REGULAR ARTICLE

Autonomic Rejuvenation of Cloud Applications as a Countermeasure to Software Anomalies

Pierangelo Di Sanzo[1] | Dimiter R. Avresky[2] | Alessandro Pellegrini*[1]

[1]DIAG, Sapienza, University of Rome, Italy
[2]International Research Institute for Autonomic Network Computing (IRIANC), Boston, MA, USA/Munich, Germany

Summary

Failures in computer systems can be often tracked down to software anomalies of various kinds. In many scenarios, it could be difficult, unfeasible, or unprofitable to carry out extensive debugging activity to spot the causes of anomalies and remove them. In other cases, taking corrective actions may led to undesirable service downtime. In this article, we propose an alternative approach to cope with the problem of software anomalies in cloud-based applications, and we present the design of a distributed autonomic framework that implements our approach. It exploits the elastic capabilities of cloud infrastructures, and relies on machine learning models, proactive rejuvenation techniques and a new load balancing approach. By putting together all these elements, we show that it is possible to improve both availability and performance of applications deployed over heterogeneous cloud regions and subject to frequent failures. Overall, our study demonstrates the viability of our approach, thus opening the way towards it adoption, and encouraging further studies and practical experiences to evaluate and improve it.

KEYWORDS:
Autonomic Computing, Software Rejuvenation, Proactive Management, Cloud Computing, Hybrid Cloud.

1 | INTRODUCTION

Nowadays, software anomalies are recognized as some of the major problems affecting performance and availability of computer applications. As an example, Pertet and Narasimhan[1] have shown that in web applications about 40% of anomalies are due to software errors such as memory leaks or unreleased locks. These anomalies might well and vastly accumulate, possibly bringing computing systems into thrashing states, which could generate performance loss and reduction of service availability. In the most severe circumstances, a system might also hang or crash due to resource exhaustion caused by the excessive accumulation of anomalies.

When dealing with distributed applications deployed at a geographical scale—such as many modern applications which are deployed on multiple cloud regions for availability/dependability reasons—addressing the aforementioned problems can be significantly complex. Indeed, many modern distributed applications involve large amounts of computing resources, complex system architectures, and may be subject to high dynamic workloads. The same is true for microservice-based applications, which often involve tens of different services deployed on virtualized infrastructures, which cooperatively implement the overall application. This makes it particularly hard to cope with failures of any nature, as well as to understand the associated causes, or to promptly perform proper correcting actions. Also, it may be the case that a full development cycle to identify the root cause of software anomalies, to fix them, and to deploy a new version of the application might require a long time span, ranging up to several weeks or months. In the meanwhile, end users expect the application to run correctly and to promptly serve their requests, with no excuse.

In this article, we explore an alternative and complementary approach to handle run-time errors in distributed cloud applications due to the accumulation of software anomalies. The approach concentrates on the *effects* of anomalies rather than on their root cause, in order to take prompt corrective actions. In particular, it relies on the well-know *software rejuvenation*[2] technique, which consists of forcing the state of the application/ system to a "clean" state, i.e. a state where the system/application is known to work correctly. Examples of basic actions for cleaning up the state include restarting the application or rebooting the system.

Nevertheless, software rejuvenation as-is does not allow to properly meet some non-functional goals of applications which suffer from software anomalies. Indeed, there are two common ways to use software rejuvenation: i) the application is detected to have reached a failure state, then it is rejuvenated, or ii) the application is rejuvenated periodically, as a preventive action, independently of the actual health of the application. Both of them are a naïve implementation of rejuvenation approaches, and generally result in sub-optimal resource exploitation and poor user experience. Indeed, in scenario i) the effects of the anomalies can be perceived by the user since rejuvenation is triggered only after that the system reached a failure state. On the other hand, in scenario ii), a too frequent rejuvenation might lead to unnecessary resource waste, and can increase the operational costs of the system. Also, in both scenarios the rejuvenation procedure commonly generates a time window of non-responsiveness, which might be unacceptable for users.

Therefore, to make proper exploitation of software rejuvenation techniques, two fundamental objectives should be targeted: 1) end-users of the application shall not notice the rejuvenation procedure, and 2) an autonomic approach should be enforced, making the system able to identify by itself the best-suited time instant at which the rejuvenation process should be carried out. Both these goals can be met via a smart *proactive* rejuvenation approach, which consists of proactively trigger rejuvenation actions at the right time to avoid that the system reaches an undesired health state. How to predict that the system is approaching an undesired state in an application-agnostic way is another aspect which requires great care.

Thus, in this article, we present the Overlay-based Cloud-oriented Elastic and Self-Healing (OCES) framework, a holistic autonomic management framework which implements the approach we propose, and that we designed targeting applications deployed on multiple cloud regions—hence, also including hybrid cloud environments. The goal of OCES is to orchestrate the whole lifetime of distributed cloud-based applications to account for software anomalies, with the aim of improving both availability and performance.

Overall, OCES can perform the following actions:

- Generating prediction models to determine when a Virtual Machine (VM) hosting an application is approaching a critical health state.
- Exploiting the generated models to monitor in real-time whether a VM should be deemed about to fail. In this case, OCES is able to autonomously manage cloud resources to instantiate a new VM instance to replace the about-to-fail one, thus allowing to eliminate or minimize the outage time of the application. Then, it silently (and consistently) runs the rejuvenation procedure to recover the failing instance, leaving it in state to be ready to be (re)used.
- Balancing the workload across the multiple cloud regions to reduce the effects of frequent failures and to optimize the overall application responsiveness. In this process, it also takes into account the possible heterogeneity of the available resources in the different regions.

We refer to OCES using the term *holistic* to emphasize that it is based on an approach that copes in a synergistic way with the different and complementary aspects of the problem of run-time management of software anomalies. Indeed, rather that simply offering techniques or tools to address separately each of them, OCES considers the problem as a whole. Indeed, it addresses all the related aspects and performs specific resolution actions, from the system monitoring and prediction of incoming failures, to the proactive and automatic replacement of failing resources, to the optimal load balancing for preventing or mitigate the effects of high failure rates in different cloud regions.

In this article, we presents the general organization of our framework, and the components which we have developed, in the attempt to make it a reference design for practitioners which could be interested in the development of similar infrastructures. The level of the details in the discussion has been explicitly balanced with this audience in mind. Overall, the novel contributions of our work can be considered the following ones:

- The evaluation of a proactive way to exploit software rejuvenation to prevent or mitigate the effect of software failures on cloud-based applications
- The exploration of a new load balancing approach that takes into account the predictions about the time to failure of cloud resources and their distribution over different cloud regions
- The development of a framework that, using in synergy the above-mentioned techniques, is able to autonomously manage the available virtual resources to improve performance and availability of cloud-based applications distributed across multiple and heterogeneous cloud regions
- An experimental study of our framework to demonstrate a practical exploitation of the framework and to evaluate it in a real-world setting.

The ultimate goal of our work is to open the way towards a new holistic approach to cope with the problem of software anomalies in cloud-based application.

We present the description of the methodologies and implementation choices which drove the creation of OCES, focusing on the single components of the framework. Then we discuss how these components are glued together in the distributed environment to carry out the autonomic management of the application. Therefore, the remainder of this article is structured as follows. We introduce the OCES approach in Section 2. Then we describe in Section 3 how ML-based prediction models are constructed. The monitoring and rejuvenation phases are described in Section 4. In Section 5 we discuss the methodology behind the controlling of VMs in a distributed Multi-Cloud environment. Finally, in Section 6 we present and discuss experimental results using a real-world application. Related work is discussed in Section 7.

2 | SOFTWARE ERRORS AND ANOMALIES MANAGEMENT IN COMPUTER APPLICATIONS: INTRODUCTION TO THE OCES APPROACH

Performance and availability of computer applications may suffer from the presence of various software anomalies. For years, systematic research is being conducted on this theme[3], trying to identify and classify the nature and the potential causes of software anomalies. We remark that anomalies (or failures) are caused by the incorrect execution of a software system due to some fault, which is the manifestation of an error in some software component[4]. There is a variety of root causes of errors in computer applications. Generally, they are classified into four categories: method, people, tool and requirement[5]. The large number of potential root causes and the variety of effects of errors, which may be also very different depending on the specific nature of the software application, led to the development of various approaches for coping with this problem. In the early stages of the application development, the problem can be addressed by targeting the identification and removal of root causes of errors, e.g. by improving software development methods, or by using effective approaches for understanding and specifying the application requirements. Once that a software release of the application or of some software component is available, approaches and tools for detecting errors can be used (e.g. static analysis[6]), with the aim of applying correcting actions to remove them (e.g. bug fixing). Finally, once that the application has been deployed and is running, it is necessary to deal with anomalies which are manifested at run-time, and adopt some approach to try to resolve them or to mitigate their effects. The latter is the goal of our framework. Indeed, the approach OCES is based on concentrates on the *effects* of anomalies rather than on their root causes and the specific software errors which cause them. To this aim, OCES relies on an application-agnostic approach for constructing ML-based prediction models of failures and to cope with them. The construction of the prediction models is composed of four basic phases. In the following, we provides an overview of the different phases and the tools used by OCES in the different phases.

In the initial offline configuration phase, OCES provides automatic facilities to generate Machine Learning (ML) models that allow to predict when some abnormal condition is expected to occur. This abnormal condition could be related to some resource exhaustion, to some violation in the quality of service (QoS) or in a degraded perception of the quality of experience (QoE) by the end users of the application. We generically call this time *Remaining Time To Failure* (RTTF). The condition(s) to be detected as a failure can be defined by the user on the basis of the values of one or more selected system features, which can reveal that the system is approaching, e.g., a hanging/crashing point or it is working in a sub-optimal way (e.g., it is showing poor performance).

The set of prediction models generated upon configuration are then used by a dedicated autonomic controller module to monitor the health of the distributed/cloud-based application. This autonomic controller monitors the RTTF of each VM by relying on ad-hoc non-intrusive software probes installed in each VM. If it determines that a VM is close to its failure point, it ignites a rejuvenation process. We note that, although OCES is designed to target cloud-based applications deployed on VMs, the approach OCES is based on can potentially be used also in the case of other forms of resources virtualization. For example, rejuvenation techniques are currently being studied also in the context of containers[7]. Possible successfully results achieved with these techniques would offer the chance to extend the exploitability of our framework also in other context, like in container-based cloud applications[8].

To minimize the outage of the application, OCES encloses special-purpose VMs which act as load balancers. These can be regarded as the connection points from the application's clients, but also serve a monitoring purpose. Indeed, they also collect information related to the processing time of each request, in order to give OCES an additional estimation of the effect of the accumulating anomalies on the response time as perceived by clients—this is a case of QoE threshold violation. This information is used in combination with the data gathered by the software probes to construct a more comprehensive monitoring system of the health of the application.

To additionally account for the application's response time, OCES also acts as a manager of cloud resources. Relying on the elastic nature of cloud deployments, whenever the load on the VMs hosted in a cloud region increases or decreases too much, VMs are added or removed to/from the pool. In this way, OCES tries to reduce the response time, to increase the availability of generic applications, and to keep low the overall cost of the deployment.

3 | APPLICATION-AGNOSTIC CONSTRUCTION OF ML-BASED PREDICTION MODELS

As mentioned, one of the first targets of OCES is to enable monitoring of generic applications[9]. To cope with the heterogeneity of software stacks and runtime environments which characterize modern cloud applications, OCES uses system-level metrics. This choice is fundamental, because it allows to fine tune the monitoring intrusiveness. Also, it enables to reduce possible side-effects on performance and availability by monitoring tools, which might accelerate the degradation of the performance of the system[10] (the so-called Heisen-monitoring effect). In fact, requiring OCES to interact with specific software stacks to probe the health of an application might make it of reduced applicability, or could introduce an overhead which might affect the responsiveness of the application itself. For these reasons, our implementation of OCES uses probes that capture system-level metrics rather than specific application metrics. Obviously, by using also specific application metrics, it would be possible to rely also on specific information related to the internal software components of the application. This could provide an advantage in terms of reliability of models for predicting failures. However, OCES can be easily extended to work also with other types of metrics, including metrics which measure specific health parameters of an application captured through internal probes of the application. Indeed, it would be sufficient to provide data captured by these probes to OCES as input features and that the user includes these features within the definition of failure condition(s). Ultimately, this demonstrates the flexibility of our approach, which can be implemented using various metrics at different level of the software stack, according to the user needs.

To make the prediction phase more general and applicable to a wide range of applications, OCES bases the construction of ML-based prediction models on a preliminary monitoring phase. In this way, it is possible to generate prediction models on demand. This generation phase can be carried out in-vitro (i.e., by relying on dedicated instances of the application), or also in production environments. The user of OCES can specify, at configuration time, what is the combination of parameters, and possible threshold values, which instruct OCES to consider the application as failed. This possibility allows the users to potentially account for any number of software anomalies, and any combination of them. Of course, the construction of the models can be carried out incrementally, i.e. by carrying out a preliminary in-vitro construction of early models, which are then continuously refined in the production environment.

In this phase, every time a failure condition (as defined by the user) is met, the OCES framework logs the occurrence time, and the system is restarted—we will discuss later in the paper how we perform a system restart, so as to consistently bring back the application in a correct operating state. This operating scheme allows the OCES framework to collect the evolution of multiple system parameters over time, showing how they inter-operate to reach the failure point. All these data are then used to build and validate a number of prediction models, which are generated by using different ML algorithms.

To provide a more exhaustive analysis of the evolution of the application towards the failure point, OCES performs a preliminary data mangling phase in which the collected system feature values are composed to generate an additional set of derived metrics. These metrics allow to capture specific higher-level aspects of the application, and they can be used by the user of the framework to specify additional conditions which could make OCES deem a VM as about-to-fail. This approach therefore gives more semantic power to the framework, making it easier to use for end users.

All the data (both collected from the software probes and derived in the mangling phase) are automatically divided into different training sets, composed of different sub-sets of the features. These training sets are then fed into multiple ML algorithms, which will generate different prediction models of the RTTF of the application. We rely on multiple ML models at this stage because, intuitively, the large spectrum of applications and software stacks which are the target of OCES might show dynamics towards the failure point which can be significantly different from each other. In this way, we can automatically compare the performance of the various generated prediction models against the actual application, thus selecting the one which is more suitable to capture the evolution of the application and the definition of “failure” as described by the end user.

In the remainder of this section we provide more details about the overall process undertaken by OCES to build customized prediction models in an application-agnostic way. The process is schematized in Figure 1, which shows the four basic phases to construct the ML-based models, including: 1) collection of health data, 2) post-processing of data traces, 3) feature selection and 4) generation and validation of the models. All these phases are described more in detail in the subsections 3.1, 3.2, 3.3 and 3.4.

3.1 | Health Data Collection via Lightweight Software Probes

For the sake of simplicity, in our presentation we refer to the case of a generic client/server application, where we assume that software anomalies are memory leaks and non-terminated threads (e.g., due to some synchronization error on some condition variable). These may occur at different rates during the lifetime of the application. However, as we introduced, OCES is not limited to this kind of applications and this kind of anomalies.

OCES relies on a thin client/server monitoring architecture. The software probes installed in the monitored VMs generate measurements (in the form of *datapoints*) in a way similar in spirit to other tools available on conventional systems, such as collectd[11]. Nevertheless, the probes are

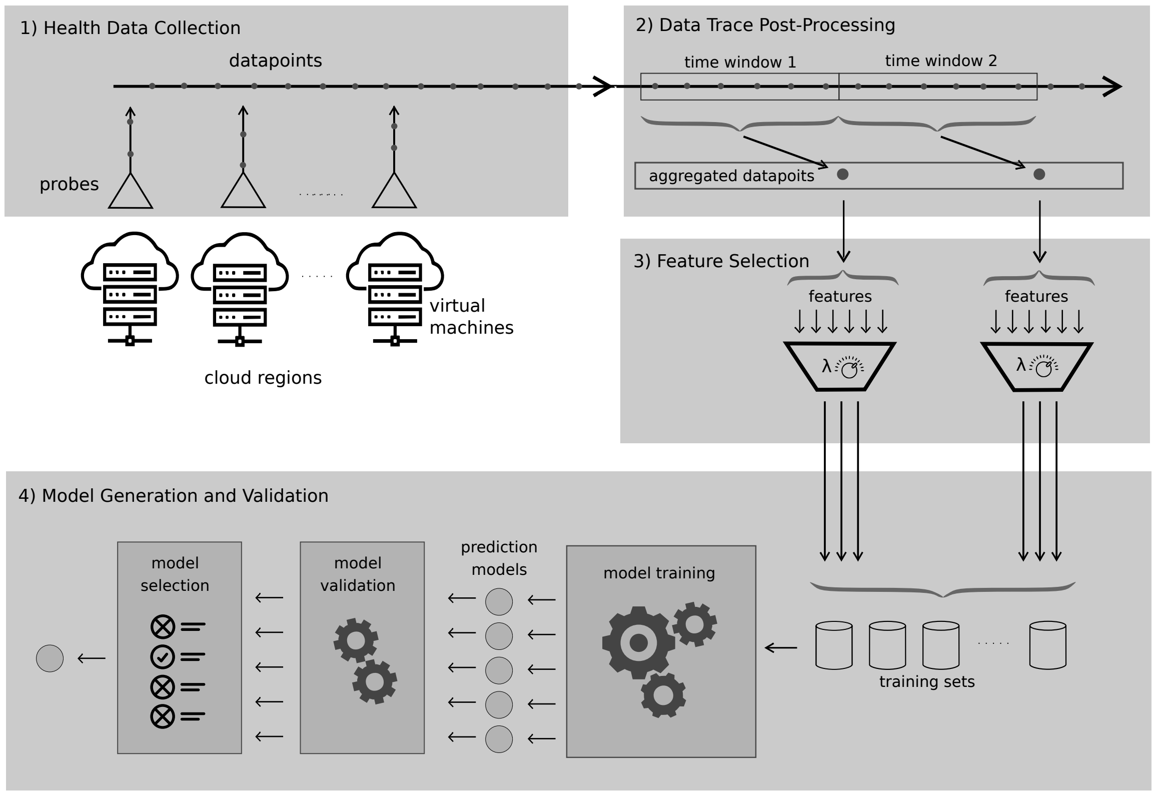


FIGURE 1 The four phases of the ML-based model construction process

.

directly connected via standard TCP/IP sockets to a separate VM hosting the OCES *data collection server*, which allows to monitor multiple VMs at a time, organizes, sorts, and stores on stable storage all the collected datapoints.

Each datapoint is timestamped with the elapsed time from system start, thus creating a *data trace* of the state of the system. When the user-specified failure condition is met, a timestamped *fail* event is added to the data trace and the system is rejuvenated[1], giving rise to a number of runs of the system.

Two different execution histories (or even the overall collection phase) highly depends on the anomaly occurrence rate and the total amount of available resources on the VMs. Indeed, for prediction models to be built with a significant degree of confidence, it is fundamental to collect a dataset which is sufficiently large. This is the reason why, as discussed before, OCES allows for *incremental construction of prediction models*. In particular, the client/server architecture which allows to monitor applications that can be run also in production environments. In case a new failure, OCES will trigger the re-generation of the prediction models, in an incremental way, using also the previously-collected data.

OCES uses a set of baseline system metrics which are collected by software probes. However, this set of metrics cab be customized by the end user according to its preferences. With the baseline configuration, each datapoint is a tuple in the form $\langle T_{gen}, n_{th}, M_{used}, M_{free}, M_{shared}, M_{buff}, M_{cached}, SW_{used}, SW_{free}, CPU_{us}, CPU_{ni}, CPU_{sys}, CPU_{iow}, CPU_{st}, CPU_{id}\rangle$, where:

- T_{gen} is the timestamp denoting the elapsed time since the system has started
- n_{th} is the number of active threads in the system
- M_{used} is the amount of memory used by applications running in the system
- M_{free} is the amount of memory freely available for usage by applications
- M_{shared} is the amount of memory used for buffers shared by applications

[1]In the case of a system deployed in a production environment, the rejuvenation process is subject to the responsiveness and consistency protocol which we shall describe later.

- M_{buff} is the amount of memory used by the underlying operating system to buffer data
- M_{cached} is the amount of memory used to cache disk data
- SW_{used} is the amount of swap space, which is currently used
- SW_{free} is the amount of swap space, which is currently free
- CPU_{us} is the percentage of CPU time dedicated to userspace processes
- CPU_{ni} is the percentage of CPU time occupied by user-level processes with a positive nice value (lower scheduling priority), CPU_{sys} is the percentage of CPU time spent in kernel mode
- CPU_{iow} is the percentage of CPU time spent waiting for a IO operations to complete
- CPU_{st} is the percentage of time a virtual CPU waits for a real CPU while the hypervisor is servicing another virtual processor (CPU steal time)
- CPU_{id} is the percentage of CPU time spent doing unfruitful work (i.e., the system is underloaded).

We selected the above-listed system metrics as the default set because they potentially allow to measure the effect on the system of many different anomalies typically affecting applications, such as memory leaks or threads stuck running on a spinlock due to some contention bug.

The output of the preliminary system monitoring phase includes a set of raw data representing the evolution of system features over time (in multiple runs). After the collection phase (or incrementally, in case the user configures OCES to do so), data histories are post processed and finally the prediction models are constructed.

3.2 | Data Trace Post-Processing

The second phase undertaken by OCES to build prediction models is a post-processing of the data traces generated by the software probes. This phase has a twofold goal: 1) performing a data munging operation on the raw generated data to correct some possible anomaly, and 2) enhancing the set of data used to build the models by adding some derived metrics, which could better capture some system dynamics. One of motivations for which the data munging phase is required is that we observed that the generation of original datapoints might incur in some skewing due to, e.g., the scheduler of the guest operating system, depending on the current workload. Particularly, as soon as the system is approaching the crashing point, this skew could have a higher impact, thus not providing a regular representation of the system behavior along the time axis. The data munging phase essentially aggregates multiple datapoints by relying on a user-defined time window. Each raw datapoint j generated from the software probes (shown in black in the figure) falls, according to its T_{gen} value, into one specific time window. All datapoints in one time window are averaged into an aggregated datapoint. Additionally, for each member of the raw tuple j, we compute its slope according to the following formula:

$$slope_j = \frac{x_j^{end} - x_j^{start}}{n}, \tag{1}$$

where x_j^{start} and x_j^{end} are the values of the feature j of the first and the last original datapoint falling in time window of the aggregated datapoint. These slopes are added as additional elements to the aggregated datapoints. Slopes allow to extract further knowledge about the dynamics of the system, which may show a highly variable behaviour. For example, some systems could show a constant increment of the resource usage over time until the crash point is actually approaching. At that time, some parameters could grow very quickly, even exponentially. The slopes, which could be interpreted as a simple approximation of a derivative, in such a scenario might be proven effective to promptly detect an upcoming crash point. As a specific case, let us consider the above-specified SW_{used} feature. If the system crashes due to memory exhaustion, SW_{used} will start growing faster when approaching the crash point. Therefore, the SW_{used} slope can be used effectively to build the prediction model.

An additional metric introduced is the *inter-generation time* among two consecutive datapoints, computed starting from T_{gen} timestamps. This additional information might allow to capture a possible delay at the level of software probes, which could be an indication of the increased load of the system (and possibly of an upcoming failure).

The timestamped *fail* events are used to generate, for each datapoint in the history, the RTTF. This is the additional fundamental information which will be used to construct the prediction models.

3.3 | Feature Selection

Aggregated datapoints have a large number of features, including all the raw metrics gathered from the software probes, the slope for each of them, the inter-generation time, and the RTTF. Since the end user is able to specify what is the set of metrics being traced by the software probes, this number could even increase highly, reaching thousands of features in some Cloud-related contexts[12]. Depending on the specific application (and its anomalies), they might not be necessary altogether to generate an accurate-enough prediction model. While building models using all these features will unlikely produce a less-accurate model, it is quite sure that the training time will increase. As mentioned, this could be unacceptable for the purposes that OCES intends to serve. Rherefore, OCES performs a selection phase on the features in the aggregated datapoints to identify an optimal subset which will not likely affect the accuracy of the model. This allows us to keep low the time required to build the models. As shown in Figure 1, the execution of this phase is optional, so that the user is able to choose whether in the next phase, the OCES Framework should consider the whole set of features, or only the ones selected during this phase. We base the feature selection phase on Lasso regularization[13]. For an aggregated datapoint $\bar{\mathrm{x}}$ and a control variable λ, Lasso regularization generates a vector $\bar{\beta}$ of factors β, whose elements are the weights of the vector $\bar{\mathrm{x}}$, which minimizes the following objective function:

$$\frac{1}{n}\sum_{j=1}^{n} V(y_j, \langle \beta, x_j \rangle) + \lambda ||\beta||_1, \tag{2}$$

where n is the number of data points from the aggregation step, each $\mathrm{x_j}$ is an input feature from every data point, $\mathrm{y_j}$ is the associated value of the dependent variable (the RTTF in our case) for the specific data point, and $\mathrm{V}(\mathrm{y_j}, \langle \beta, \mathrm{x_j} \rangle)$ is equal to $(\mathrm{y_j} - \beta^{\mathrm{T}}\mathrm{x_j})^2$.

The calculated vector β includes a (sub-)set of non-zero weights. All features associated with a zero weight can be therefore filtered out from the training set. Generally, while increasing the value of λ, more elements of the vector β are likely equal to 0. Particularly, these elements are likely those which have a smaller weight in the evaluation of the RTTF. Thus the effect of using higher values of λ is the reduction of the number selected features to be used in the ML models. Of course, using too high values of λ could have the effect of overly reducing the size of the training set, hence calling for a selection of the best-suited model, based on its accuracy. As the output of this phase, a number of training sets, each one including a sub-set of selected features and added metrics, is produced.

3.4 | Model Generation and Validation

Various ML methods exist to build prediction models. OCES is designed to generate multiple prediction models from the same datasets using different ML methods, and to compare their prediction accuracy, in order to select the best one. We have included in the default set of ML methods of OCES six linear and non-linear ML methods, namely Linear Regression[14], M5P[15], REP-Tree[16], Lasso as a Predictor, Support-Vector Machine (SVM)[17], and Least-Square Support-Vector Machine[18]. A description of these six methods is reported in the Appendix of this article. Of course, the set can be customized by the user by adding other methods or removing some of them.

After having generated all the models, they are evaluated to determine which one offers the best accuracy for the prediction of the RTTF. To this end, OCES computes a number of performance metrics by running predictions on a sub-set (validation set) of samples, possibly not used for the model training. The performance metrics which are used by OCES to estimate models' accuracy and overall goodness are:

- Mean Absolute Prediction Error (MAE): The average of the differences between predicted and real RTTF.
- Relative Absolute Prediction Error (RAE): It is relative to a simple predictor, namely the average of the actual measurement. RAE normalizes the total absolute error by dividing it by the total absolute error of the simple predictor.
- Maximum Absolute Prediction Error (MAPE): It is the maximum prediction error.
- Soft-Mean Absolute Prediction Error (S-MAE): It is calculated as the MAE, except that it is left-bounded by 0.
- Training Time: The time required by the learning method to build the model.
- Validation Time: The time required to complete the validation process.

If there is a single winner on all the metrics, OCES automatically selects it as the best-suited prediction model. On the other hand, the end user is asked to pick one of the models. This manual choice allows to trade off between accuracy and timeliness, depending on the nature of the application and (possibly) the SLAs that the application has to offer to the end users—this latter point explicitly accounts for the incremental setup for model generation by OCES.

4 | MONITORING AND REJUVENATING VIRTUAL RESOURCES

OCES uses the prediction models to predict the RTTF of VMs that host the applications. Every time that OCES detects that a VM instance is approaching a predicted failure point, it automatically spawns a new VM instance to replace the about-to-fail VM and redirects the load to the new VM. In the while, it allows the about-to-fail VM to complete its pending requests and then triggers a rejuvenation procedure.

We remark that OCES is designed to be independent of a specific rejuvenation technique. It allows the user to decide the most suitable technique to be used for each application. For example, often the simple VM restart (or restarting only the processes of the application) is a sufficient action to report the application to an healthy state. In other cases, the user may decide to use more complex rejuvenation techniques, such as checkpoint-based rejuvenation schemes[19]. Obviously, the application should support these kinds of schemes. In Section 6.3, we discuss the technique that we used in our experimental study.

To monitor the health of the VMs, OCES uses he same client/server architecture and software probes described in Section 3.1 used to gather datapoints. An additional VM, which we call the Virtual Machine Controller (VMC) is in charge of monitoring and controlling k couples of VMs (the slave VMs) acting as (replicated) servers. The slave VMs of a couple c_x (with $x \in [0, k-1]$) are named VM1$_x$ and VM2$_x$, respectively.

The VMC client is composed of several software modules, each one serving a different purpose for the monitoring/rejuvenation process, namely:

- A Communication Unit (CU), which interacts and exchanges data/control messages with all slave VMs;
- A Prediction Unit (PU), which evaluates the prediction model generated and selected in the previous phase of the lifetime of the OCES framework, which estimates the RTTF of all slave VMs as soon as a new datapoint is generated by a software probe;
- A Load Balancing Unit (LBU), which acts as the entry connection points for the clients, which forwards requests to some slave VMs, and which monitors the number of active connections to each slave VM;
- A Managing Unit (MU), which is in charge of rejuvenating, adding, and removing VMs from the pool of active cloud instances.

On the other hand, the architecture of the clients installed in the slave VMs to monitor the health and actuate the logic dictated by the VMC is composed by the following software modules:

- A Communication Unit (CU), which interacts with the VMC;
- A Measurement Unit (MeU), which assembles health datapoints by relying on the software probes;
- A Local Managing Unit (LMU), which sends datapoints to the VMC and receives control commands from the VMC.

The MU maintains the set of slaves VMs organized in couples[2], in order to determine, among the available resources, which should be active/inactive at a given time instant. At system startup, for each couple a single VM is started and marked as *active*. The other VMs are placed in the *stand-by* state. The LBU has access to the states of VMs, and upon a client connection, it will transparently redirect the request to one of the slave VMs, therefore acting as a reverse proxy.

On each active VM instance, the MeU component will act in a way similar to the collection phase. Namely, system features will be recorded, assembled into datapoints, and sent to the VMC. Upon the receipt of a datapoint, the PU will evaluate the prediction model, and generate a RTTF estimation.

By using the RTTF estimates, the VMC can implement an *on-line control loop* for the health of each VM instance. For each couple x of VMs currently managed by OCES, the control loop is implemented so as to execute the following steps:

1. If the predicted RTTF for VM1$_x$ is higher than a (user-tunable) threshold T, the instance is considered *healthy*, and no further action is taken.
2. On the other hand, the MU executes the following steps:
   (a) VM1$_x$ is marked as *stand-by*;
   (b) VM2$_x$ is marked as *active* and activates the VM instance—from now on, the LMU of VM2$_x$ will send health datapoints for the newly-activated VM, and the LBY will direct connections towards VM2$_x$;
   (c) It instructs the the LBU to notify when no active connections towards VM1$_x$ are present;
   (d) Once the LBU confirms that no active connections are in place (or if a user-tunable timer expires) it sends the *rejuvenate* command to VM1$_x$;

For the sake of clarity, we report in Figure 2 an example execution of the control loop when only two VMs belonging to the same couple are managed and monitored by OCES. At system startup, only VM1$_x$ is active. The LMU periodically sends health datapoints to the VMC, which predicts the estimated RTTF. After some time, VM1$_x$ is deemed to be about to fail, and the rejuvenation process begins. The VMC immediately activates VM2$_x$, but VM1$_x$ remains active still for a while, namely until the LBU notifies that no pending connections are present (or the timer expires). The rejuvenation of VM1$_x$ then takes place, with the VM being shut down and rejoining the pool in the *stand-by* state. After some additional time, VM2$_x$ might approach the failure point, and the same process takes place so as to rejuvenate VM2$_x$.

[2]Of course, additional VM couples can be added to the pool at any time, if the application architecture or dynamics requires to do so, e.g. in the case of microservice-based applications. Similarly, existing pairs can be removed, thus making an effective usage of the elastic capabilities of cloud environments, also accounting for cost effectiveness.

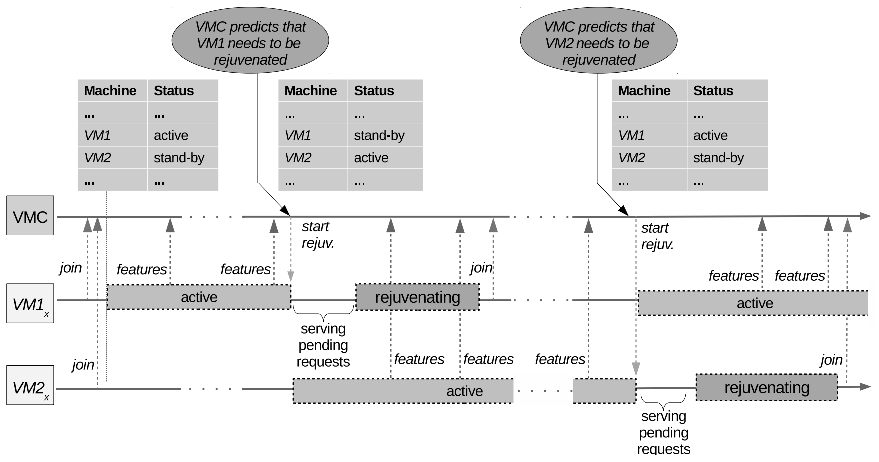


FIGURE 2 MLSR-Framework control flow diagram with a couple of VMs.

We note that, depending on the application, it could be required to execute additional consistency-oriented tasks during the rejuvenation process . As an example, some sessions tokens might required to be migrated to the new instance, or in-memory data could required to be flushed to distributed or remote databases. Also, with applications that use distributed and replicated database, data synchronization may be required. For example, this is the case of multiple copy databases[20] and with parallel accesses to shared data[21]. In our context, if a VM instance hosts a database, when the instance is replaced by a new one, data could require to be transferred to the new instance. Further, in same cases data may require to be synchronized with other databases hosted by some other VM instance of the application. Indeed, there may be the possibility that new data updates executed in parallel on other databases are not received during the data transfer process. These issues have been extensively dealt with in the literature, and several solutions to address these aspects have been already presented, in particular to reduce the perception by the end users of data synchronization[22,20]. Basically, the proper technique to be used depends on the nature of the application. For this reason, we designed OCES such that it is independent of the specific data synchronization techniques required by the application. Basically, OCES can triggers a specific routine in charge of launching the data synchronization procedure that has to be used with each specific application.

The user-specified threshold T is a *safety value*, which takes also into account the average time required by the application to perform all tasks necessary to shutdown consistently the VM, also accounting for pending requests processing. The companion control variable, namely the expiration time, ensures that OCES is able to rejuvenate the machine also in case a too high workload is being experienced, at the cost of losing some request. Of course, the dynamic management of VM pools which we describe later, in conjunction with the presence of reverse proxies implemented in the form of an LBU, allows to reduce the likelihood that this scenario arises. With respect to T, a too high value might force a too early rejuvenation, even though the VM is still running with an acceptable health. In Section 6, we discuss in more details the effects of T, leveraging results of an experimental study.

5 | MANAGEMENT OF VIRTUAL RESOURCES FOR INCREASED RESPONSIVENESS

The OCES framework allows to manage any number of VMs across multiple cloud regions. Regions can also be heterogeneous in terms of computing resources and distributed ate geographical scale[23,24]. Each single region is managed according to the architecture described in Section 4. In this section, we discuss the inter-region management of the application offered by OCES.

The cross-region management is based on two main components:

- Controllers, which manage local resources and communicate with controllers running in other cloud regions.
- LBUs, which are the same software components discussed in Section 4, but which can also redirect connections to LBUs hosted in different cloud regions.

Having multiple LBUs is typical of several real-world scenarios, where geographically-distributed applications provide multiple entry points, depending on the source location of the clients—DNS servers typically hide away the complexity of application reachability through multiple entry points. In other words, the availability of multiple LBUs intrinsically enables the possibility to rejuvenate VM instances, even in case the new instances are assigned a different IP address. Indeed, clients will connect to one of the available LBUs, which will in turn forward the requests to the destination VMs.

OCES implements a new load balancing approach that uses the Mean Time To Failure (MTTF) of VMs as an input parameter to decide how redirect the incoming requests. The MTTF of each VM in each region can be straightforwardly calculated using the ML models (e.g. by predicting the RTTF at time 0). The load balancing approach aims at balancing the load across the cloud regions in order to avoid that different regions are subject to highly different failure rates (and consequently rejuvenation rates) of VMs. This may be caused, e.g., by overloaded and underloaded regions. Highly different failure rates, in turn, can amplify the load unbalancing between overloaded and underloaded regions. Another cause of unbalancing is often the heterogeneity of resources available for an application in different cloud regions (e.g. a different number of VMs, or VMs with different computing power).

The specific goal of OCES is to balance the failure rates in the different regions, based on the so-called Region Mean Time To Failure (RMTTF) of each region. More in detail, among all the VMCs of all regions, one works as a leader. The VMC of a region i periodically sends to the leader VMC the last value of RMTTF, say $\mathsf{lastRMTTF_i}$, calculated as the average MTTF of all active VMs in the region i. When the leader VMC receives $\mathsf{lastRMTTF_i}$ at time t, the current RMTTF of the region i, say $\mathsf{RMTTF_i^t}$, is (re-)calculated by using the following weighted average:

$$RMTTF_i^t = (1-\beta)\cdot RMTTF_i^{t-1} + \beta \cdot lastRMTTF_i, \tag{3}$$

where $\mathsf{RMTTF_i^{t-1}}$ is the previous value of RMTTF and $0 \leq \beta \leq 1$.

OCES ships with three different load balancing policies. The goal of the policies is therefore to decide the fraction $\mathsf{f_i}$ of global incoming requests to be forwarded to a cloud region i to ensure that the different values of the current RMTTF of all regions converge (fast) to the same value. We present the three policies in the following. In Section 6.4 we present an experimental study to assessment of their capability in different application scenarios.

Policy 1: Sensible Routing

The first policy shipped by OCES is based on the work by Wang and Gelenbe[25], and is called *sensible routing*. Assuming to have N cloud regions, the fraction $\mathsf{f_i}$ of global incoming requests to be forwarded to cloud region i is calculated as:

$$f_i = \frac{RMTTF_i^t}{\sum_{j=1}^{N} RMTTF_j^t}. \tag{4}$$

The intuition behind sensible routing is that the fraction i should be proportional to the weight of the RMTTF over the RMTTF as observed in all regions.

Policy 2: Available Resources Estimation

A single numeric parameter is used by the *available resources estimation* policy, to abstract the amount of available resources in a cloud region. The idea behind available resources estimation is that, while software anomalies accumulate, each resource is “consumed” in a linear fashion. This means that if the accumulation of the anomalies is proportional to the number of incoming requests, then also resource consumption will be proportional to the requests. Therefore, the estimation of the resources which are available, at a given time, in a given region i can be obtained by the following equation:

$$Q_i = RMTTF_i^t \cdot f_i \cdot \lambda, \tag{5}$$

where λ is the global incoming request rate. The factor $\mathsf{f_i} \cdot \lambda$ estimates the incoming request rate of region i. Equation (5) grounds on the assumption that a higher RMTTF in the face of the same amount of incoming requests, under the proportionality assumption stated before, tells that that region has a higher amount of resources which can be used to serve requests.

At this point, the fraction $\mathsf{f_i}$ of requests to be forwarded to region i can be deemed proportional to the (estimated) number of available resources:

$$f_i = \frac{Q_i}{\sum_{j=1}^{N} Q_j}. \tag{6}$$

Policy 3: Exploration

The exploration policy exploits a hill climbing[14]-based search algorithm to estimate the Average RMTTF (ARMTTF) over all regions:

$$ARMTTF = \frac{\sum_{i=1}^{n} RMTTF_i^t}{N}. \tag{7}$$

A debit/credit approach is then used to increase/decrease the fraction of requests $\mathsf{f_i}$ assigned to each cloud region. In particular, if $\mathsf{RMTTF_i^t} > \mathsf{ARMTTF}$, the value is decreased, while it is increased if $\mathsf{RMTTF_i^t} < \mathsf{ARMTTF}$.

The set of overloaded regions is then determined as $\mathsf{OL} = \{i : \mathsf{RMTTF}_i < \mathsf{ARMTTF}\}$. Each region in this set receives a new value of f_i computed as:

$$f_i^{next} = \frac{RMTTF_i}{ARMTTF} \cdot f_i \cdot k, \tag{8}$$

where k is a constant scaling factor. Of course, the equality $\sum_{i=1}^{n} f_i = 1$ must hold. To this end, the *total variation of the flow of overloaded regions* is computed as:

$$\Delta f^{<} = \sum_{i \in UL} (f_i^{next} - f_i). \tag{9}$$

Then, the set of underloaded regions is determined as $\mathsf{UL} = \{i : \mathsf{RMTTF}_i > \mathsf{ARMTTF}\}$). Each region in this set received a new value of F_i determined as:

$$f_i^{next} = \frac{\Delta f^{<}}{\sum_{i=1}^{N} RMTTF_i} \cdot f_i \cdot k, \tag{10}$$

where k is the same scaling factor used before. Overall, new fractions are determined as:

$$f_i^{next} = \begin{cases} \frac{\mathsf{RMTTF}_i}{\mathsf{ARMTTF}} \cdot f_i \cdot k & \text{if } \mathsf{RMTTF}_i < \mathsf{ARMTTF} \\ \frac{\Delta f^t_<}{\sum_{i=1}^{n} \mathsf{RMTTF}_i} \cdot f_i \cdot k & \text{otherwise} \end{cases}. \tag{11}$$

5.1 | Global Management of Resources

We now discuss how these policies are applied by OCES in order to orchestrate the available virtual resources hosted in multiple cloud regions. The baseline assumption is that clients can connect to any cloud region, which is an assumption suitable for DNS-based geographical dispatching of incoming requests, also in the case of microservice-based applications. We therefore assume that each client can connect to an LBU in some region. The goal of the global management is to ensure the every region i processes the established fraction of request f_i.

In OCES, we reach this goal by means of a *global forward plan*. Upon the calculation of a new value of f_i by the leader controller—any of the aforementioned policies can be selected by the end user—the amount of incoming requests to be forwarded to external LBUs is determined. We note that this scheme basically implements a multi-level reverse proxy scheme, typically designed for simple load balancing purposes. In our case, the ultimate goal is to reduce the impact of rejuvenation on the overall distributed virtualized infrastructure.

To determine a global forward plan, we implement a distributed state-machine approach, in which four different states are envisaged, namely: Monitor, Analyze, Plan, and Execute. At system startup, all controllers are in the Monitor state. In this state, a controller enforces the actions previously discussed in Section 4. After some time, the controllers enter the Analyze state, in which ML-based predictions are used to compute the RMTTF_i value for each cloud region. The leader VMC will then gather all the estimate from all the other VMCs.

At the end of the gather phase, VMCs enter the Plan state. This is a dummy state for slave VMCs, as all the computation takes place at the leader. In this state, we apply the user-specified policy to compute the new value of f_i^t, for all the regions i in the system, where t is the *epoch timestamp* of the new workload distribution phase. At this point, the leader controller scatters the f_i^ts to the other VMCs, and the system globally transitions to the Execute state. In this state, each controller instructs their local LBU to forward incoming requests to other regions, so as to meet the value specified by the new f_i^t. It is the leader VMC that governs the state transitions, by means of control messages set to the other controllers, and manages the advancement from one epoch to the next.

It could be the case that this policy is not enough to counteract a sudden spike of workload received by some cloud region. OCES can be configured to instruct the VMCs to autonomically add new resources to the pool of locally-hosted VMs if the system observes a transient but strong degradation. In particular, each LBU monitors the time which is required to serve a request from incoming clients. Explicitly neglecting the network trip time, which is rarely an issue for modern cloud service providers, the VMC uses this additional measure by the local LBU to compute an estimation of the response time as seen by the clients. If the response time falls above some user-defined threshold, the VMC will add additional pairs of VMs to the local pool.

6 | EXPERIMENTAL ASSESSMENT

6.1 | Benchmark Setup

We have conducted an extensive assessment of all the software components of the OCES framework, and their interactions, in order to study the viability of our approach. We used a hybrid cloud environment, composed of a dedicated 32-core HP ProLiant server with 100 GB RAM located

TABLE 1 Weights assigned with $\lambda = 10^9$

Parameter	Weight
M_{used} (slope)	0.000019235560086
M_{free} (slope)	0.000236946638676
SW_{used} (slope)	0.000263386541515
SW_{free} (slope)	0.000263386541515
M_{free}	0.000263386541515
M_{shared}	0.000263386541515

in Munich (Germany), equipped with VMware Workstation 10.4 as the hypervisor, and a set of AWS EC2 instances in the Frankfurt and Ireland regions, for a total of 3 geographically distributed cloud regions. We refer to the AWS instances in Ireland as Region 1, to the AWS instances in Frankfurt as Region 2, and on the VMWare instances in Frankfurt as Region 3. In Region 1, we have used 6 `m3.medium` EC2 instances, while on Region 2 we have used 12 `m3.small` EC2 instances. VMs in Region 3 have been provided with 2 virtual CPU cores, 1 GB of memory, and 4 GB of virtual disk space. All VMs were equipped with Ubuntu 10.04 Linux Distribution (kernel version 2.6.32-5-amd64).

The real-world application being hosted by the VMs is a Java implementation[26] of the standard TPC-W benchmark[27], i.e. a multi-tier e-commerce web application. TPC-W benchmark specifies an e-commerce workload that simulates the activities of a retail on-line store. In particular, it emulates a number of users that browse and order books from the website. This is composed of 14 different web pages, each one dedicated to a specific activity, such book searching, book detail visualization, book ordering, cart visualization, order checking, etc. Each VMs of our experimental cloud environment hosts an implementation of the application tier.

We artificially introduced bugs in the TPC-W implementation to randomly generate software anomalies at run-time, in the form of memory leaks and unterminated threads, which can mimic compute resources stuck in the access to a critical section. The anomaly generation is probabilistic with respect to client connections—10% of requests generate a memory leak, 5% of requests generate an unterminated thread. This led to scenarios where each VM (thus each cloud region) can show different anomaly-occurrence patterns. We varied the number of active clients (towards each cloud region) in the interval $[16, 512]$, ensuring that the clients connected to each LBU where significantly different in number. We have run our experiments considering data traces associated with 200 different runs of the application. For consistency reasons, we have hosted the DBMS of the application in a separate VM, not subject to artificial software anomalies.

We organize the experimental data into different subsection. In Section 6.2 we study the accuracy of the prediction models' generation, which has been previously presented in Section 3. Section 6.3 shows the behavior of OCES when a single cloud region is used, to stress test the autonomic management which has been previously described in Section 4. Finally, Sections 6.4 and 6.5 present data related to the distributed management of the incoming requests by OCES, as it has been previously presented in Section 5.

6.2 | Prediction Models Accuracy

The first step in the model generation is associated with Lasso regularization to perform feature selection. We have carried out experiments to evaluate model's accuracy both when using the features selected during the regularization phase and when all the features are used. For the specific setup which we have used to carry out our experimental assessment, we report in Table 1 the number of features associated with a non-zero weight in the β vector when setting $\lambda = 10^9$. As mentioned, a higher value for λ determines a more strong selection of important features, possibly with an effect on accuracy. his is exactly the reason behind the selection of this value for λ and for the comparison with models generated when using all the features.

The first interesting aspect related to the data reported in Table 1 is that slopes play an important role in the construction of the prediction model. In more details, slopes associated with memory usage and swap usage play the most important role. This is an expected result, considering that one of the anomalies which we have injected in TPC-W is related to the generation of memory leaks upon each client connection. Therefore, the regularizatino phase correctly identifies memory consumption as one of the root causes for a failure of our application. Nevertheless, a strong regularization (λ is large) fails to capture the effects of other anomalies in the application, namely unterminated threads. Therefore, although the generation of prediction models in this scenario could be expected to be faster, we also expect the accuracy of the generated models to slightly drop.

To study this aspect, we report in Table 2 the accuracy obtained by training the models using the various ML algorithms in the two aforementioned scenarios, namely when using all the features in a datapoint and when using only the features which "survived" the regularization phase. For the sake of brevity, we only show the *Soft-Mean Absolute Error* (S-MAE) performance metric (in seconds). As it can be seen, we find a confirmation of

TABLE 2 Soft Mean Absolute Error—10% Threshold

Using all parameters		Using only parameters selected by Lasso	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	137.600	Linear Regression	156.603
M5P	79.182	M5P	118.292
REP Tree	69.832	REP Tree	108.476
SVM	132.668	SVM	146.594
SVM2	132.675	SVM2	146.607
Lasso ($\lambda = 10^1$)	405.187	Lasso ($\lambda = 10^1$)	405.187
Lasso ($\lambda = 10^3$)	405.178	Lasso ($\lambda = 10^3$)	405.178
Lasso ($\lambda = 10^5$)	404.823	Lasso ($\lambda = 10^5$)	404.823
Lasso ($\lambda = 10^7$)	399.023	Lasso ($\lambda = 10^7$)	399.023
Lasso ($\lambda = 10^9$)	392.469	Lasso ($\lambda = 10^9$)	392.469

TABLE 3 Training Time

Using all parameters		Using only parameters selected by Lasso	
Algorithm	**Training (seconds)**	**Algorithm**	**Training (seconds)**
Linear Regression	0.30	Linear Regression	0.08
M5P	3.10	M5P	1.58
REP Tree	0.56	REP Tree	0.17
SVM	417.41	SVM	164.96
SVM2	391.69	SVM2	205.65

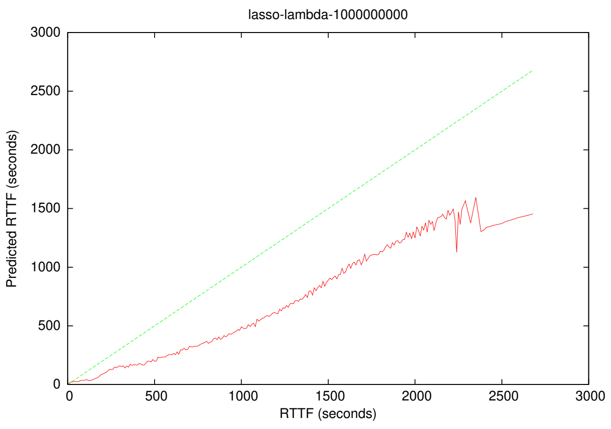
our expectation: the too aggressive regularization phase has lost some information related to (secondary-order) anomalies, and therefore provides an accuracy which is reduced. We note, anyhow, that the accuracy drop is not much elevate, and therefore a strong regularization can be viable in case the end user wants to trade accuracy off timeliness of model generation—we will shortly discuss the impact of regularization on model generation time.

An interesting preliminary result from this experimentation is that REP-Tree shows the best accuracy in both scenarios. In comparison with REP-Tree, M5P increases the error in the order of 10%. All other ML methods show higher errors. We note that this could be due to the fact that both REP-Tree and M5P divide the model space in smaller portions, and evaluate for each portion a different linear approximation. These could be regarded, anyhow, as two of the best candidate models for now.

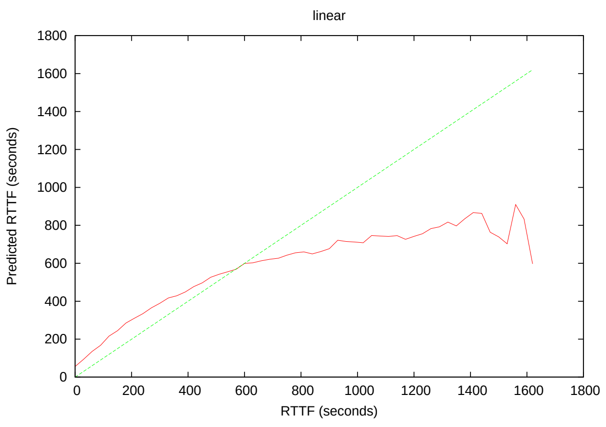
The final effect of regularization on the training time can be assessed by the data in Table 3, which shows the time (in seconds) required to construct a prediction model starting from all the features or only the ones selected by the regularization phase. As expected, the effect of regularization is that of significantly reducing the total time required to build a model. Again, the user can decide if this is a suitable trade-off for their application—or also if a smaller λ value should be selected. By the results, the best models (in terms of generation time) are Linear Regression, REP-Tree and M5P.

To assess in a different way the accuracy of the generated prediction models, we present a graphical representation of their behavior in Figure 3, this time only in the scenario where all the parameters are used for model generation. The six plots in Figure 3 show, for each ML algorithm, how much the generated model deviates from the ground truth. In particular, we show on the x-axis the actual RTTF, while the y-axis shows the RTTF as predicted by each model. The 45°-slope green line is the ground truth—intuitively, RTTF increases as we get father from the failure point. Indeed, the failure point is at the origin. Therefore, these plots tell, for each actual RTTF value on the x-axis, what is the corresponding RTTF value predicted by each model generated with the different algorithms (the red lines).

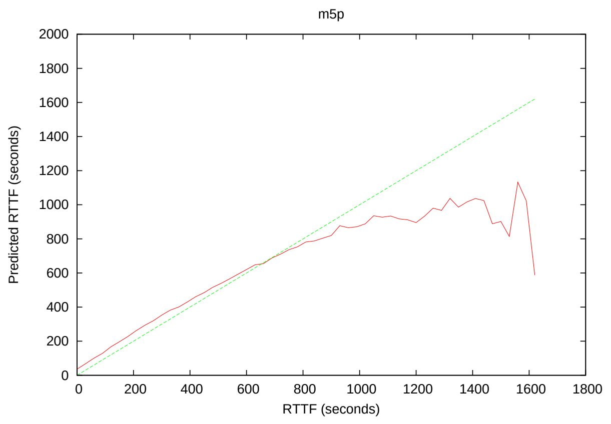
In general, we observe that prediction errors are higher when we are farther away from the failure point. We explain this behavior of the model by the fact that, when the accumulation of anomalies is reduced (i.e., we are far from the failure point) the possible runtime dynamics of the system are very varied. On the other hand, when the failure point is approaching, we observe a similar degradation of system features (e.g., the system incurs thrashing due to high usage of the swap space). The OCES data collection and model generation infrastructure is therefore able to capture



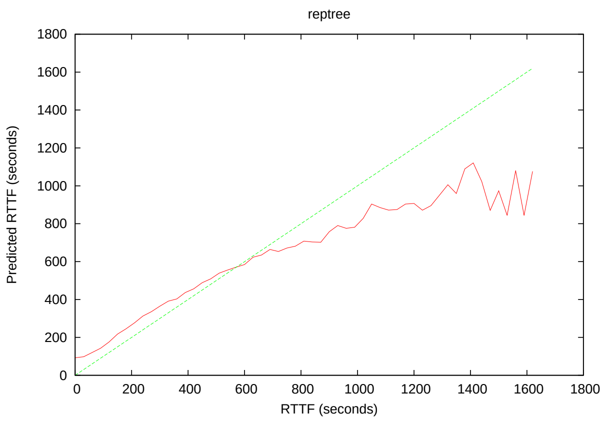
(a) Lasso ($\lambda = 10^9$)



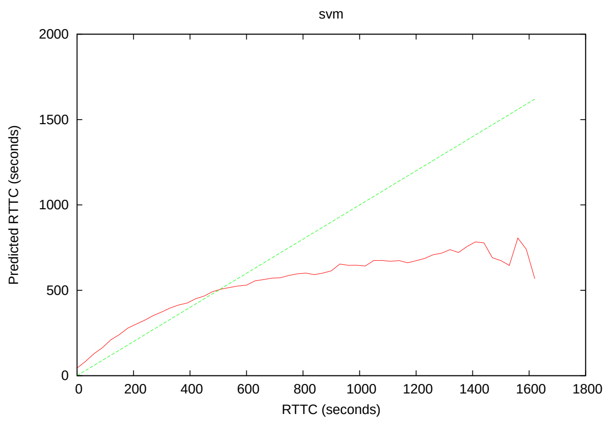
(b) Linear Regression



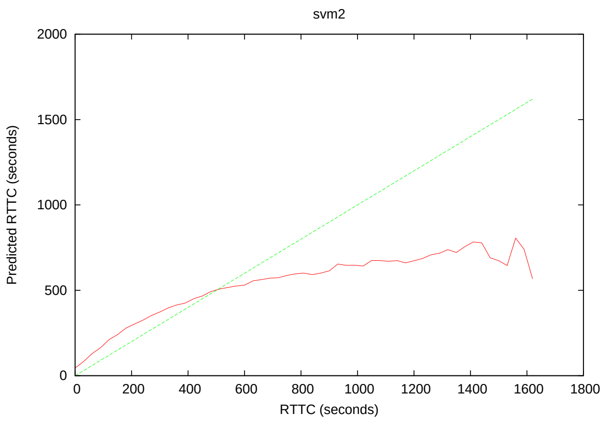
(c) M5P



(d) REP-Tree



(e) SVM



(f) LS-SVM

FIGURE 3 Fitted Models using all Parameters

these dynamics. However, we note that a higher prediction error is not a significant problem when we are far from the failure point, because (provided that the threshold T is safely tuned) the likelihood that a prediction error will trigger a rejuvenation phase is low. Except when using Lasso as the prediction model, the prediction error becomes very low when the actual RTTF value (on the x-axis) is lower than 600 seconds. We consider 5 minutes to be more than enough to perform a clean rejuvenation of VMs, in practical environments.

Also the results in Figure 3 confirm that REP-Tree and M5P are the best candidates in terms of accuracy. For all the consideration on the performance on prediction models (also taking in account the generation time), we have used M5P in the latter part of our experimentation. We have also used the models generated when relying on all the features in the datasets.

6.3 | Evaluation of Monitoring and Rejuvenation Capabilities

To analyze the behavior of OCES in a single cloud region, we have conducted two sets of experiments, in two different scenarios. The first experimental scenario involves only 2 VMs, so the VMC in the region can only trigger a rejuvenation phase without having the possibility to redirect client requests to a different set of running instances. In the second scenario, a total of 3 couples (for a total of 6 VMs) is used, thus allowing the VMC to redirect client requests to other instances. In the first scenario, VMs are subject to both memory leaks and non-terminated threads. In the second scenario, the first couple of VMs is subject to both memory leaks and non-terminated threads, the second couple is only subject to non-terminated threads, and the third couple is only subject to memory leaks. We note that, in the latter scenario, the rejuvenation process for each about-to-fail VM is triggered independently of the other couples. As a rejuvenation action, we chose VM rebooting, since it is suitable to the aim of our experimental study. Indeed, the effect of rebooting is to return the VM in its staring state, thus also restarting all the processes of the application on that machine. Obviously, this leads to kill all unterminated threads and to remove memory leaks of the application.

In both scenarios, we have set the threshold $T = 300$ seconds. Every time that an instance is rejuvenated, the probability distribution used to generate software anomalies (in the form of memory leaks and unterminated threads) is randomly generated from a Poisson process. As for memory leaks, the size of the leaked memory buffers is randomly selected between 10 Kb and 1 Mb upon each generation of a memory leak. We have used 32 concurrent clients in the first scenario, and 64 in the second one.

The first scenario has been run without interruption for one week. Figure 4 shows the data related to a short time window extracted from the whole experiment. We show the trend of various system-level metrics, as captured by the software probes, namely the number of active threads, free memory, used swap memory, and (total) CPU usage. Additionally, we report the response time measured by placing software probes in the emulated web browsers which generate client requests—they have been installed in a separated and dedicated VM—and the predicted RTTF for the VM that was activated upon each switching.

The results show that, as expected, the accumulation of anomalies leads to a continuous decrease in free memory, with a subsequent increase in the usage of swap. This is an additional indication that the outcome of the regularization process, discussed in Section 6.2, delivers reliable results. Similarly, the number of active threads grows. A direct effect of the anomalies on the end users is observed in the response time, which grows proportionally with the accumulation of anomalies. This can be regarded as a side effect of the thrashing state in which the VM operates in degraded mode, due to the accumulation of the anomalies. In the plots, we identify with vertical dotted red lines the time instant at which the rejuvenation procedure is initiated. After a rejuvenation, thanks to the overtaking of the healthy VM, the available resources bring the application in an anomaly-free state. The predicted RTTF immediately grows, and the response time (as observed by the users) drops down. These results show that our approach is effective at keeping low the response time observed by the users (under a cap of 4.5 seconds), proving the viability of our approach. During the whole experiment, less than 500 client requests were not served by the running VMs due to the rejuvenation phase (triggered by expired user-defined timers), which can be regarded as an important result given the objectives of our work.

A for the second scenario, in Figure 5 we report for brevity the results for a limited time window. Given the stochasticity in the generation of the anomalies, each VM suffers from a different accumulation phenomenon. Again, we indicate with vertical dotted red lines the time instants in which the rejuvenation process (for one of the VMs in the pool) is initiated. The results show that the VMC is able to manage independently the VMs without any loss of timeliness in the rejuvenation process initiation, and that the framework is able to effectively react to differentiated distributions of anomalies.

6.4 | Evaluation of Multiple Region Management Policies

In this section we report the results of the evaluation of the different policies we presented in Section 5 used to manage the virtual resources distributed different cloud regions. We present the results with all the three policies using all the three regions of our experimental environment.

Figure 6 reports, for each policy, the evolution of the RMTTF for every region, the fraction f_i for each region, and the average response time measured by all clients. By the results, we can draw the following observations:

- Policy 1 does not make RMTTF values converge. Indeed, the RMTTF values stabilize to different values. Further, the values of f_i are subject to oscillations.
- Policy 2 exhibits a better performance. The values of the RMTTF converge quite quickly, and f_i shows less-oscillating values. We associate this better behavior with the explicit estimation, proper of Policy 2, of the available resources on each cloud region.
- Policy 3 converges better than Policy 1, however the values of RMTTF and f_i are less stable with respect to Policy 2.

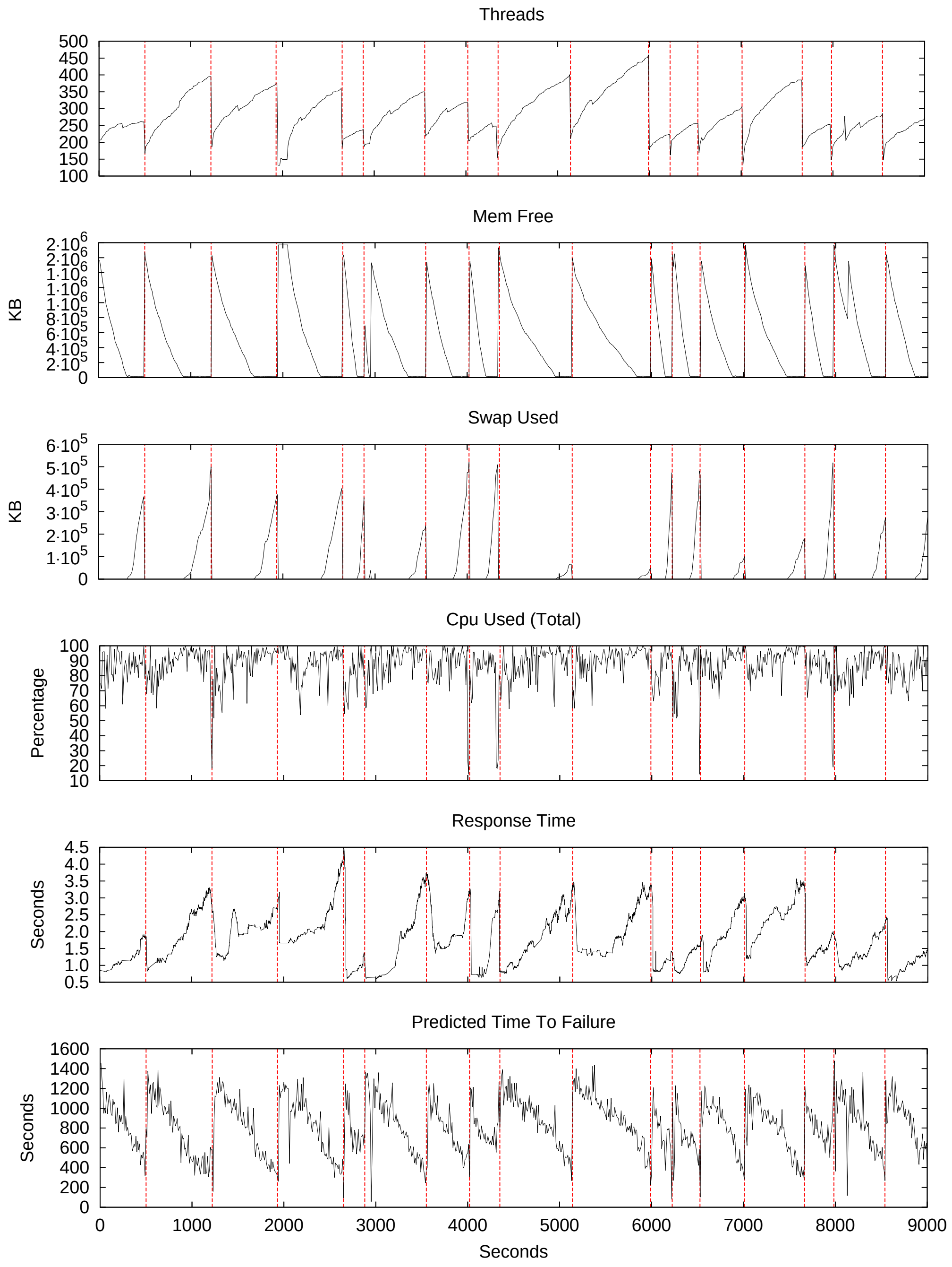


FIGURE 4 System features, response time and predicted RTTF with 2 VMs.

The oscillation of the values of f_i have the drawback, in the distributed scenario, that client requests could be repeatedly migrated across different regions. Depending on the organization of the database backing the application, this phenomenon could lead to sub-optimal performance levels—we recall that we have do not explicitly deal with consistency of databases, thus requiring the application to rely on a (possibly distributed) database infrastructure for data consistency. Overall, we can conclude that Policy 1, based on sensible routing, is more suitable for less-heterogeneous environments. On the contrary, when heterogeneity is very high, the quickest convergence and the most stable results are provided by Policy 2, which is based on explicit available resources accounting. Exploration approaches, such as Policy 3, are similarly valid, yet they can suffer more from their intrinsic randomness.

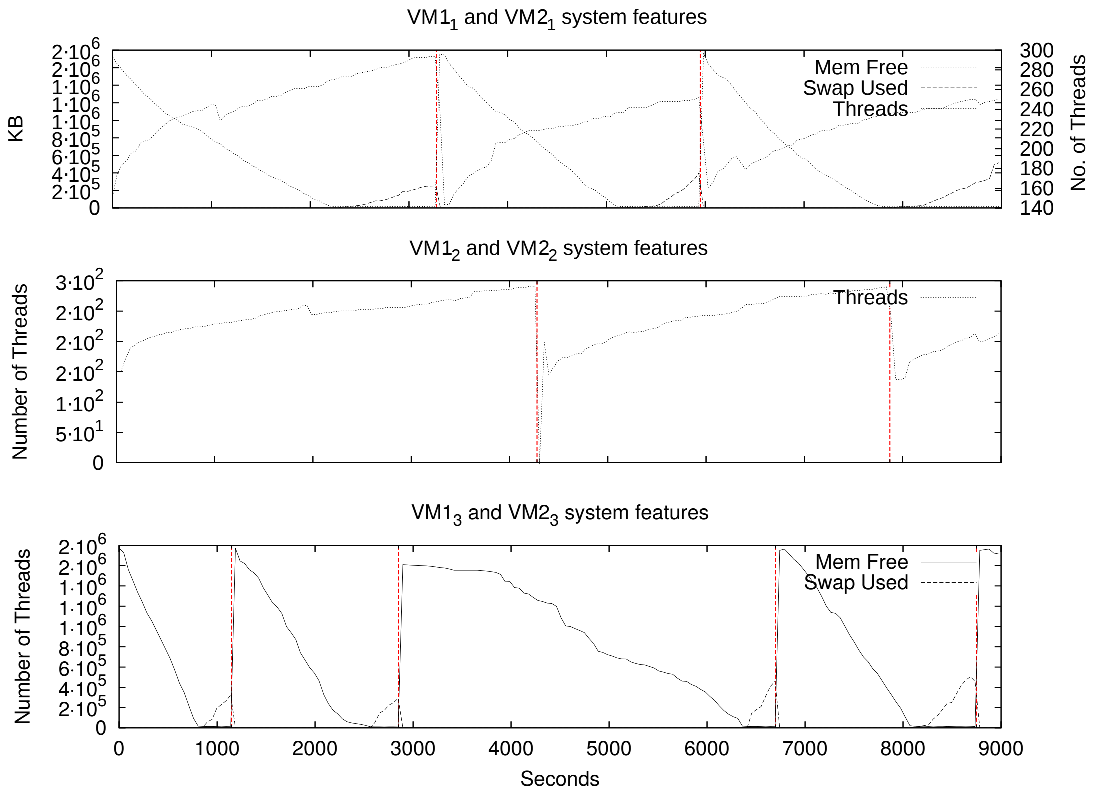


FIGURE 5 System features, response time and predicted RTTF with 6 VMs.

6.5 | Evaluation of Multi-Cloud Management in Dynamic Scenarios

We extended our study on the behavior of OCES with multiple cloud regions with three additional experiments which reproduce dynamic scenarios regarding the composition of the cloud environment and the workload profile. In the first one, we show the effects on the RMTTF when a cloud region joins the system. The results are reported in Figure 7(a). In this experiment, we have started our framework using only Region 1. After about 50 seconds (the time is marked by a vertical dotted red line in the figure) Region 2 joins the system. At this point, the two controllers start exchanging information. By the figure, we can see that, before Region 2 joins the system, the RMTTF of Region 1 is about 1000 seconds (also, it is equal to the global RMTTF). After Region 2 joins the system, the RMTTF of Region 1 and Region 2 start to converge to comparable values. Overall, the global RMTTF increases. This shows that OCES is able to distribute the global workload between the two regions, in a way to level out the RMTTF of the two cloud regions and to reduce the global VMs failure rate.

In Figure 7(b), we show the results of an experiment in which Region 1 and Region 2 have been used to assess the behavior of the load balancing approach of OCES. Emulated web browsers are connected only to Region 1, which is the only region managed by OCES at the beginning of the experiment. All client requests are therefore initially processed by Region 1. After about 22 minutes (first vertical dotted red line in the figure), Region 2 joins the system, and OCES Framework starts balancing the workload. Around minute 55 (second vertical dotted red line in the figure), Region 2 leaves the system. By the results of this experiment with churn of regions, we note that when Region 2 joins, its RMTTF is higher than Region 2—all VMs in this region are in a perfectly-healthy state, because no anomaly has accumulated yet—and the forward probability rapidly increases for Region 2. At minute 34, the system reaches an equilibrium: the forward probability is constant (about 0.5 per region), and the RMTTF of the two regions becomes almost the same. After that Region 2 leaves, the RMTTF of Region 1 returns to the same low value as in the first part of the experiment. This synthetic experiment shows that OCES is able to take benefit from the elastic nature of cloud computing, and that the client request forwarding approach is able to reduce the likelihood of a rejuvenation, provided that enough virtual resources are present in the system.

The third experiment tries to assess the behavior of OCES when the workload varies over time and is overall imbalanced. In the results reported in Figure 7(c), the load generated by the emulated web browsers connected to Region 1 changes over time, while it is constant for Region 2. After minute 22 (first vertical dotted red line in figure), the client request generation rate increases from 300 to about 700 requests per second. This highest value is reached around minute 75. Initially, there are only Region 1 and Region 3 (Region 2 joins the system after about 90 minutes—second vertical dotted red line in figure). We note that, while the incoming request rate increases, the RMTTF of both regions decreases. However, although the incoming request rate increases only for Region 1, OCES is able to keep balanced the RMTTF of both cloud regions. Finally, when

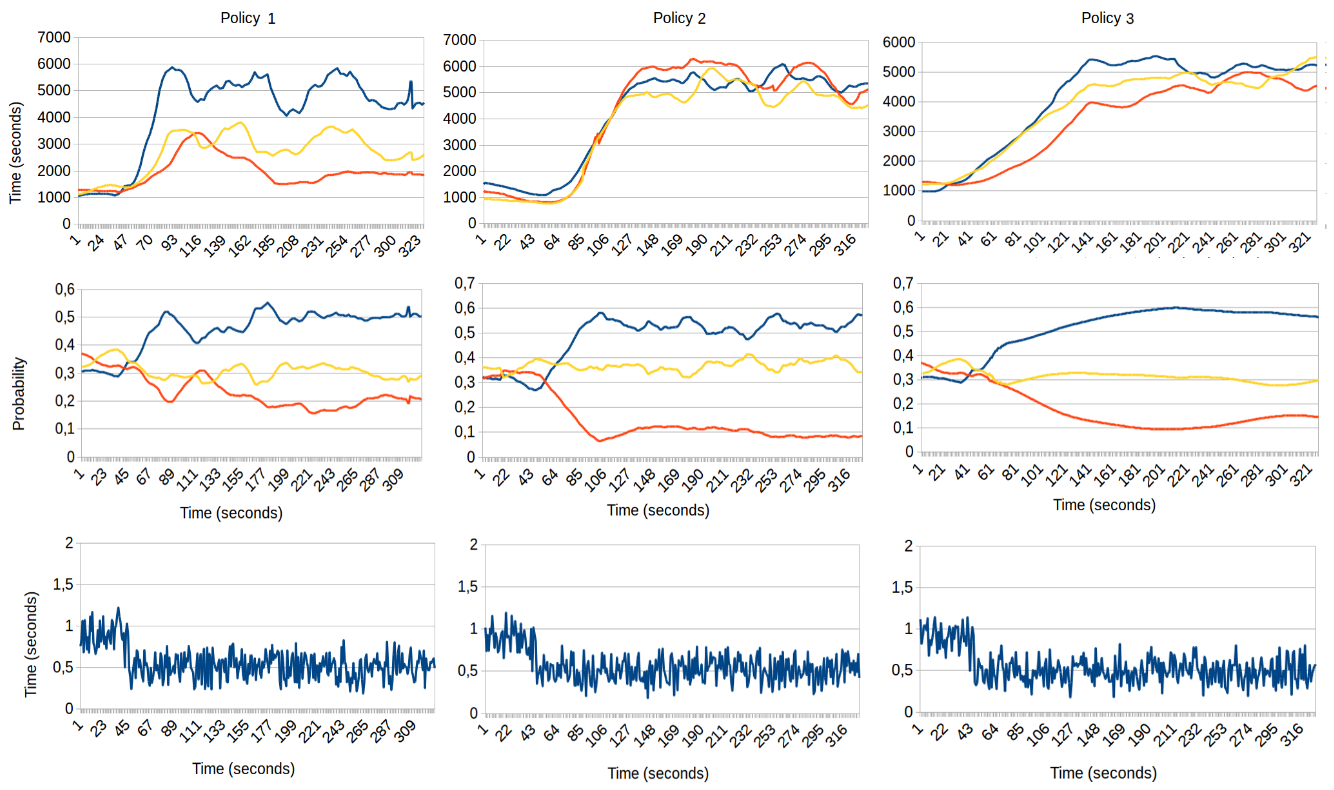


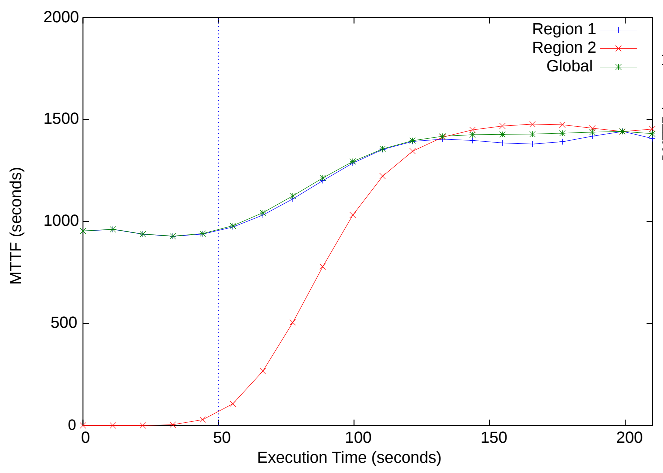
FIGURE 6 Results using 3 regions. The first row shows RMTTF, the second row shows the workload factor f_i, the third row shows the response time measured by the clients of the system.

Region 2 joins the system, the RMTTF of the all cloud regions starts to increase, as long as the forward probabilities changes and reaches an equilibrium. This shows that OCES has been able to cope with different incoming client request rates in different cloud regions, as well as with variations due to scale up/scale down of the system.

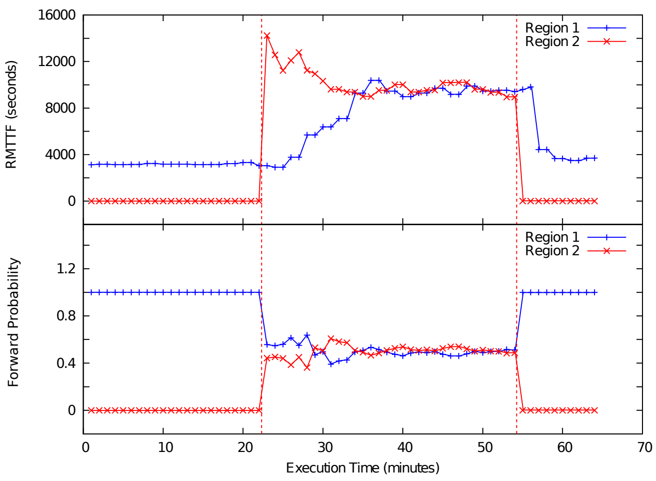
7 | RELATED WORK

Cloud computing has become a fundamental asset for both the Academy and the Industry. Moreover, the overly-connected world in which we live, requires applications to be available at all costs, independently of how complex their development lifecycle is[28]. These aspects have generated a new spark in the research literature to decide methodologies and tools to support a continuous operation of applications, in face of possible problems they (or their supporting runtime environments) may experience[29].

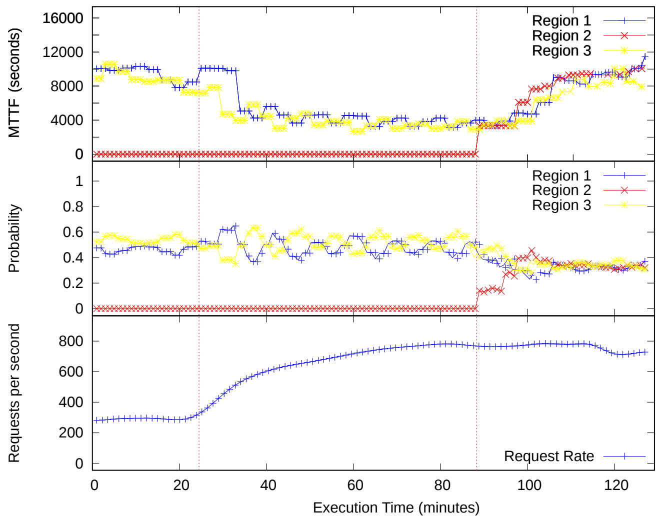
A number of techniques proposed to mitigate the problem of software anomalies are based on the static analysis of code of the application[6]. Basically, they analyse the source code or the compiled code of the application, without requiring the its execution. As an output, they provide a list of warnings, highlight potential weakness and/or provide useful software metrics, which may help programmers to address the potential anomalies of the application. Various automatic static analysis tools have been proposed[6]. Siavvas et. al.[30] provide a literature review of tools targeting two important sub-fields of static analysis, i.e. (i) prediction of vulnerable software components, and (ii) optimum checkpoint recommendation. As for the first type of tools, they aim at prioritizing testing and security inspection in order to find potential vulnerabilities in specific software components, and to help programmers to fix them. They are typically based on vulnerability prediction models which rely on ML techniques. Differently, optimum checkpoint recommendation aims at identifying and suggesting to programmers the optimum locations the in source code of the applications where checkpoints should be placed. An approach based on a mathematical model to estimate the optimum number of instructions of an application that should be executed between two consecutive checkpoints is described by Siavvas et. al.[19]. The model is designed for applications subject to failures, which are addressed by re-executing a portion of the application instructions starting from the last checkpoint. The goal of the model is the minimization of the total execution time of the application by identifying the optimal checkpoint placement. Siavvas et. al. also propose[31] another



(a) RMTTF variation when a new cloud region joins the system



(b) MTTF and Forward Probability (2 regions, 500 req/sec)



(c) Request rate, MTTF, Forward Probability (3 regions, var. rate

FIGURE 7 Experimental Results when using 3 Cloud Regions.

interesting application of statistical analysis. They use statistical analysis to assess the relationship between the so-called Technical Debt (TD)[32] and software security. TD is a measure inspired by the financial debt and can be used to estimate long-term quality problems of software. The authors show that an increase in the TD of software applications could indicate an increase of their vulnerabilities and vice versa. Consequently, TD may help programmers to evaluate the potential vulnerability degree of their applications a then to try to mitigate them. Overall, all the solutions we described above offer techniques aimed at the identification and the removal of the potential the software anomalies of an application. Differently, the approach we presented in this article is a distinct way to address this kinds of problems, since it targets the improvement of availability and performance of applications at the execution time. Accordingly, it can be considered a complementary approach, that can be used, e.g., before that software anomalies are identified and removed in applications already in the production stage, or when their removals are unfeasible or too costly.

Several proposals have addressed the problem of detecting upcoming failures in cloud-hosted systems. Sahoo et al.[33] base their detection on time series and rule-based classification, taking as input a set of logs collected over a year of execution. Adamu et al.[34] rely on ML-based prediction models to detect hardware failures in real-time cloud environments. Liu et al.[35] propose a composite model-based approach to detect aging of cloud resources, in the context of anomalies. A regression-based transaction model, which reflects the resource consumption model of the application has been used by Cherkasova et al.[36], which notably does not require to explicitly probe the system as we do. Yin et al.[37] rely on multi-layer neural networks in order to identify the state in which virtual instances are running, so as to determine the current quality of service. An approach based on experiments to extract interaction-related failure indicators has been proposed by Li et al.[38]. The calculation of derived metrics has been also explored[39], although without the final goal of building multiple prediction models. Other approaches[40][41][42] have explicitly dealt with the detection of upcoming failures, also in the context of hybrid clouds[43]. Overall, these works show the importance of promptly detecting what we call "failure points" in virtualized applications—as discussed in the article, any QoS/QoE is handled as a failure point in our approach. Differently from these contributions, we do not concentrate on a specific technique to detect an upcoming failure. Rather, we have proposed an extensible framework which allows the user to specify what should be considered a failure point (thus increasing the applicability of our proposal) which can autonomously derive a set of different prediction models, enabling the user to select the best-suited one.

A different research line has dealt with the construction of reliable prediction models when the data to be used for training is unreliable[44] or when bias and variance in the data could cause inaccurate prediction[45]. This important research line is orthogonal to the one carried in this article. Indeed, we believe that such results could be easily embedded in our framework, provided its extremely modular nature.

There have been also been several proposals to detect anomalous behavior of applications running in distributed (cloud-based) environments targeting cybersecurity[46,47]. While we do not explicitly tackle security aspects in OCES, in principles the modular configuration of the failing condition could be extended to account for metrics which exhibit anomalies also at the level of security. Nevertheless, we believe that a more promptly response than the one required to rejuvenate VMs for performance/availability issues should be enforced, thus possibly requiring an extension of the proposal in this article.

Autonomicity, self-* properties and real-time detection are also approaches which are well studied in the literature[48,49,50,51,52], also with a special focus on resource monitoring[53,54]. Capacity allocation has also been dealt with[55], although from the perspective of economic savings by choosing the best-suited instances. Our proposal retains all the capabilities of the aforementioned works in the literature, although we propose a general software architecture which embodies all aspects of the applications, independently of their deployments.

As for load balancing in the cloud, a number of techniques and tools have been presented in literature. A survey that classifies various techniques has been published by Afza et al.[56]. The basic goal of load balancing techniques in the cloud is ensuring that no (virtual) resource is either overloaded or under-loaded compared to other ones. Load balancing can take into account various metrics, such as throughput, response time, resource utilization, availability and energy consumption[57,58,59,60]. One major challenge with cloud applications is distributing the workload over a dynamic computing environment, also considering that the application deployment may span different and heterogeneous cloud regions. Load balancing can be applied also to optimize the workload distribution in hybrid clouds, i.e. composed of both private and public cloud regions. As an example, an approach based on a mathematical model designed to optimize the distribution over local and remote cloud services is proposed by Gelembe et al.[60]. Specifically, it uses a model that takes into account both performance and energy-related issues.

Compared to the techniques that have been presented in literature, the novelty that we introduced with OCES is that it takes into account a new parameter in the load balancing decision process, i.e. the RMTTF of the different cloud regions. In our experimental study, we evaluated three different load balancing policies that use this parameter. Independently of the specific results of the three policies, we showed that it is possible to use RMTTF within load balancing techniques to mitigate the problem of high failure frequencies of some cloud regions compared to other ones. Ultimately, this paves the way for investigating the integration of this kind of metric also within other load balancing techniques presented in literature.

8 | CONCLUSIONS

In this article, we explored a new holistic approach to cope with the problem of software anomalies in cloud-based applications. In particular, we presented OCES, a framework that implements this new approach and offers specific tools to manage applications deployed over heterogeneous cloud regions distributed at geographical scale. OCES offers a new way to handle run-time failures due to software anomalies, and represents a complementary solution to other approaches which aim at detecting and removing software errors, such as static analysis. Indeed, OCES focuses on the effects of anomalies at run-time rather than on their root cause, and promptly performs repairing actions. With our framework, we evaluated a proactive way to exploit software rejuvenation to prevent system failures, and we introduced a new load balancing approach, based on the predicted MTTF and RMTTF, to cope with the high failure rates in different and heterogeneous cloud regions.

The experimental results we observed with a real-word setting show that, by putting together the potential advantages of proactive rejuvenation and load balancing, we can be able to both prevent system failures and control the operation response time, keeping it below a given threshold. Also, they show that is possible to prevent highly unbalanced failure/rejuvenation rates in different cloud regions, despite the heterogeneity of regions in terms of anomalies occurrence and available computing resources. Ultimately, the results demonstrate an improvement of the overall system performance, thanks to the overall reduction of the average global response time of operations, and an improvement of the system availability perceived by the user, thanks to a reduction of the global failure rates.

Overall, our experience demonstrates the viability of our approach. We note that, on the one hand, OCES addresses various tasks related to the problem of run-time management of software anomalies, and proposes a set of techniques and tools cope with them. On the other hand, it represents only one of the possible implementations of our approach. Therefore, also different techniques and/or tools could be used and evaluated to address the different tasks. Accordingly, we believe that our experience opens the way to further studies to improve, extend or complement, also with additional techniques and tools, the approach we proposed.

ACKNOWLEDGMENTS

The research presented in this article has been supported by the European Union via the EC funded project PANACEA, contract number FP7 610764.

References

1. Pertet Soila, Narasimhan Priya. *Causes of Failure in Web Applications*. CMU-PDL-05-109: Carnegie Mellon University; 2005.
2. Kolettis Nick, Fulton N Dudley. Software Rejuvenation: Analysis, Module and Applications. In: *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*FTCS:381–390IEEE Computer Society; 1995; Washington, DC, USA.
3. IEEE . IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*. 2010;:1–23.
4. Anu Vaibhav, Hu Wenhua, Carver Jeffrey C, Walia Gursimran S, Bradshaw Gary. Development of a human error taxonomy for software requirements: A systematic literature review. *Information and Software Technology*. 2018;103:112–124.
5. De Felice Fabio, Petrillo Antonella, eds.*Theory and Application on Cognitive Factors and Risk Management - New Trends and Procedures*. InTech; 2017.
6. Gosain Anjana, Sharma Ganga. Static Analysis: A Survey of Techniques and Tools. In: Mandal Durbadal, Kar Rajib, Das Swagatam, Panigrahi Bijaya Ketan, eds. *Advances in Intelligent Systems and Computing* AISC, vol. 343: Springer, New Delhi 2015 (pp. 581–591).
7. Torquato Matheus, Vieira Marco. An Experimental Study of Software Aging and Rejuvenation in Dockerd. In: *2019 15th European Dependable Computing Conference*EDCC:1–6IEEE; 2019.
8. Taherizadeh Salman, Stankovski Vlado. Dynamic Multi-level Auto-scaling Rules for Containerized Applications. *The Computer Journal*. 2019;62(2):174–197.
9. Pellegrini Alessandro, Di Sanzo Pierangelo, Avresky Dimiter R., Sanzo Pierangelo Di, Avresky Dimiter R.. A Machine Learning-based Framework for Building Application Failure Prediction Models. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*DPDNS:1072–1081IEEE; 2015.
10. Neville-Neil George V.. The observer effect. *Communications of the ACM*. 2017;60(8):29–30.
11. Forster Florian, Harl S. collectd–the system statistics collection daemon https://collectd.org/2012.
12. Shicong Meng, Ling Liu. Enhanced Monitoring-as-a-Service for Effective Cloud Management. *IEEE Transactions on Computers*. 2013;62(9):1705–1720.
13. Tibshirani Robert. Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society, Series B*. 1994;58:267–288.
14. Alpaydin Ethem. *Introduction to Machine Learning*. The MIT Press; 3rd ed.2014.
15. Wang Yong, Witten Ian H. Inducing Model Trees for Continuous Classes. In: *Procedings of the 9th European Conference on Machine Learning*:128–137; 1997.
16. Chipman Hugh A, George Edward I, Mcculloch Robert E. Extracting Representative Tree Models From a Forest. In: *IPT Group, IT Division, CERN*:363–377; 1998.
17. Cortes Corinna, Vapnik Vladimir. Support-Vector Networks. *Machine Learning*. 1995;20(3):273–297.
18. Suykens J A K, Vandewalle J. Least Squares Support Vector Machine Classifiers. *Neural Processing Letters*. 1999;9(3):293–300.

19. Siavvas Miltiadis, Gelenbe Erol. Optimum checkpoints for programs with loops. *Simulation Modelling Practice and Theory*. 2019;97:101951.

20. Gelenbe Erol, Sevcik Ken. Analysis of Update Synchronization for Multiple Copy Data Bases. *IEEE Transactions on Computers*. 1979;C-28(10):737–747.

21. Chesnais A., Gelenbe E., Mitrani I.. On the modeling of parallel access to shared data. *Communications of the ACM*. 1983;26(3):196–202.

22. Silva Luis Moura, Alonso Javier, Torres Jordi. Using Virtualization to Improve Software Rejuvenation. *IEEE Transactions on Computers*. 2009;58(11):1525–1538.

23. Pellegrini Alessandro, Sanzo Pierangelo Di, Avresky Dimiter R.. Proactive Cloud Management for Highly Heterogeneous Multi-cloud Infrastructures. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*:1311–1318IEEE; 2016.

24. Di Sanzo Pierangelo, Pellegrini Alessandro, Avresky Dimiter R. Machine Learning for Achieving Self-* Properties and Seamless Execution of Applications in the Cloud. In: *Proceedings of the Fourth IEEE Symposium on Network Cloud Computing and Applications*NCCA:51–58IEEE Computer Society; 2015.

25. Wang Lan, Gelenbe Erol. Adaptive Dispatching of Tasks in the Cloud. *IEEE Transactions on Cloud Computing*. 2015;pp(1):1–1.

26. Bezenek Todd, Cain Trey, Dickson Ross, et al. Characterizing a Java implementation of TPC-W. In: *Proceedings of the Third Workshop On Computer Architecture Evaluation Using Commercial Workloads*; 2000.

27. Smith Wayne D. *TPC-W: Benchmarking an ecommerce solution.* 2000.

28. Pietrantuono Roberto, Russo Stefano. Software Aging and Rejuvenation in the Cloud: A Literature Review. In: *Proceedings of the 2018 IEEE International Symposium on Software Reliability Engineering Workshops*ISSREW:257–263IEEE; 2018.

29. Boutaba Raouf, Salahuddin Mohammad A., Limam Noura, et al. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*. 2018;9(1):16.

30. Siavvas Miltiadis, Gelenbe Erol, Kehagias Dionysios, Tzovaras Dimitrios. Static Analysis-Based Approaches for Secure Software Development. In: Gelenbe Erol, Campegiani Paolo, Czachórski Tadeusz, et al. , eds. *Communications in Computer and Information Science* Communications in Computer and Information Science (CCIS), vol. 821: Springer, Cham 2018 (pp. 142–157).

31. Siavvas Miltiadis, Tsoukalas Dimitrios, Janković Marija, et al. An Empirical Evaluation of the Relationship between Technical Debt and Software Security. In: Konjović Zora, Zdravković Milan, Trajanović Miroslav, eds. *Proceedings of the 9th International Conference on Information Society and Technology*ICIST:199–203Society for Information Systems and Computer Networks; 2019.

32. Cunningham Ward. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*. 1993;4(2):29–30.

33. Sahoo R. K., Oliner A. J., Rish I., et al. Critical event prediction for proactive management in large-scale computer clusters. In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*KDD:426ACM Press; 2003; New York, New York, USA.

34. Adamu Hussaini, Mohammed Bashir, Maina Ali Bukar, Cullen Andrea, Ugail Hassan, Awan Irfan. An Approach to Failure Prediction in a Cloud Based Environment. In: *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*:191–197IEEE; 2017.

35. Liu Jing, Tan Xueyong, Wang Yan. CSSAP: Software Aging Prediction for Cloud Services Based on ARIMA-LSTM Hybrid Model. In: *2019 IEEE International Conference on Web Services*ICWS:283–290IEEE; 2019.

36. Cherkasova Ludmila, Ozonat Kivanc, Mi Ningfang, Symons Julie, Smirni Evgenia. Anomaly? Application Change? or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change. In: *Proceedings of the 2008 International Conference on Dependable Systems and Networks*IEEE Computer Society; 2008.

37. Yin Yonghua, Wang Lan, Gelenbe Erol. Multi-layer neural networks for quality of service oriented server-state classification in cloud servers. In: *Proceedings of the International Joint Conference on Neural Networks*; 2017.

38. Li Luyi, Lu Minyan, Gu Tingyang. Extracting Interaction-Related Failure Indicators for Online Detection and Prediction of Content Failures. In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops*ISSREW:278–285IEEE; 2018.

39. Simeonov Dimitar, Avresky Dimiter R.. Proactive Software Rejuvenation Based on Machine Learning Techniques. In: Avresky DimiterR., Diaz Michel, Bode Arndt, Ciciani Bruno, Dekel Eliezer, eds. *Cloud Computing* Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, vol. 34: Springer Berlin Heidelberg 2010 (pp. 186–200).

40. Alonso Javier, Belanche Lluis, Avresky Dimiter R.. Predicting Software Anomalies Using Machine Learning Techniques. In: *Proceedings of the 2011 IEEE 10th International Symposium on Network Computing and Applications*NCA:163–170IEEE Computer Society; 2011.

41. Cotroneo Domenico, Natella Roberto, Pietrantuono Roberto, Russo Stefano. Software aging and rejuvenation: Where we are and where we are going. In: *Proceedings - 2011 3rd International Workshop on Software Aging and Rejuvenation, WoSAR 2011*:1–6; 2011.

42. Zhang Hui, Jiang Guofei, Yoshihira Kenji, Chen Haifeng. *Proactive Workload Management in Hybrid Cloud Computing.* 2014.

43. Han Yiming, Chronopoulos A T. A Resilient Hierarchical Distributed Loop Self-Scheduling Scheme for Cloud Systems. In: *IEEE 13th International Symposium on Network Computing and Applications*; 2014.

44. Zhao Zilong, Cerf Sophie, Birke Robert, et al. Robust Anomaly Detection on Unreliable Data. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*:630–637IEEE; 2019.

45. Yan Yongquan. Variance analysis of software ageing problems. *IET Software.* 2018;12(1):41–48.

46. Aditham Santosh, Ranganathan Nagarajan, Katkoori Srinivas. LSTM-based memory profiling for predicting data attacks in distributed big data systems. In: *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017*; 2017.

47. Chawathe Sudarshan S.. Monitoring IoT networks for botnet activity. In: *NCA 2018 - 2018 IEEE 17th International Symposium on Network Computing and Applications*; 2018.

48. Cui Lei, Li Bo, Li Jianxin, Hardy James, Liu Lu. Software Aging in Virtualized Environments: Detection and Prediction. In: *Proceedings of the 18th International Conference on Parallel and Distributed Systems*ICPADS:718–719IEEE; 2012.

49. Gulenko Anton, Wallschläger Marcel, Schmidt Florian, Kao Odej, Liu Feng. A System Architecture for Real-time Anomaly Detection in Large-scale NFV Systems. In: *Procedia Computer Science*; 2016.

50. Gonzalez Nelson Mimura, De Brito Carvalho Tereza Cristina Melo, Miers Charles Christian. Multi-phase proactive cloud scheduling framework based on high level workflow and resource characterization. In: *Proceedings - 2016 IEEE 15th International Symposium on Network Computing and Applications, NCA 2016*; 2016.

51. Drăgan Ioan, Fortiş Teodor-Florin, Iuhasz Gabriel, Neagul Marian, Petcu Dana. Applying Self-* Principles in Heterogeneous Cloud Environments. In: *Cloud Computing*Computer Communications and Networks:255–274Springer International Publishing; 2017.

52. Mijumbi Rashid, Hasija Sidhant, Davy Steven, Davy Alan, Jennings Brendan, Boutaba Raouf. Topology-Aware Prediction of Virtual Network Function Resource Requirements. *IEEE Transactions on Network and Service Management.* 2017;.

53. Araujo Jean, Braga Céfanys, Belmiro Neto José, Costa Adriano, Matos Rubens, Maciel Paulo. An Integrated Platform for Distributed Resources Monitoring and Software Aging Mitigation in Private Clouds. *Journal of Software.* 2016;11(10):976–993.

54. Palmieri Roberto, Di Sanzo Pierangelo, Quaglia Francesco, Romano Paolo, Peluso Sebastiano, Didona Diego. Integrated monitoring of infrastructures and applications in cloud environments. In: Alexander Michael, D'Ambra Pasqua, Belloum Adam, et al. , eds. *Euro-Par 2011: Parallel Processing Workshops* Lecture Notes in Computer Science (LNCS), vol. 7155: Springer, Berlin, Heidelberg 2011 (pp. 45–53).

55. Ardagna Danilo, Casolari Sara, Colajanni Michele, Panicucci Barbara. Dual time-scale distributed capacity allocation and load redirect algorithms for cloud systems. *Journal of Parallel and Distributed Computing.* 2012;72(6):796–808.

56. Afzal Shahbaz, Kavitha G.. Load balancing in cloud computing – A hierarchical taxonomical classification. *Journal of Cloud Computing.* 2019;8(22).

57. Arulkumar V, Bhalaji N.. Load balancing in cloud computing using water wave algorithm. *Concurrency and Computation: Practice and Experience.* 2019;.

58. Semmoud Abderraziq, Hakem Mourad, Benmammar Badr, Charr Jean-Claude. Load balancing in cloud computing environments based on adaptive starvation threshold. *Concurrency and Computation: Practice and Experience.* 2020;32(11).

59. Gelenbe Erol, Finkel David, Tripathi Satish K.. Availability of a distributed computer system with failures. *Acta Informatica.* 1986;23(6):643–655.

60. Gelenbe Erol, Lent Ricardo, Douratsos Markos. Choosing a Local or Remote Cloud. In: *Second IEEE Symposium on Network Cloud Computing and Applications*NCCA:25–30IEEE; 2012.

61. Quinlan J Ross. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1993.

9 | APPENDIX

As discussed in Section 3.4, the basic set of ML methods of OCES includes six linear and non-linear methods, specifically: Linear Regression, M5P, REP-Tree, Lasso as a Predictor, Support-Vector Machine (SVM), and Least-Square Support-Vector Machine. For completeness, in this section, we provide a brief description of these ML methods. For a detailed description, the reader can refer to the references we report for each of them.

Linear Regression[14] is an approach for modeling the relationship between a (scalar) dependent variable and one or more explanatory variables. Given a data set defined as $\{y_i, x_{i1}, \ldots, s_{ip}\}_{i=1}^{n}$ of n statistical units, a linear regression model assumes that the relationship between the dependent variable y_i and the p-vector of regressors x_i is linear. This relationship is modeled through a disturbance term or error variable ε_i—an unobserved random variable that adds noise to the linear relationship between the dependent variable and regressors. Thus the model takes the form:

$$y_i = \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^{\mathrm{T}} \beta + \varepsilon_i, \quad i = 1, \ldots, n \tag{12}$$

M5P[15] is a decision tree with the possibility of linear regression functions at the nodes. First, a decision-tree induction algorithm is used to build a tree, but instead of maximizing the information gain at each inner node, a splitting criterion is used that minimizes the intra-subset variation in the class values down each branch. The splitting procedure in M5P stops if the class values of all instances that reach a node vary very slightly, or only a few instances remain. Second, the tree is pruned back from each leaf. When pruning, an inner node is turned into a leaf with a regression plane. Third, to avoid sharp discontinuities between the subtrees a smoothing procedure is applied that combines the leaf model prediction with each node along the path back to the root, smoothing it at each of these nodes by combining it with the value predicted by the linear model for that node.

REP-Tree[16] is a fast decision tree learner. It builds a decision/regression tree using information gain/variance and prunes it using reduced-error pruning (with backfitting). Only sorts values for numeric attributes once. Missing values are dealt with by C4.5's method[61] of using fractional instances.

Lasso as a Predictor[13] generates, depending on a parameter λ, a vector β whose elements are the weights of the vector x_j, which minimizes the objective function shown in Equation (2). The application of Lasso as a Predictor is grounded on the same mathematics used for Lasso Regularization, but the goal is different. In fact, while during the regularization process we are interested in determining the β vector, during the prediction we exploit the already-computed β vector to evaluate the prediction model, which is expressed as a closed-form equation.

Support-Vector Machine[17] constructs a hyperplane or a set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data point of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

Least-Square Support-Vector Machine[18] Given a training set $\{x_i, y_i\}_{i=1}^{N}$ with input data $x_i \in \mathbb{R}^n$ and corresponding binary class labels $y_i \in \{-1, +1\}$, the SVM classifier, according to Vapnik's original formulation[17], satisfies the following conditions:

$$\begin{aligned} w^{\mathrm{T}}\phi(x_i) + b &> 1, \quad \text{if} \quad y_i = +1, \\ w^{\mathrm{T}}\phi(x_i) + b &< -1, \quad \text{if} \quad y_i = -1. \end{aligned} \tag{13}$$

which is equivalent to $y_i \left[w^{\mathrm{T}}\phi(x_i) + b\right] \geq 1, \quad i = 1, \ldots, N$ where $\phi(x)$ is the non-linear map from original space to the high (and possibly infinite) dimensional space.

How to cite this article: Pierangelo Di Sanzo, Dimiter R. Avresky, and Alessandro Pellegrini (2020), Autonomic Rejuvenation of Cloud Applications as a Countermeasure to Software Anomalies, *Software: Practice and Experience, 2020;00:1–6.*