudden spikes in load.

[e]Secures scalability requirement is fulfilled, uses proactive/reactive control to get resources before/on demand.

[f]It is difficult to build a model that provides the mapping between request-level SLA and resources.

[g]Conteneralization by Linux kernel cgroups (control groups) has lightweight nature, but containers have to share a common kernel. It can be managed by technologies like Docker, DockerSwarm, Kubernetes, LXC, CoreOS.

[h]Linux kernel cgroups, Hyper-V for Windows

[i]It does not introduce the overhead of full virtualization neither on the hypervisor nor the guest operating systems.

[j]Without hypervisor overhead the container capacity auto-scales dynamically with computing load.

[k]Supports scalability requirements indirectly; works better in a composition with other control approaches.

[l]Very effective under high load and strict SLA function definitions. The approach is dealing well with run-times that are a mixture of different load profile functionalities or parameters sensitive functions.

[m]Just-in-time style of control is able to explore and find the mapping between SLAs and resources needs.

[n]Strong support if the control is applied through weaved-in block to the application code directly.

(iii) When SLA functions per action are not defined explicitly, but the cloud service provider desires to secure the operation of the more important functions, the controller can be used internally by the cloud management framework. This could be useful, for example, in a cloud-computing execution model like serverless computing that requires running an application integrated with the framework. (iv) When it is important to secure productivity or mitigate the risks of over-spending on less essential computation, action termination can be exposed to cloud clients directly on a Software as a Service basis in a form of an API.

Lastly, as the action level framework shows/exposes the details of the financial and computational performance of an application, it could become part of an augmented operation–toolbox to support human administrators coordinating autonomous controllers that operate in high-dimensional enterprise environments. This audit and visualisation, built on the top of the data, can offer insights into the dynamics of the system and of the algorithms employed. This is a specialised domain where human administrators require additional support mechanisms as they frequently experience problems when analysing high-frequency sampled time–series data from weeks and months to arrive at a judgement on a particular control decision. This could also work as a framework–level addition to cloud–provider–based Web Application Firewalls (WAF) offering additional data scanning to prevent DoS attacks.

8. Conclusions and Future Work

Cloud–computing, hybrid clouds, but also on–premises systems demand more and more autonomous control integrated within cloud deployable enterprise information systems. To this end, this paper presented an approach that is driven by SLA requirements to design and test a neural adaptive controller with the goal to satisfy the quality of the services provisioned and the financial performance of the cloud services. These services can be provided by systems deployed to IaaS, PaaS but also SaaS models placed in complex spaces of many SLA and action performance characteristics. Metrics signals gathered during normal operation of the enterprise system are used to build a database of time sequences that effectively contains definitions of hidden causality chains present in the system. These data are used to train control blocks on how calls should be managed and what is the most promising decision to feed the actuator blocks at each time step.

Experiments using different control schemes based on this approach provide evidence that system's revenue generation and cloud service provider performance can be significantly improved by applying termination control coupled with adaptive controller.

Further validation of the approach is needed and enhancements, particularly through designing more complex control blocks for scheduling in order to control specific operating areas. To this end, in the future we will continue enhancing the framework with scheduling algorithms (Section 5.1), especially those that are dealing better with overloaded systems, and apply more complex system architectures, allowing to test memory allocation or network consumption as key resources for distributed systems.

Another promising area for further work would be to extend the control block and the neural model of action types. Such extension would support collecting and processing run–time data directly without evaluation that unavoidably comes with longer feedback time. Furthermore, using adaptive thresholds for defining high-mark and low-mark states (Section 5.3) would be useful as it would allow the controller to regulate how aggressive or smooth the applied control is; for example, reducing the frequency of terminations when the system under control has not reached certain operating regions yet, or smoothen the effect of the control in certain areas of the system state space. Lastly, we see potential in an extension that would allow the controller to switch off the decision block when the system is not significantly loaded. This will reduce the activity of the control block to sampling only, and would allow implementing a risk mitigation strategy for preventing potential terminations that do not bring additional benefit since the available resources are free to support a good level of operations.

Acknowledgement

This work was partially supported by Solid Software Solutions[13].

References

Ahmad, Raja Wasim, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Shiraz, Abdullah Yousafzai, and Feng Xia. 2015. "A survey on virtual machine migration and server consolidation frameworks for cloud data centers." *Journal of network and computer applications* 52: 11–25.

Al-Dahoud, Ahmad, Ziad Al-Sharif, Luay Alawneh, and Yaser Jararweh. 2016. "Autonomic cloud computing resource scaling." In *4th International IBM Cloud Academy Conference (ICACON 2016), University of Alberta, Edmonton, Canada, IBM*, .

Almeida, Jussara, Virgílio Almeida, Danilo Ardagna, Ítalo Cunha, Chiara Francalanci, and Marco Trubian. 2010. "Joint admission control and resource allocation in virtualized servers." *Journal of Parallel and Distributed Computing* 70 (4): 344–362.

Apache. 2004. "Version 2.0." *The Apache Software Foundation: Apache License, Version 2.0, January 2004* http://www.apache.org/licenses/LICENSE-2.0.

Arlitt, Martin, and Tai Jin. 1998. "1998 World Cup Web Site Access Logs." The Internet Traffic Archive, sponsored by ACM SIGCOMM. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

Arlitt, Martin, and Tai Jin. 2000. "A workload characterization study of the 1998 world cup web site." *IEEE network* 14 (3): 30–37.

Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau. 2015. *Operating Systems: Three Easy Pieces.* 0th ed. Arpaci-Dusseau Books.

Baldini, Ioana, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, et al. 2017. "Serverless computing: Current trends and open problems." In *Research Advances in Cloud Computing*, 1–20. Springer.

Barroso, Luiz André, Jimmy Clidaras, and Urs Hölzle. 2013. "The datacenter as a computer: An introduction to the design of warehouse-scale machines." *Synthesis lectures on computer architecture* 8 (3): 1–154.

Barroso, Luiz André, and Urs Hölzle. 2007. "The case for energy-proportional computing." *Computer* 40 (12).

Beloglazov, Anton, Jemal Abawajy, and Rajkumar Buyya. 2012. "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing." *Future Generation Computer Systems* 28 (5): 755–768.

Beloglazov, Anton, and Rajkumar Buyya. 2010. "Energy efficient allocation of virtual machines in cloud data centers." In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 577–578. IEEE.

Berl, Andreas, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. 2010. "Energy-efficient cloud computing." *The computer journal* 53 (7): 1045–1051.

Bi, Jing, Haitao Yuan, Wei Tan, MengChu Zhou, Yushun Fan, Jia Zhang, and Jianqiang Li. 2017. "Application-aware dynamic fine-grained resource provisioning in a virtualized cloud data center." *IEEE Transactions on Automation Science and Engineering* 14 (2): 1172–1184.

Busoniu, Lucian, Robert Babuska, and Bart De Schutter. 2008. "A comprehensive survey of multiagent reinforcement learning." *IEEE Trans. Systems, Man, and Cybernetics, Part C* 38 (2): 156–172.

Buyya, R., C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. 2009. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility." *Future Generation computer systems* 25 (6): 599–616.

[13]http://www.solidsoftware.pl/

Buyya, Rajkumar, Saurabh Kumar Garg, and Rodrigo N Calheiros. 2011. "SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions." In *Cloud and Service Computing (CSC), 2011 International Conference on*, 1–10. IEEE.

Calheiros, Rodrigo N, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. 2011. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms." *Software: Practice and Experience* 41 (1): 23–50.

Cherkasova, Ludmila, and Peter Phaal. 2002. "Session-based admission control: A mechanism for peak load management of commercial web sites." *IEEE Transactions on computers* 51 (6): 669–685.

Demirci, Mehmet. 2015. "A survey of machine learning applications for energy-efficient resource management in cloud computing environments." In *Machine Learning and Applications (ICMLA), 2015 IEEE 14th International Conference on*, 1185–1190. IEEE.

Dmitry, Namiot, and Sneps-Sneppe Manfred. 2014. "On micro-services architecture." *International Journal of Open Information Technologies* 2 (9).

Duolikun, Dilawaer, Tomoya Enokido, and Makoto Takizawa. 2017. "An energy-aware algorithm to migrate virtual machines in a server cluster." *International Journal of Space-Based and Situated Computing* 7 (1): 32–42.

Dutreilh, Xavier, Aurélien Moreau, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. 2010. "From data center resource allocation to control theory and back." In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, 410–417. IEEE.

Duy, Truong Vinh Truong, Yukinori Sato, and Yasushi Inoguchi. 2010. "Performance evaluation of a green scheduling algorithm for energy savings in cloud computing." In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 1–8. IEEE.

Fan, Xiaobo, Wolf-Dietrich Weber, and Luiz Andre Barroso. 2007. "Power provisioning for a warehouse-sized computer." In *ACM SIGARCH computer architecture news*, Vol. 35, 13–23. ACM.

Ferrer, Ana Juan, Francisco HernáNdez, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, RaüL Sirvent, et al. 2012. "OPTIMIS: A holistic approach to cloud service provisioning." *Future Generation Computer Systems* 28 (1): 66–77.

Fischer, Andreas, Juan Felipe Botero, Michael Till Beck, Hermann De Meer, and Xavier Hesselbach. 2013. "Virtual network embedding: A survey." *IEEE Communications Surveys & Tutorials* 15 (4): 1888–1906.

García-Valls, Marisol, Tommaso Cucinotta, and Chenyang Lu. 2014. "Challenges in real-time virtualization and predictable cloud computing." *Journal of Systems Architecture* 60 (9): 726–740.

Göbel, Johannes, Philip Joschko, Arne Koors, and Bernd Page. 2013. "The Discrete Event Simulation Framework DESMO-J: Review, Comparison To Other Frameworks And Latest Development." In *ECMS*, 100–109.

Hathaway, David H, Robert M Wilson, and Edwin J Reichmann. 2002. "Group sunspot numbers: Sunspot cycle characteristics." *Solar Physics* 211 (1-2): 357–370.

He, Yunlong, Jun Huang, Qiang Duan, Zi Xiong, Juan Lv, and Yanbing Liu. 2014. "A Novel Admission Control Model in Cloud Computing." *arXiv preprint arXiv:1401.4716* .

Hellerstein, J., S. Parekh, Y. Diao, and D.M. Tilbury. 2004. *Feedback control of computing systems.* Wiley-IEEE Press.

Hoang, Ha Nguyen, Son Le Van, Han Nguyen Maue, and Cuong Phan Nhat Bien. 2016. "Admission control and scheduling algorithms based on ACO and PSO heuristic for optimizing cost in cloud computing." In *Recent Developments in Intelligent Information and Database Systems*, 15–28. Springer.

Hwang, Kai, Xiaoying Bai, Yue Shi, Muyang Li, Wen-Guang Chen, and Yongwei Wu. 2016. "Cloud performance modeling with benchmark evaluation of elastic scaling strategies." *IEEE Transactions on Parallel and Distributed Systems* 27 (1): 130–143.

Jakob, Nielsen. 1993. "Usability engineering." *Fremont, California: Morgan* .

Jensen, E Douglas, C Douglas Locke, and Hideyuki Tokuda. 1985. “A Time-Driven Scheduling Model for Real-Time Operating Systems.” In *RTSS*, Vol. 85, 112–122.

Kim, Kyong Hoon, Anton Beloglazov, and Rajkumar Buyya. 2009. “Power-aware provisioning of cloud resources for real-time services.” In *Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science*, 1. ACM.

Kim, Kyong Hoon, Anton Beloglazov, and Rajkumar Buyya. 2011. “Power-aware provisioning of virtual machines for real-time Cloud services.” *Concurrency and Computation: Practice and Experience* 23 (13): 1491–1505.

Kiran, Mariam, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. 2015. “Lambda architecture for cost-effective batch and speed big data processing.” In *Big Data (Big Data), 2015 IEEE International Conference on*, 2785–2792. IEEE.

Lee, Young Choon, and Albert Y Zomaya. 2012. “Energy efficient utilization of resources in cloud computing systems.” *The Journal of Supercomputing* 60 (2): 268–280.

Leontiou, Nikolaos, Dimitrios Dechouniotis, and Spyros Denazis. 2010. “Adaptive admission control of distributed cloud services.” In *Network and Service Management (CNSM), 2010 International Conference on*, 318–321. IEEE.

Liu, Shuo, Gang Quan, and Shangping Ren. 2010. “On-line scheduling of real-time services for cloud computing.” In *Services (SERVICES-1), 2010 6th World Congress on*, 459–464. IEEE.

Lorido-Botran, Tania, Jose Miguel-Alonso, and Jose A Lozano. 2014. “A review of auto-scaling techniques for elastic applications in cloud environments.” *Journal of grid computing* 12 (4): 559–592.

Malawski, Maciej, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. 2015. “Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds.” *Future Generation Computer Systems* 48: 1–18.

Mao, Ming, and Marty Humphrey. 2012. “A performance study on the vm startup time in the cloud.” In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 423–430. IEEE.

Mao, Ming, Jie Li, and Marty Humphrey. 2010. “Cloud auto-scaling with deadline and budget constraints.” In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, 41–48. IEEE.

Messina, Fabrizio, Giuseppe Pappalardo, Corrado Santoro, Domenico Rosaci, and Giuseppe ML Sarné. 2014. “An agent based negotiation protocol for cloud service level agreements.” In *WETICE Conference (WETICE), 2014 IEEE 23rd International*, 161–166. IEEE.

Messina, Fabrizio, Giuseppe Pappalardo, Corrado Santoro, Domenico Rosaci, and Giuseppe ML Sarné. 2016. “A multi-agent protocol for service level agreement negotiation in cloud federations.” *International Journal of Grid and Utility Computing* 7 (2): 101–112.

Miller, Robert B. 1968. “Response time in man-computer conversational transactions.” In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, 267–277. ACM.

Mohana, RS, and P Thangaraj. 2013. “Machine learning approaches in improving service level agreement-based admission control for a software-as-a-service provider in cloud.” *Journal of Computer Science* 9: 1283–1294.

Moreno, Ismael Solis, Renyu Yang, Jie Xu, and Tianyu Wo. 2013. “Improved energy-efficiency in cloud datacenters with interference-aware virtual machine placement.” In *Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on*, 1–8. IEEE.

Muppala, Sireesha, Guihai Chen, and Xiaobo Zhou. 2014. “Multi-tier service differentiation by coordinated learning-based resource provisioning and admission control.” *Journal of Parallel and Distributed Computing* 74 (5): 2351–2364.

Padala, Pradeep, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. 2007. “Performance evaluation of virtualization technologies for server consolidation.” *HP Labs Tec. Report* 137.

Page, Bernd, Wolfgang Kreutzer, and Björn Gehlsen. 2005. *The Java simulation handbook: simulating discrete event systems with UML and Java.* Shaker Verlag.

Ranaldo, Nadia, and Eugenio Zimeo. 2016. "Capacity-driven utility model for service level agreement negotiation of cloud services." *Future Generation Computer Systems* 55: 186–199.

Salehi, Mohsen Amini, Bahman Javadi, and Rajkumar Buyya. 2012. "Preemption-aware admission control in a virtualized grid federation." In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, 854–861. IEEE.

Scheffler, Konrad, and Steve Young. 2002. "Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning." In *Proceedings of the second international conference on Human Language Technology Research*, 12–19. Morgan Kaufmann Publishers Inc.

Sharifian, Saeed, Seyed A Motamedi, and Mohammad K Akbari. 2008. "A content-based load balancing algorithm with admission control for cluster web servers." *Future Generation Computer Systems* 24 (8): 775–787.

Sikora, T. D., and G. D. Magoulas. 2013. "Neural Adaptive Control in Application Service Management Environment." *Evolving Systems 4(4)* 267–287.

Sikora, Tomasz D, and George D Magoulas. 2014. "Search-guided activity signals extraction in application service management control." In *Computational Intelligence (UKCI), 2014 14th UK Workshop on*, 1–8. IEEE.

Sikora, Tomasz D, and George D Magoulas. 2015. "Evolutionary approaches to signal decomposition in an application service management system." *Soft Computing* 1–22.

Sim, Kwang Mong. 2013. "Complex and concurrent negotiations for multiple interrelated e-markets." *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics* 43 (1): 230–245.

Stallings, William. 2014. *Operating Systems: Internals and Design Principles— Edition: 8.* Pearson.

Sutton, Richard S. 1992. "Introduction: The challenge of reinforcement learning." In *Reinforcement Learning*, 1–3. Springer.

Sutton, Richard S, and Andrew G Barto. 1998. *Reinforcement learning: An introduction.* MIT press Cambridge.

Tim Lechler, Johannes Goebel Peter Wueppen Ruth Meyer Philip Joschko Fredrich Broder Malte Unkrig Christian Mentz Nicolas Knaak Gunnar Kiesel Tim Janz, Soenke Claassen. 2014. "DESMO-J: A Framework for Discrete-Event Modeling and Simulation." April. `http://desmoj.sourceforge.net/`.

Tran, Dang, Nhuan Tran, Giang Nguyen, and Binh Minh Nguyen. 2017. "A Proactive Cloud Scaling Model Based on Fuzzy Time Series and SLA Awareness." *Procedia Computer Science* 108: 365–374.

Urdaneta, Guido, Guillaume Pierre, and Maarten Van Steen. 2009. "Wikipedia workload analysis for decentralized hosting." *Computer Networks* 53 (11): 1830–1845.

Urgaonkar, Bhuvan, and Prashant Shenoy. 2004. "Sharc: Managing CPU and network bandwidth in shared clusters." *IEEE Transactions on Parallel and Distributed Systems* 15 (1): 2–17.

Vogels, Werner. 2008. "Beyond server consolidation." *Queue* 6 (1): 20–26.

Xiong, Pengcheng, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Calton Pu, and Hakan HacigümüŞ. 2011. "ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers." In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 15. ACM.

Yu, Minlan, Yung Yi, Jennifer Rexford, and Mung Chiang. 2008. "Rethinking virtual network embedding: substrate support for path splitting and migration." *ACM SIGCOMM Computer Communication Review* 38 (2): 17–29.

Yuan, Haitao, Jing Bi, Wei Tan, and Bo Hu Li. 2016. "CAWSAC: Cost-aware workload scheduling and admission control for distributed cloud data centers." *IEEE Transactions on Automation Science and Engineering* 13 (2): 976–985.

Zhang, Qi, Lu Cheng, and Raouf Boutaba. 2010. “Cloud computing: state-of-the-art and research challenges.” *Journal of internet services and applications* 1 (1): 7–18.

Zhang, Wen-Jun, and Yingzi Lin. 2010. “On the principle of design of resilient systems–application to enterprise information systems.” *Enterprise Information Systems* 4 (2): 99–110.

Zheng, Wei, and Rizos Sakellariou. 2013. “Budget-deadline constrained workflow planning for admission control.” *Journal of grid computing* 11 (4): 633–651.

Zhu, Xiaomin, Laurence T Yang, Huangke Chen, Ji Wang, Shu Yin, and Xiaocheng Liu. 2014. “Real-time tasks oriented energy-aware scheduling in virtualized clouds.” *IEEE Transactions on Cloud Computing* 2 (2): 168–180.

Zhu, Zhaomeng, Gongxuan Zhang, Miqing Li, and Xiaohui Liu. 2016. “Evolutionary multi-objective workflow scheduling in cloud.” *IEEE Transactions on parallel and distributed Systems* 27 (5): 1344–1357.