Biswas et al. Journal of Cloud Computing: Advances, Systems and Applications (2017) 6:29
DOI 10.1186/s13677-017-0100-5

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

A hybrid auto-scaling technique for clouds processing applications with service level agreements

Anshuman Biswas[1*], Shikharesh Majumdar[1], Biswajit Nandy[1] and Ali El-Haraki[2]

Abstract

This research focuses on the automatic provisioning of cloud resources by an intermediary enterprise. This enterprise provides a virtual private cloud for a single client enterprise by using resources from a public cloud. The intermediary cloud provider is controlled by a broker that uses techniques to dynamically control the number of resources used by the client enterprise. The research presents a hybrid auto-scaling technique based on a combination of a reactive approach and a proactive approach to scale resources based on user demand. The primary goal of this auto-scaling technique is to achieve a profit for the intermediary enterprise while maintaining a desired grade of service for the client enterprise. The second goal of the technique is to reduce costs for the single client enterprise. The technique supports both on-demand requests and requests with service level agreements (SLAs). This paper describes the auto-scaling algorithms associated with the hybrid technique and includes a discussion of system design and implementation experience for a prototype system. A detailed performance analysis based on simulations and measurements made based on the prototype is presented.

Keywords: Hybrid auto-scaling on clouds, Resource allocation, Dynamic resource provisioning, Scheduling with SLAs, Resource management on clouds, Machine learning

Introduction

The cloud environment offers an organization the ability to shift its IT operations from a traditional capital expenditure (CAPEX) model [1], in which an organization procures dedicated hardware that depreciates over a period of time to an operational expenditure (OPEX) model, which allows the use of a shared cloud infrastructure with facilities that allow paying for only the amount of resources used by an organization without needing to procure dedicated hardware ahead of time. This is an important feature of cloud computing, which offers users the ability to acquire resources by paying a fixed price per unit time for a cloud resource. This pricing model is also known as an on-demand or a pay-as-you-go pricing model [2]. The practice is similar to that of utility bills, where payment is made only for the resources that have been used. Public cloud providers such as Amazon with its S3 and EC2 services use the on-demand pricing model [3] when charging for various cloud resources. In a typical cloud environment, an organization acquires its IT infrastructure from a cloud provider for a period and then runs their applications inside that infrastructure. This type of service offered by cloud providers is known as Infrastructure as a Service (IaaS).

Another important characteristic of cloud computing is elasticity. Elasticity allows an organization to scale rapidly to meet customer demands. Elasticity is formally defined as the degree to which an application or system deployed inside a cloud infrastructure, autonomously adapts its capacity to workload demands over time [4].

The ability to provide on-demand computational resources appears to be unlimited from the perspective of an organization using a cloud provider's services. This offers the organization the ability to potentially purchase any number of resources at any time. Each organization that rents out resources from a cloud provider is known as a tenant of the cloud provider. Additionally, cloud

* Correspondence: anshuman@sce.carleton.ca
[1]Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada
Full list of author information is available at the end of the article

providers support multitenancy by allowing multiple organizations to rent their cloud resources. The cloud provider enables sharing of resources due to its ability to virtualize physical resources, which helps increase its revenue. Multitenancy also enables an increase in utilization and efficiency for cloud resources that otherwise may not be utilized to their maximum capacity. The challenge for a cloud provider offering its infrastructure to its customers is to maintain a quality of service (QoS) that guarantees metrics such as a low response time, high throughput and high service availability. Typically, to provide such a QoS guarantee, the cloud provider needs to maintain a service level agreement (SLA). Failure to comply with the metrics will violate the terms agreed upon in the SLA and potentially cause the cloud provider to lose business [1].

Additionally, the organization may be interested in ensuring that its requests, specific to the applications deployed on the infrastructure, are completed in a timely manner. One way to accomplish this is to introduce SLAs for the workload provided by cloud users to allow user requests to specify an earliest start time and deadline for the requests. These types of requests are also referred to as advance reservation (AR) requests in the literature [5]. AR requests are important features of clouds and distributed systems [6]. The different properties of an AR request are described in more detail in "Performance evaluation" section. Associating a deadline with requests processed by a cloud has been considered in several works (see [7] and [6] for example) and is receiving a great deal of attention from researchers. The authors of [8] and [9] evaluate auto-scaling mechanisms which consider both user performance requirements and pricing for a workload characterized by jobs with deadlines. Note that in addition to jobs with deadlines, the technique discussed in this paper can handle jobs without deadlines referred to as On Demand (OD) request by associating an arbitrarily large deadline with the respective requests that is larger than the deadline of any AR. An OD request is executed on a best effort basis [10]. Moreover, the matchmaking and scheduling algorithms used in this paper handles requests for which the arrival time and the earliest start time coincide by setting the earliest start time to be equal to the time of arrival of the request. Whether or not a request is an OD and irrespective of the earliest start times of requests the system can be profitable. A discussion of profitability and the conditions that need to be satisfied for achieving a profit are discussed in "Profitability and cost analysis" section. As indicated earlier, the proposed system is designed to work with both advanced reservation AR and OD requests. In the experiments evaluating the auto-scaling techniques discussed in this paper, a workload comprising 80% AR and 20% OD requests has been used. The auto-scaling techniques described in the paper work with any given mix of ARs and ODs including a workload with 100% OD requests. Without deadlines, the proposed system will accept all the requests and schedule them on a best effort basis.

This approach works even when the start time and the request arrival time coincide by either locating a resource that can accommodate the new request immediately. If no suitable resources are found, the system either rejects the request, or if the request is deemed profitable by the broker, a new resource is acquired. If a new resource is acquired, there is an overhead to start the resource. In such a case, if the deadline cannot be met after delaying the start time of the execution of the request, the request is rejected.

Auto-scaling helps prevent over-provisioning which leads to higher cost savings and higher resource utilization. Instead of forcing an organization to acquire resources before deploying an application, resources may be incrementally added or removed as the demand for the application fluctuates over time. Moreover, instead of overprovisioning an application infrastructure to meet peak workload demands, an organization may employ auto-scaling to control expenditure during low usage periods while still being able to satisfy peak demands during high usage periods. Auto-scaling also increases resource utilization and decreases the idle time of resources compared with an overprovisioned infrastructure, in which the system resources remain idle and power is unnecessarily consumed [11].

In cloud auto-scaling mechanisms, metrics that are monitored to decide when to perform auto-scaling operations include CPU utilization, memory utilization, and bandwidth usage. Such infrastructure-level performance metrics are good indicators for conveying the load on a resource. Based on certain predefined thresholds for these metrics, a resource may be acquired or released. Determination of this threshold requires extensive experimentation with the system. Moreover, as noted in [12], basing an auto-scaling decision on a single performance metric may not be adequate, as choosing an auto-scaling metric incorrectly may cause a system to operate with more or less resources than desired. Choosing appropriate performance metrics and determining precise threshold values for these metrics is not a trivial task and becomes more challenging if the workload pattern is continuously changing [13]. Currently, auto-scaling mechanisms that use thresholds do not take application performance into consideration when adding a fixed number of resources. This problem of choosing the appropriate threshold becomes more challenging when the application model used in the cloud environment is complex and the metrics are limited to the utilization of computational elements. Although the

system introduced in this paper auto-scales computational resources, the techniques described do not use traditional metrics such as CPU utilization to perform the auto-scaling operations. It relies on two approaches when deciding to auto-scale resources. The first approach is reactive in the sense that it reacts to changes in the system state. It utilizes a threshold-based mechanism using grade of service (GoS) criteria. These GoS criteria help in ascertaining whether the QoS guarantee, set forth by the entity using this system, is being complied with. The second approach is to utilize a prediction mechanism that relies on performing an analysis of past workload to determine the future resource demands and the necessity of scaling resources based on the predicted future workload.

Another factor that other auto-scaling mechanisms described in the literature often overlook is the time required to boot a VM instance. Though instance acquisition requests can be made at any time, instances are not available to users immediately. Such instance start-up lag is typically due to the time required to, for e.g., find the right spot for the requested instances in a cloud data centre, download the specified OS image, boot the virtual machine, and finish network setup [14]. Based on our previous experiences [15] and that of other researchers [14], it could take as long as 5 min to start a t1 micro instance in an Amazon cloud, and such startup lag can vary dynamically over time. Clients must take into account this instance startup time factor when procuring instances from a cloud provider, especially if their workload comprises of AR requests.

This paper focuses on employing a broker that runs within an organization known as the Intermediary Enterprise (IE) and auto-scales the number of resources acquired from a public cloud provider. The resources are required to execute AR and OD requests sent by another organization known as a Single Client Enterprise (SCE) or the User. SCE is an entity comprising of multiple users that generate requests that need to be serviced. SCE sends all these requests to IE on behalf of these users. IE is charged by the public cloud provider to rent resources, and IE, in turn, charges SCE to execute requests to recover its operational expenditure. The broker auto-scales the resources to increase the profit earned from its operation while attempting to reduce the cost incurred by SCE (explained in more detail in "Pricing" section). The reduction in cost is justified by comparing with a system where the requests would be scheduled by SCE without broker intervention. This type of a system is defined in "Alternative systems" section.

A system in which IE buys resources by the hour from a public cloud provider and sells it to the user by seconds can be attractive to both users as well as IE. It is attractive to the users when the user requests do not utilize the full hour of a resource offered by a cloud provider and therefore the user incurs a lower cost when charged by seconds in comparison to buying the resource from the public cloud that charges the user by the hour. The profit for IE results from re-using the newly idled resource for running another user request that has arrived on the system that will lead to a higher earning than the hourly charge it pays to the public cloud service provider. Note that such a sharing of resources among requests does not give rise to any resource contention because each resource is used by one request at a given point in time. The scheduler allocates a resource to a request for a given interval of time. Only when the user request finishes execution, the resource is allocated to another request that is scheduled to execute on this resource next. This system is applicable in all such environments where multiple user requests need to be serviced.

In addition to an independent IE providing service to its SCE client, such a system may also be adopted by an IT department of an organization that accepts user requests from other departments. The IT department would act as a broker while each department would function as an SCE. The IT department would procure resources from a public cloud provider and pay by the hour while charging the users by the second. The profits earned by the IT department could be utilized to keep it self-sufficient and operational. Two performance metrics have been used in this paper. They are *Broker profit* and *User cost*, that form an important basis to measure the efficacy of the system, which strives to achieve a high broker profit as well as a reasonable user cost.

To investigate the behaviour of the system, an event-driven simulation that tests the validity of the proposed algorithm was initially developed. Once the simulator led to encouraging results, a prototype of the system that utilizes resources from the Amazon public cloud was developed. This paper focuses on presenting the results obtained from the simulator and the prototype of the proposed system that implements a novel hybrid auto-scaling technique. The key contributions of this paper are summarized.

- A novel hybrid auto-scaling algorithm that uses a price model that can lead to an increase in profit for a broker (intermediary enterprise) and a reduction in user cost at the same time.
- A framework using the hybrid auto-scaling algorithm, with the ability to address SLA-driven AR requests as well as OD requests.
- A detailed performance analysis based on both simulation and measurements made on a proof-of-concept prototype of the framework, focusing on broker profit and user cost for a system subjected to

various combinations of system and workload parameters.
  - Key insights into system behaviour and performance are presented.
  - The performance analysis demonstrates how an intermediary enterprise hosting a broker can earn profit while decreasing the user cost in comparison to a system in which users directly acquire their resources from the public cloud.
- Comparison of the proposed hybrid framework for auto-scaling with frameworks that only scale purely proactively [16] or reactively [17].
  - To showcase the advantages of using a hybrid approach, the paper compares such traditional auto-scaling mechanisms (proactive and reactive) with the proposed hybrid algorithm.

The rest of the paper is organized as follows. Related work is discussed in "Related work" section whereas "System overview" section provides a system overview for the proposed hybrid system. "Broker architecture" section describes the broker architecture while "Reactive and proactive auto-scaling" section describes the auto-scaling algorithms. "Experimental prototype" section discusses the experimental platform while "Performance evaluation" section presents the performance evaluation and observations from the experiments conducted. The concluding remarks are presented in "Conclusions" section.

Related work

This section discusses a representative set of existing work on auto-scaling systems in cloud environments. "General auto-scaling approaches" section discusses general auto-scaling approaches, whereas "Resource management frameworks" section discusses resource management frameworks. "Hybrid auto-scaling approaches" section addresses work that highlights auto-scaling performed via hybrid approaches.

General auto-scaling approaches

Traditional techniques for auto-scaling are easy to deploy and use a mechanism to manage the number of resources assigned to an application hosted on a cloud platform (e.g., [18]). The ease-of-use of these rules make them appealing to cloud users. However, creating the rules for auto-scaling requires an effort from the application manager, who needs to select a suitable performance metric or logical combination of metrics and the values of several additional parameters, mainly thresholds that are used for scaling resources. Thresholds are the key to ensuring that the rules implemented are correct. In particular, as described in [19]. The ease-of-use of these rules make them appealing to cloud users. However, creating the rules for auto-scaling requires an effort from the application manager, who needs to select a suitable performance metric or logical combination of metrics and the values of several additional parameters, mainly thresholds that are used for scaling resources. Thresholds are the key to ensuring that the rules implemented are correct. In particular, as described in [20] thresholds need to be carefully tuned in order to avoid oscillations in the system. Oscillations occur if the auto-scaling system keeps changing the number of resources in order to satisfy the threshold rules. To prevent this problem, [20] introduces a cool-down or calm period, a time during which no further scaling decisions can be committed, which is implemented once a scaling action has been carried out. Conditions in the rules are usually based on a single or at most two performance metrics, such as the average CPU load of the VMs, the response time, or the input request rate. The research presented in this work performs auto-scaling based on the expected profit for an entity known as a broker. Since a broker controls the resources and acquires them only when they are deemed profitable, our work also makes use of a threshold for the blocking ratio as another criterion to ensure that the client enterprises have a guarantee regarding the minimum proportion of requests scheduled. A system known as SCAling, which uses an analytic model that monitors a global metric when auto-scaling, is described in [21]. SCAling is an approach for auto-scaling driven by SLAs. The global metric is used to manage cloud elasticity, i.e., to ensure that the number of used instances increases seamlessly during demand spikes to maintain performance and decreases automatically during demand lows to minimize costs. Our work also uses the grade of service as an SLA, acting as a global objective based on which a user can drive resource acquisition.

Resource management frameworks

The authors of [22] present a resource management framework called Anchor where resource management policies are separated from the management mechanism. The highlight of Anchor is a new many-to-one stable matching theory that efficiently matches VMs with heterogeneous resource needs to servers. Amazon offers a service known as CloudWatch [23] for monitoring applications running on their instances. CloudWatch allows a system-wide visibility of resource utilization, application performance and operational health which enables users to spot trends and make automated decisions based on the user's cloud environment. Haizea [12] is a resource a lease manager with a scheduler module that offers dynamic resource allocation features via the lease scheduler. However, [22] and [23] do not address the scheduling of ARs. Additionally, none of the works discuss proactive resource management.

Aneka [6] is a framework for developing distributed applications on the cloud. It provides developers with an API for managing resources and expressing the business logic of applications by using programming abstractions. Aneka provides tools monitoring a deployed infrastructure, scaling resources based on the completion times of requests. If the current set of resources is unable to complete a newly arriving request before its deadline, additional resources may be acquired. Resources are released when certain threshold conditions are met. The authors of [24] propose a system that employs a cloud brokerage service to reserve resources from a public cloud provider instead of acquiring them on-demand. Their broker reserves resources at a bulk price that is lower than the on-demand price. The broker shares the resources with multiple users while offering them per-second billing. The goal of this paper is to reduce user cost. The approach discussed in [24] does not ARs when provisioning resources. In addition, the approaches in both [6] and do not consider broker profit as a factor when deciding whether to acquire or release resources. Finally, these approaches do not employ proactive auto-scaling via the technique introduced in this paper. Amazon's CloudWatch also utilizes target metrics specified by its users to trigger actions upon reaching their thresholds. Proactive resource management may attempt to predict the future workload when making scaling decisions [25]. Moore et al. [25] uses a similar technique for resource provisioning. However, resource provisioning may also be achieved without predicting the future workload as discussed in [26], where the authors propose a technique for dynamically adapting thresholds to meet QoS targets.

Hybrid auto-scaling approaches

The authors of [25] propose a real-time cloud capacity framework, offering a hybrid elasticity controller employing both reactive rule-based and proactive model-based elasticity mechanisms in a coordinated manner. The hybrid controller examines the scale-up condition which, in a purely reactive auto-scaling environment, is used to acquire new resources, and builds incrementally updateable predictive models to enable a system to proactively scale up before this condition is met. The research presented in [25] demonstrates that, compared to a purely reactive controller, or a purely proactive controller, the hybrid controller can make better provisioning decisions. The authors of [27] propose an alternate hybrid control mechanism that maintains the history of the session arrival rate seen. The authors assume a multitiered web application with a web server and a database server that needs to be auto-scaled. Provisioning is done prior to each hour based on the worst load seen in the past. Their workload predictor is based on a technique that uses past observations of a workload to predict the peak demand that will be seen over a period of T time units. The proactive provisioning technique allocates capacity to address the worst-case load by using the tail of the arrival rate distribution to predict peak demand. The reactive controller acts on short time scales to increase the resources allocated to a service in case the predicted value is less than the actual load that arrives. The authors of [28] propose a hybrid scaling technique that utilizes reactive rules for scaling up (based on CPU usage) and a regression-based approach for scaling down. After a fixed number of intervals in which the response time is satisfied, they calculate the required number of application-tier and database-tier instances using a polynomial regression. The authors propose a reactive system for scale-up operations and a proactive system for scale-down operations.

This paper presents a hybrid approach that also improves performance compared to a purely proactive or reactive approach. However, the scaling decisions are based primarily on profit for the auto-scaling controller whereas the authors of [25], [27] and [28] do not incorporate cost-awareness in their scaling decisions. Moreover, this paper addresses AR requests that allow users to specify an SLA individually for each request. Researchers include scaling metrics monitor workload arrival rate and a QoS condition for auto-scaling, both of which are also considered in this work. To the best of our knowledge, none of the existing papers have discussed auto-scaling based on broker profit. A broker-based approach is used by other researchers, some of whom refer to the broker as a resource manager. This research proposal makes use of a novel differential pricing mechanism for end users to achieve a profit for the intermediary enterprise (using a broker).

System overview

This section presents an overview of the system and of the different components of the broker and how the broker interacts with the SCE and the public cloud provider. The section is divided into two subsections. The first subsection (Auto-scaling techniques) explains the different auto-scaling techniques, while the second (Pricing) presents a cost model to explain different pricing mechanisms used in this paper.

An overview of the system is presented in Fig. 1. As explained in "Introduction" section, IE is responsible for handing requests from SCE. IE uses the broker which controls resources acquired from the public cloud to form a "private" cloud which is also referred to as a Virtual Private Cloud (VPC) [29]. Ideally, the broker runs in one of the instances of the public cloud acquired by the intermediary enterprise or in an external location controlled by the intermediary enterprise. The broker

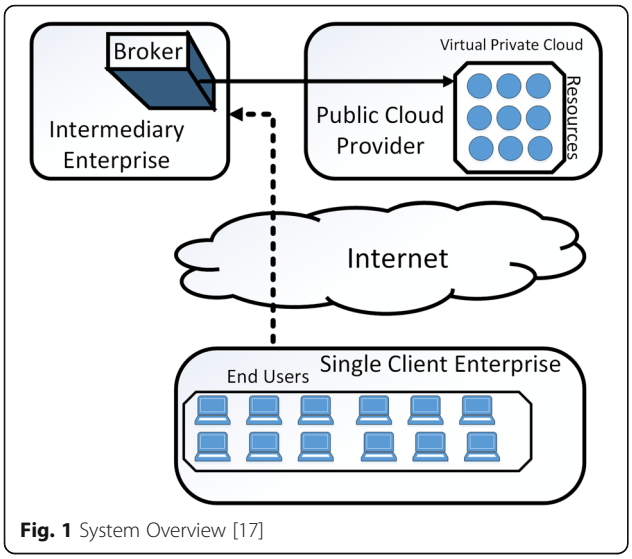

Fig. 1 System Overview [17]

may run in an alternate location as well but must possess the ability to communicate with the resources of the VPC. To keep the resources private, they are connected to a private subnet while the broker executes on the public subnet (accessible via an external network such as the Internet). This allows the VPC to remain secure in case an intruder tries to access the private resources. The users inside SCE send requests to the broker over the Internet. The broker allows the users to execute their jobs on instances acquired by the broker. The broker charges the users of the private cloud a higher price per unit time than that for the cloud providers to earn a profit. However, the broker charges users per second rather than on an hourly basis. This ensures that users pay only for the time during which their request is executed. The goal for this broker is to maximize profit for the private cloud provider (IE) while attempting to reduce the cost for the client enterprise (SCE) that comprises the users for the private cloud. This reduction in cost is determined by comparing with a system in which, instead of renting instances from the intermediary cloud provider, the user rents resources directly from a public cloud provider. This system is discussed in more detail in "Alternative systems" section and its performance is compared with the performance of the proposed system.

A broker-based approach allows an abstraction to be provided to users by removing the necessity to manage resources in a cloud infrastructure. It allows users to submit requests to an application and retrieve the results later. This approach also allows a broker to share resources between different user requests. A broker has two responsibilities: handling the matchmaking and scheduling of requests and managing the auto-scaling of resources. A broker dynamically controls the number of resources using a threshold-based mechanism and utilizes a prediction system based on a machine learning approach to predict future workloads. The primary criterion used in this paper is to auto-scale resources based on profit accrued by an IE. In addition, the paper uses another criterion for acquiring resources based on a GoS specified by the client enterprise. Although the system described in this paper uses the blocking ratio to describe the GoS, the framework can be adapted to other metrics as well. The blocking ratio (B) is the proportion of requests that cannot be completed before the expiry of their deadlines and are therefore rejected by the system. The specified value of B, referred to as B_{spec}, which is provided by the user, is the desired value of B maintained by the broker and is stored in the GoS component. Similar GoS criteria have been used by other researchers [30]. A previous work [31] by the authors also dealt with auto-scaling resources in a cloud environment while monitoring the value of B. However, this paper utilizes the concept of auto-scaling based on broker profit. Resources are acquired when either a profit is earned by the broker after acquiring the additional resource necessary for satisfying the user request or when rejecting the request would violate the GoS criteria with B exceeding B_{spec}. Auto-scaling with profit and GoS was first introduced in a previous work [17] by authors addressing reactive auto-scaling.

Auto-scaling techniques

Auto-scaling techniques can be classified into two categories. Reactive rule-based methods define scaling conditions based on a target metric reaching some threshold and are offered by several cloud providers such as Amazon. Proactive auto-scaling approaches tend to be based on time series analysis, control theory, reinforcement learning, or queuing theory [25]. One strategy for achieving proactive auto-scaling is to use a workload predictor to determine the required number of resources to satisfy the predicted demand and acquire those resources when needed, based on the prediction. A third hybrid technique, proposed in this paper, combines the reactive and the proactive controllers. Proposals for this technique include using a proactive method and a reactive method to determine when to provision resources over a long time-scale (hours and days) and a short time-scale respectively. This paper combines both the reactive technique and proactive technique by invoking the reactive algorithm upon each request arrival and the proactive technique after a predetermined set of request arrivals.

Pricing

Cost is an important issue worth considering when using a cloud environment. The pricing of cloud services allows users to choose a cloud service provider that suits

their requirements, either reserving cloud resources in advance or buying those resources on-demand. A cloud provider is equipped with a price-setting mechanism which sets the current price for a resource based on market conditions, user demand, and the current level of utilization of the resource. Pricing can either be fixed or variable depending on the market conditions [32]. For fixed pricing, instances are charged by the hour. A fraction of an hour is counted as a whole hour. Therefore, part of the resource time remains unutilized if machines are shut down before a whole hour of operation. In addition to the full hour principal, clouds now usually offer various instance types, such as high-CPU and high I/O instances [2]. This research tackles the problem of addressing this unutilized time due to idling resources by allowing users to rent out resources by paying per second rather than per hour. This allows users to pay for only the time that the resource is utilized to satisfy their request. The broker earns a profit by allowing multiple requests to share the same resource, thereby minimizing the amount of idle time for each resource.

The public cloud provider charges the broker *c_pub* dollars an hour per resource. However, the broker charges the user *c_pvt* dollars per second. The first is referred to as the broker cost rate while the second as the user cost rate. These amounts need to be appropriately chosen such that the user is charged considerably more per hour than c_pub. If the value of c_pvt is too low, the broker may find situations where it runs into a loss. Since the total user cost incurred by the client enterprise is also the earnings for the intermediary enterprise provider (broker), the broker profit is calculated using the following equation:

$$\textbf{Broker profit}(\textbf{BP}) = \textbf{Total User Cost}(\textbf{UC}) - \textbf{Broker Cost}(\textbf{BC}) \tag{1}$$

Here, BC is the cost incurred by the broker when acquiring resources from the public cloud provider. BP may be calculated for any period and is denoted as BP^a and BP^p, signifying the actual (Experimental parameters) and predicted values (Proactive auto-scaling) respectively. The reason for having two values for broker profit is that the proposed system has a prediction stage in which resources are auto-scaled based on the predicted profit from a future workload. Hence, this predicted profit is referred to using the subscript 'p', whereas the actual profit accrued by the broker during its operation is referred to using the subscript 'a'. As indicated earlier, in addition to broker profit, this paper also considers the cost savings for users. The total cost saving for all requests sent by SCE is computed as the sum of savings achieved with all the requests that were processed by the system. The cost saving for a single request, with a service time of *c* seconds is the difference between the cost that would be incurred if the resource were procured directly from the public cloud provider and that incurred when it is acquired from IE:

$$\textit{Cost Savings for one request} = \left\lceil \frac{c}{3600} \right\rceil \times c\_pub - c \times c\_pvt \tag{2}$$

Consequently, for a given set of k requests with service times $\{c_1, c_2, \ldots, c_k\}$ seconds, the total cost savings is calculated as:

$$\textbf{Total Cost Saving}(\textbf{CS}) = \sum_{i=1}^{k} \left(\left\lceil \frac{c_i}{3600} \right\rceil \times c\_pub - c_i \times c\_pvt \right) \tag{3}$$

Broker architecture

This section describes the broker architecture along with its internal components. The section is divided into two sub-sections. The first sub-section, "Auto-scaling criteria" section explains the different auto-scaling criteria used when making auto-scaling decisions, while the second section, "Matchmaking and scheduling" section presents the techniques for matchmaking and scheduling a request based on the available resources.

As explained in "System overview" section, the broker is primarily responsible for the match-making and scheduling operations for an incoming request as well as auto-scaling a pool of resources. The following subsections describe the components of the broker system which is shown in Fig. 2. The system is divided into five distinct components. The first component is the *Request Handler* (RH) which accepts requests from SCE. The primary responsibility of RH is to receive requests from users and forward the requests to MMS which decides whether to accept or reject the request. Once the broker decides on a request, RH informs SCE of this decision. RH also triggers a chain of operations that may lead to

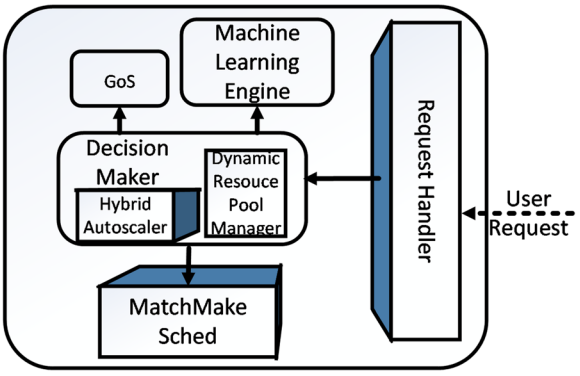

Fig. 2 Broker Architecture

the broker performing an auto-scaling operation (described in "Auto-scaling criteria" section). The second component known as *MatchMakeSched* (MMS) makes the decision to accept or reject a request and is discussed in more detail in "Matchmaking and scheduling" section. The third component is *Decision Maker* (DM) which manages resource provisioning. DM (described in "Reactive auto-scaling" section), which manages auto-scaling. DM uses the fourth component known as *Machine Learning Engine* (MLE), to predict future requests based on past workloads. Operation of MLE is further explained in "Proactive auto-scaling" section. The final component is the GoS module, which stores the grade of service set by SCE, which is responsible for maintaining the desired GoS level. This component is also described in "Reactive auto-scaling" section.

At any time, the broker controls a set of N resources in the VPC: $Res = \{Res_1, Res_2, \ldots Res_N\}$, where N changes dynamically with user demand. The broker is run in two separate threads and each thread runs independently of the other thread. The two threads decide on when auto-scaling operations are required without being influenced by the functions of the other thread. RH runs in Thread 1, listening for incoming requests from the User module is activated upon every request arrival. Thread 1 also runs a reactive auto-scaling component. This component decides to scale up an additional resource when the current resources controlled by the broker are not adequate to meet request deadlines. However, the new request must either accrue a profit for the broker or the GoS criteria must be unfulfilled by rejecting the request and allowing the value of B to cross B_{spec}. Thread 2 performs the proactive auto–scaling operations which are run after a predetermined interval. Both proactive and reactive auto-scalers are discussed in greater detail in "Reactive and proactive auto-scaling" section.

Each thread can trigger a change in the number of resources and thus facilitate the auto-scaling operations. Once requests have been scheduled, they cannot be cancelled by the scheduler. The decision to accept or reject the request is sent back to the user. Thus, the broker guarantees execution of a request that has been accepted by the system, by the specified deadline.

As explained in "System overview" section, the user is part of SCE, which comprises multiple users. The users are employees working for SCE. User requests are characterized by the following:

- *Arrival Time (AT)* – The time at which the request arrives at the system.
- *Earliest Start Time (EST)* – The earliest time at which the request can begin execution.
- *Service Time (ST)* – The time taken by the request to execute.
- *Deadline (DL)* – The time by which the request must complete its execution.
- Type (T) – The type of request: AR or OD.

Similar request characteristics for clouds have been used by other researchers [33].

An important component of the broker is the *Dynamic Resource Pool* Manager (DRPM). DRPM handles the task of communicating with the public cloud provider and acquiring or releasing resources. DRPM is devised to be interoperable with any number of cloud providers. In this paper, only one public cloud provider, i.e., Amazon Web Services (AWS) is used. DRPM offers a single Application programming interface (API) to the broker via which additional resources can be requested. Once a request is received, DRPM uses the API exposed by the required cloud provider to acquire the resources and transfer the details of the newly acquired resources to the broker. Hence, resources may be acquired from any public or private cloud provider, as long as this component can communicate with the cloud and has proper authorization to acquire resources from the cloud provider. In this paper, we assume that cloud providers can provide as many resources as requested by the broker.

Additional considerations may be required when public cloud providers limit the maximum number of resources that can be acquired by the broker on-demand. In such situations, DRPM may be configured to impose a limit on the number of resources that may be acquired. In this research, the limit is kept at a higher threshold than the number of resources desired by the system to avoid a circumstance in which the public cloud provider may refuse additional resources to be provided within a reasonable amount of time. If the number of resources acquired by DRPM reaches the limit imposed by a cloud service provider, resources will no longer be available to the broker. This may lead to requests being rejected and consequently a violation of the GoS set by SCE.

Auto-scaling criteria

Upon request arrival, the hybrid algorithm determines whether the incoming request is to be accepted (determined by MMS) and if accepted, whether additional resources need to be acquired to satisfy the request. Upon acceptance of an AR request, this request must be completed by a deadline included in the SLA. Upon request arrival, a resource is acquired if the following occurs:

(i) No available resource satisfies the request SLA
AND
The acquisition of the resource will lead to a profit
OR

(ii) Rejecting the request will lead to a B that is higher than the specified GoS.

Note: A resource is released after the timer set for that resource expires and if no further requests on that resource are scheduled. Further details on the resource timer are given in "Reactive and proactive auto-scaling" section.

After every k requests, the broker also runs the proactive algorithm to calculate the appropriate number of resources needed in the future to satisfy the workload demands. The proactive auto-scaling approach used in the proposed technique is based on a time series analysis [34] that uses different machine learning algorithms. MLE utilizes the functions of a software library called Weka [35]. The algorithm accomplishes the prediction by looking at past workload trends and predicts the workload in the future, thereby extrapolating the appropriate number of resources required to satisfy the demand while maintaining the GoS as well as accruing a profit for the broker. A set of resources is acquired if the following occurs:

(iii) No available resources satisfy one or more of the upcoming requests
AND
The upcoming predicted set of requests incurs a profit for the broker.
OR

(iv) Rejecting some of the upcoming predicted requests leads to a B that is higher than the specified GoS.

Matchmaking and scheduling

As explained in "System overview" section, when a request arrives at RH, the MMS component is invoked. Once a request enters MMS, a matchmaking algorithm determines a resource on which the request can be executed. A scheduling algorithm determines the order in which the requests allocated on a given resource are executed. The component uses a *first fit* (FF) strategy for matchmaking and an *Earliest Deadline First* (EDF) strategy for scheduling [36]. The architecture for MMS allows any other algorithm for matchmaking and scheduling to be used as an alternative. The first fit algorithm works by scanning resources acquired by the system sequentially and allocates the request to the first resource capable of handling the request. Each time a resource is considered by the matchmaker, the system runs the scheduling algorithm. With EDF, the requests allocated on a given resource are scheduled in accordance with their deadlines. A request with the shortest deadline is executed first followed by the one with the next shortest deadline and so on. If the newly arrived request can be allocated and scheduled on an existing resource such that its deadline can be met, the request is accepted. If the request cannot be scheduled, then the reactive auto-scaler is invoked.

Both the matchmaking and scheduling algorithms described are based on user estimates of job execution times. Such estimates are often error prone [7]. Handling these inaccuracies is a responsibility of the matchmaking and scheduling algorithm used by the system and not auto-scaling algorithm that this paper focuses on. Farooq et al. [37] have described methods for handling such inaccuracies by using a pre-scheduling mechanism based on overbooking and a run time exception handler. Such mechanisms can be incorporated in the matchmaker and scheduler deployed by MMS that is not the subject of attention of this paper. Using the auto-scaling techniques in conjunction with additional matchmaking techniques available in the literature, such as [37] forms an interesting direction for future research.

The operations of the reactive auto-scaler are described in "Reactive auto-scaling" section. OD requests are treated as a special case of AR requests in the system where the deadlines are set to a time that is the sum of the current time and a large interval such that there is sufficient time to guarantee the completion of the request before its (artificial) deadline. OD requests do not have SLA deadline requirements and hence may be scheduled after all AR requests have been executed. As a result, OD requests are scheduled similar to AR requests but may be moved around to allow AR requests to be executed prior to their execution. For space limitations, further details of these algorithms are not discussed but interested readers are referred to [10], which describes similar scheduling and matchmaking algorithms.

After a request is scheduled, MMS computes two additional characteristics for the request:

- Scheduled Start Time (SST) – The time at which the request is scheduled to begin execution.
- Scheduled Finish Time (SFT) – The time at which the request is scheduled to complete execution.

This component is also used by DM to simulate the operations for predicting upcoming requests and is discussed further in "Proactive auto-scaling" section. After every k requests, the next set of k future requests is predicted. These predicted requests are also submitted to MMS directly, instead of going through RH. However, MMS does not distinguish between actual requests and predicted requests and addresses both types of requests in the same manner.

Reactive and proactive auto-scaling

This section describes the reactive and proactive auto-scaling algorithms that are used by the hybrid broker.

The section is divided into two sub-sections. The first sub-section, "Reactive auto-scaling" section explains the inner workings of the reactive auto-scaler, while the next section, "Proactive auto-scaling" section presents the techniques used by the proactive auto-scaler.

As discussed in "Broker architecture" section, the system has two threads which execute simultaneously. The auto-scaling operation of resources is handled by both threads and is triggered separately. This section describes the approach taken by each thread in auto-scaling the resources controlled by the broker.

Reactive auto-scaling

Further details of the operations performed by thread 1 are herein discussed. The RH module is handled by thread 1, which handles a new request arrival, as explained before in "Broker architecture" section. This thread also uses MMS, DM as well as the GoS module. MMS, running inside thread 1 must determine whether a new request arrival may be satisfied by the set of existing resources already acquired by the broker. If no existing resource can satisfy the SLA requirements of the request, criteria (i) and (ii) described in "Auto-scaling criteria" section is used to determine whether an additional resource is to be acquired so that the request can be accepted. If no additional resource can be acquired, the request is rejected. For calculating the broker profit, the Eq. (1) provided in "Pricing" section is utilized. However, the equation is modified in Eq. (5) to accommodate the characteristics of a request *i*. Regarding this i^{th} request with a service time of c_i seconds, the Reactive Auto-scaler (RA) checks it against the following condition:

$$\mathbf{BP_i} > 0 \tag{4}$$

Where,

$$\mathbf{BP_i} = \boldsymbol{G} \times (\boldsymbol{c_i} \times \boldsymbol{c\_pvt}) - \left(\left\lceil \frac{\boldsymbol{c_i}}{\mathbf{3600}} \right\rceil \times \boldsymbol{c\_pub} \right) \tag{5}$$

Here, G is a constant greater than 1. G is introduced to account for the possibility of a resource generating a higher profit from future requests in the time remaining rather than considering only the i^{th} request when computing broker profit. In the system, multiple requests can share a resource. Sharing of a resource increases the utilization of the resource, thus allowing the broker to accrue a higher profit for the same resource. When acquiring a resource, the broker not only considers the amount earned by executing the newly arrived request but also accounts for the possibility of a future request accruing a higher profit that the profits earned by only the current request.

G allows the system to account for this ability of the broker to earn more by sharing the resources among various requests. For example, considering a hypothetical scenario, assume that the i^{th} request has a service time of 1800 s. However, the broker needs the request to be executed for 2500 s to accrue a profit. Using a service time of 1800 s may lead to rejection of the request. However, using a value of G of 1.4 increases the assumed user cost to 2520 times the rate per second and may allow the request to be accepted. The value of G may lead the broker to incur a loss if the estimate made by G cannot be fulfilled by future requests.

This method for calculating the broker profit determines whether the cost of acquiring the resources from the public cloud by the broker is lower than the fees earned from the SCE. When the criterion is met and BP_i is greater than 0, the DM issues a command to the DRPM to acquire a new resource. However, the resource must be acquired only for the time interval required by the request. This is to ensure that the broker does not have idle resources, which cost the IE money. Hence, to achieve this, each new resource acquired by the system has the following characteristics:

Start time for the j^{th} resource ($start_j$) –

- $\mathbf{Start_j}$ = (**EST of the request expected to execute earliest for the j^{th} resource**) – **E**
  Note that E is the additional time taken to accommodate the start time of a resource. Our observation of Amazon's EC2 was that a resource required 120 to 240 s to start.

Stop time for the j^{th} resource ($stop_j$) –

- $\mathbf{stop_j} = \mathbf{start_j} + \left\lceil \frac{\boldsymbol{numSeconds}}{\mathbf{3600}} \right\rceil$

Here, numSeconds is the difference in seconds between the expected completion time of the last request on resource j and $start_i$, A ceiling function converts a real number to the closest integer value that is greater than or equal to the number. Since a typical public cloud provider charges by the hour, the resource is held until the end of this "paid hour" period, by using the ceiling function. Each resource j has an internal timer that is set to expire at the time $stop_j$. Hence, even though the resource must be switched off after the request finishes execution, any remaining time might be used by another request.

Grade of service

If BP_i is less than 0, RA must check with another component that it makes use of in thread 1. This component is known as the GoS component. Even though the broker does not accrue a profit from the request, RA needs to check whether the GoS criteria are satisfied. RA consults with the GoS component that contains configuration details set by the SCE and agreed upon by the IE. Failing to meet the specified GoS value may result in

a violation of the service level agreement guaranteed by the IE. The GoS criteria used in this paper is the blocking ratio (B). B is the ratio of the number of requests rejected by the system to the total number of systems sent by the SCE. The blocking ratio B of the system is continuously monitored. Irrespective of whether a profit will be accrued, a resource is acquired if the following is true:

$$\mathbf{B} > \mathbf{B_{spec}} \tag{6}$$

Hence, if none of the available resources can satisfy the SLA for request i, the broker uses the following rules to decide whether to acquire a resource from the public cloud.

- Rule I: When $BP_j > 0$, acquire the j^{th} resource
- Rule II: When $B > B_{spec}$, acquire the j^{th} resource
- To release a resource, B_{spec}, acquire the j^{th} resource

To release a resource, the following rule is followed:

- Rule III: When ($stop_j$) = current time, release j^{th} resource

Rule I states that the j^{th} resource is acquired if a profit is generated for the broker. Rule II states that the request must be accepted if the system fails to meet the specified grade of service guaranteed by IE. Rule III specifies that the release of the j^{th} resource occurs when the current system time is equal to the stop time of the resource.

Rule II may force the broker to accrue a loss for a request, that needs to be accepted to keep B_{spec} at the desired level and maintain the GoS criteria. This may occur when the arrival rate of user requests as well as the service time of the user requests is low. Additionally, a value of B_{spec} close to 0 may force the broker to acquire new resources for every request and that may lead to a loss for accepting requests with low job execution times.

Next, the reactive auto-scaling algorithm given in Table 1 is described next. For each incoming request Req_i, the algorithm is executed. Acquisition of resources by RA follows criterion (i) or (ii). However, before a resource can be acquired, MMS must determine whether the existing set of resources can accommodate the request. This is shown in lines 1–7. DM sends each request to MMS which decides whether one of the resources in the current set of resources can be allocated to meet the request deadline as explained in "Auto-scaling criteria" section.

If the request cannot be accommodated in the existing set of resources, the auto-scaling algorithm inside RA must be executed to determine whether a profit may be accrued from the request based on Rule I, after acquiring a new resource from the public cloud provider and scheduling the new request on that resource. An exception to this case is when the value of B exceeds the specified GoS condition (B_{spec}). This follows from the scaling up rule specified in Rule II. In this case, the resource is acquired, irrespective of whether a profit is generated by scheduling the new request on a newly acquired resource to meet the GoS criteria guaranteed by IE to SCE. The process of resource acquisition is shown in lines 8–14. The start and stop times of the resource to be acquired are set in line 12 and line 13 respectively. Line 14 performs the action of acquiring the new resource by communicating the request to DRPM. Resources are released when the current time reaches the stop times computed in line 13 and stored in their respective timers.

Table 1 Reactive Auto-Scaling Algorithm

```
1.  for each resource Res_j, from j = 1 to N do
2.      if Req_i can be scheduled on Res_j
3.              Schedule Req_i // Schedule using MMS
4.          if Res_j.stopTime < Req_i.endTime
5.                  Res_j.stopTime ← Req_i.endTime
6.          break
7.  end for
8.  if Req_i not scheduled
9.              UC_i ← G × (c_i × c_pvt)
10.     BC_i ← ⌈c_i/3600⌉ × c_pub
11.     if (UC – BC) > 0 or B > B_spec
12.       startj ← (EST of earliest Reqj) - 300
13.                 stopj ←startj + ⌈numSeconds/3600⌉
14.                 acquireResource(start,stop)
```

The algorithm describes only the scale-up operation. Resources are scaled down automatically when the current time becomes greater than the stop time shown in Rule III. The stop times are set in two cases, when a new resource is started or when an existing resource is determined to be profitable for a new request and the stop time is extended.

Proactive auto-scaling

Upon request arrival, the hybrid algorithm determines whether the incoming request is to be accepted (determined by MMS) and if accepted, whether.

Thread 2 is responsible for proactive auto-scaling after a predetermined time based on the predicted user demand. DM implements a proactive auto-scaler (PA) for auto-scaling resources. The hybrid auto-scaler auto-scales proactively by using two helper modules, GoS and MLE. When auto-scaling proactively, DM uses MLE to predict user demand. MLE uses a machine learning algorithm to predict the future workload. Initially, MLE requires a training period to operate accurately. In this paper, the training period is chosen as 10% of the total experimental run time. During this phase, no requests are accepted by the system. However, a modification of the system can still accept requests and schedule them in a reactive fashion. Once the training phase is completed, the system can start normal operations of rejecting requests if not deemed profitable for the broker and auto-scaling after a fixed time interval. DM invokes the operation of MLE after every k requests. MLE predicts the characteristics of the upcoming k requests, Req = {Req_1, Req_2, …, Req_k}.

In this paper, most experiments are conducted using Linear Regression (LR) as the machine learning algorithm. Each request characteristic is predicted separately using a time series analysis. MLE allows other machine learning algorithms to work, in place of LR. In addition to LR, another machine learning algorithm known as Support Vector Machines (SVM) has been discussed.

Next, the operations of thread 2 are discussed. At the beginning of each prediction operation, DM asks for the next k requests to arrive from MLE, denoted by R_k. MLE provides all five characteristics, i.e., AT, EST, ST, DL and T, described earlier in "Broker architecture" section, for each of the k predicted requests. After getting a response, DM simulates the resource management operations for these predicted future requests using MMS. This invokes a mock scheduling and matchmaking operation. However, MMS operates in the same way and does not differentiate between a real request sent by SCE and the predicted request sent by the DM.

Based on the output of MMS, DM decides whether to acquire new resources or change the stop time for existing resources to schedule the requests on the existing resource (discussed later in "Matchmaking and scheduling" section). After receiving the characteristics of the k predicted requests from MLE, DM invokes MMS and gets back a true or false response. If MMS can successfully schedule the request on one of the existing resources, it returns a true value, and DM takes no further action (except for adjusting the stop time of the resource, if required). However, if MMS is unable to schedule the predicted request, the DM has to decide whether a resource will need to be procured in the future using the auto-scaling criteria (iii) and (iv) described in "Auto-scaling criteria" section. It should be noted that even though the requests are only predicted, DM needs to ensure that it consults the GoS component if there is a likelihood of the request causing the GoS criteria to fall below the agreed threshold.

Each new resource, predicted to be acquired by the system has the same start and end time characteristics as described when a new request arrival triggers a resource acquisition. In addition, as mentioned earlier in "Pricing" section, *c_pub* and *c_pvt* are the broker cost rate and user cost rate respectively. Hence, the following factors must be taken into consideration when determining the predicted profit for the broker, finally leading to a predicted profit function:

- *Predicted Broker Cost* ($\boldsymbol{BC_i^p}$) – The predicted cost for running the i^{th} resource is given by: $[(stop_i - start_i)] \times c\_pub$
- *Predicted service time* ($\boldsymbol{s_i}$) – The predicted total service time for q requests on the i^{th} resource is given by: $\sum_{j=1}^{q} st_j^i$ in seconds
- *Predicted Total User Cost* ($\boldsymbol{UC_i^p}$) – The predicted cost required for users to execute requests on the i^{th} resource is given by: $[s_i] \times c\_pvt$
- *Predicted Broker Profit* ($\boldsymbol{BP_i^p}$) –The predicted profit earned by servicing user requests on the i^{th} resource is given by:

$$BP_i^p = (UC_i^p - BC_i^p)$$

If BP_i^p is greater than 0, then the request is accepted. Note that the superscript p denotes predicted values. The actual values of these parameters are denoted by superscript a and used in "Experimental parameters" section. The predicted broker profit is used in the following way by DRPM that handles the task of communicating with the public cloud provider and acquiring or releasing resources.

GoS is used when a newly arrived request is about to be rejected. If the GoS criteria are unsatisfied, then the request is accepted regardless of broker profit. The GoS criterion, as explained earlier in "Reactive auto-scaling" section is B_{spec}. However, since the value of B which is monitored does not actually change with the number of

requests that have been predicted by MLE, we use another metric called B^P which stores the predicted value of B. The predicted value of the blocking ratio B^P for the system is continuously monitored. Irrespective of whether a profit will be accrued, a resource is acquired if the following is true:

$$\mathbf{B^P > B_{spec}} \quad (7)$$

Once the resource p to be acquired has been identified, rules similar to those mentioned in "Reactive auto-scaling" section are followed for the acquisition and release:

To acquire resources, the two rules below are followed:

- Rule IV: When $BP_i^P > 0$, acquire the j^{th} resource
- Rule V: When $B^P > B_{spec}$, acquire the j^{th} resource

To release resources, the following rule is followed:

- Rule VI: When $(stop_j)$ = current time, release the j^{th} resource

Next, the predictive auto-scaling algorithm (Table 2) is described next. The algorithm determines whether new resources need to be acquired or not, based on predicted workload. The stop times are set in two cases, when a new resource is started or when an existing resource is determined to be profitable in the future and the stop time is extended. Acquisition of resources suggested by MMS follows Rule I. DM invokes MLE to predict the next set of k requests before starting this algorithm. The system state is characterized by the list of resources acquired by the system and the list of requests for each resource. Lines 1–13 describe how DM sends each request to MMS which decides whether the current set of resources may meet the request deadline. If not, a new resource to be acquired is suggested. Each predicted request is sent to MMS to predict whether the request may be scheduled on an already running request. If the request cannot be scheduled using the existing set of resources, DM uses the algorithm (shown in Table 3) which helps in selecting a new resource. Profit calculation is performed by comparing UC_i^P for using the i^{th}-resource with BC_i^P for that resource. Only if there is a profit (using Rule I) is a new resource acquired (line 5) by setting a start time and a stop time for the resource.

Experimental prototype

This section describes the experimental setup. "Programming language/framework" section describes the programming language and framework used in the implementation of the prototype. "Implementation" section describes the details about the implementation and challenges faced while deploying the broker on a real cloud environment.

Programming language/framework

The experimental platform is implemented using the Java programming language. Representational State Transfer (ReST) is chosen as the architectural pattern for designing the distributed framework. The system needs two different communication operations:

- Communication between the client enterprise and the broker.
- Communication between the broker and the resources in the private cloud.

The messages transferred using the REST architecture are in the form of the JavaScript Object Notation (JSON)

Table 2 Proactive Auto-Scaling Algorithm (Part A)

```
1.  for each pred. request Req_i, from  i=1 to k do
2.          for each resource Res_j, from  j = 1 to N do
3.                  if Req_i can be scheduled on Res_j
4.                          Schedule Req_i
5.                  if Res_j.stopTime < Req_i.endTime
6.                          Res_j.stopTime ← Req_i.endTime
7.                  break
8.          end for
9.  if Req_i not scheduled
10.         Schedule Req_i on ResN+count;
11.         nRes ← ResN+count.
12.         Increment count by 1.
13. End for
14. Call selectResource ()
```

Table 3 Proactive Auto-Scaling Algorithm (Part B)

```
1.  for all nRes_i, from j = 1 to count, do
2.          UC_p^j ← [s_j] × [c_pvt]
3.          BC_p^j ← [(stop_j – start_j)] × [c_pub]
4.  updateB()
5.          if BC_p^j > 0 or B^p > B_spec
6.                  acquire(start_j,stop_j)
7.  end for
```

[38]. JSON is a syntax for storing and exchanging data that is an easier to use alternative to the Extensible Markup Language (XML). In comparison to XML, JSON is much easier for humans to read or write. JSON can also be parsed faster by machines.

The platform is compatible with different cloud platforms such as Amazon Web Services and OpenStack. DRPM, one of the components of the broker, uses a software development kit (SDK) to communicate with each of the two cloud providers. Other cloud providers can also work with this system, provided that DRPM has the correct SDK to invoke resource creation operations as well as the correct API keys that allow DRPM access to the cloud provider resources. More details regarding DRPM are given in "Broker architecture" section.

The platform uses Spring as the underlying framework to achieve loose coupling between different components. The Spring context is used to provide the dependency injection design pattern, which removes hard-coded dependencies and makes the system loosely coupled, extendable and maintainable. The application also uses the Spring web module to create web components. The application employs the REST end points as web services to facilitate communication among the internal components of the broker, between the broker and SCE, and between the broker and the VPC. Gradle is used to build and deploy the application. It is written as a Groovy script. Gradle also helps to support external dependencies using Maven. The external projects used by this application are listed below:

- *WEKA* – A machine learning software used by the prediction operations.
- *Log4J* – A Java-based logging utility.
- *Date4J* – A Java-based date library utilized to achieve nanosecond time precision, used in all modules that depend on measuring time.
- *SSJ* – A Java-based stochastic simulation library for generating random numbers and distributions.
- *AWS* – An SDK produced by Amazon for controlling resources in the Amazon cloud.
- OpenStack4j – An SDK produced by OpenStack for controlling resources in the OpenStack cloud.

Initially, a simulator that allows rapid prototyping was created to test the behaviour of the system by implementing an event-driven simulation. This version is the prototype of the system defined in "Pricing" section. One of the main benefits of the simulation framework is that experiments may be completed in less time compared to the experiments for the prototype implementation. This is due to the simulator being able to "simulate" a request execution and not having to wait for requests to use CPU cycles to finish execution. The simulator possesses the ability to skip to a point where a request has finished its execution. This is an event-driven simulation that simulates the resources of a public cloud by building an interface similar to the cloud API used by OpenStack and Amazon Web Services. More details about the simulation platform are given in "Pricing" section.

Implementation

This paper is based on dual implementation: i) a prototype implementation that uses resources from a real cloud provider and ii) a simulation, as explained in "Auto-scaling techniques" section. For the prototype, synthetic requests with a desired set of request attributes were generated on a desktop computer outside the cloud environment. This computer acted as the source of all requests, i.e., as the client enterprise and was equipped with an Intel Core i7, 4 CPUs (2.8 GHz) and a 12 GB of RAM. The request was sent to a broker running on the public cloud provider using REST. After generating a single request, this user module waited for the given inter-arrival time duration before generating a successive request. The request characteristics such as earliest start time, service time and deadline were generated using the respective distribution discussed in "Broker architecture" section. For the simulation, instead of sending the requests to a broker running on the cloud, they were sent to a broker running on the same computer as the one from which the requests were being generated. The resources were simulated inside the same computer, and the simulation was carried out in an event-driven fashion.

The resources in the prototype were acquired from the Amazon's EC2 cloud service. The broker ran inside the EC2 cloud on a *t2.small* instance. This instance was kept in a public subnet accessible to the Internet. The rationale behind this is to allow clients to communicate with the broker without any restrictions. All ports except 8080 were closed to allow only RH, running as a web service, to be available to SCE. The resource pool, controlled by the broker using DRPM, was placed in a private subnet without Internet access. Each resource was running on an Amazon *t2.micro* instance.

When the broker received a new request via RH, it used MMS to determine whether the request could be scheduled on one of the resources available to the broker. Scheduling of the request was performed as explained in "Matchmaking and scheduling" section. The decision to accept or reject a request was sent back to the user using the REST framework. If MMS allowed the request to be scheduled, the broker sent the request to the chosen resource via REST, where it was queued. The matchmaker would select the first resource in its queue and start executing a request after its earliest start time. The execution of a request was emulated by running a for-loop to keep the CPU busy for the predetermined service time duration of the request. Once the request finished execution on the resource, the resource informed the broker of the request completion.

For the simulator, the properties of the resources were stored as static properties in a Java class file, and any request submitted to the resource would be queued until the start time of the request. At the start time of a request, the events that needed to occur to start the request were triggered by a local scheduler in the resource object where the request was to be executed. This models the prototype implementation, where the local scheduler on each resource handles a set of requests scheduled by MMS on the specific resource. Both the simulator and the prototype were tested using various combinations of system and workload parameters. Comparing the simulator and prototype results, a close match between the measurement results obtained using the scaled prototype and the simulation results was observed. In most cases, a difference of 2 to 7% was observed, with the largest difference being 9%. Further discussion is provided in "Experimental results" section.

Performance evaluation

This section provides an analysis of the performance of the hybrid broker and compares its performance with that of another system from the literature. The section also compares the proactive broker with previous works by the author, who discussed purely proactive [16] or reactive [17] approaches. Evaluation of the system performance takes both broker profit and user cost into account.

Experimental parameters

This section focuses on how changing parameter values affects the performance of the system. Each experiment is performed by changing one parameter at a time while holding the other parameters at their default values (see Tables 4 and 5). The default values are shown in italics. Column 2 specifies the set of all values used for the respective parameter.

Workload Parameters:

The values for each parameter used are given in Table 4. Similar parameter values have been used by other researchers [10, 39]. In these experiments, 20% of the requests sent by the user were OD requests, while the remaining were AR requests. Workloads comprising a higher and lower proportion of OD requests are observed to lead to similar sets of conclusions. Details of the experiments have not been included due to scope limitations. While systems with a single mean arrival rate have typically been studied in the literature, in some systems, λ can vary based on the time of day or day of the week. In addition to a system subjected to a single arrival rate, a system subjected to two arrival rates ($\lambda_{low} = 0.0027$ requests/s and $\lambda_{high} = 0.0053$ requests/s) was also experimented with. Other values of arrival rates may also be used but the aforementioned rates are deemed adequate for performing the comparative analysis presented in this paper.

Load Factor (f) – is the ratio of the number of requests generated with an arrival rate of λ_{low} to the total number of requests generated during the experiment. Thus, if the number of total requests is k, then the number of requests with an arrival rate λ_{low} is: $f \times k$ and the number of requests with arrival rate λ_{high} is: $(1\text{-}f) \times k$.

Mean Service Time (S) – characterizes the execution times of the requests that are uniformly distributed [39].

Earliest Start Time (EST) – The earliest start time for the request is computed by adding a value V, which is uniformly distributed between 0 and 12 h, to the arrival time of the request. The deadline for the request is computed as:

$$\begin{aligned}\text{Deadline for a request}(\text{DL}) = {} & \text{Earliest Start Time of request} \\ & + \text{Request Service Time} \\ & + \text{Laxity},\end{aligned}$$

Table 4 Summary of Workload Parameters

Load Factor: **f**	0, 0.2, *0.4*, 0.6, 0.8, 1
Mean Service Time (in mins): **S**	*50*
Laxity Factor: **Lf**	*1*
Mean Arrival Rate (in reqs/s): **λ**	0.0083, 0.0067, *0.0053*, 0.0037, *0.0027*

Table 5 Summary of System Parameters

Machine Learning Algorithm: **MLA**	*Linear Regression*, Support Vector Machine
Number of predictions: **P**	*10*, 20, 30, 40, 50, 100
User cost rate (in $ per sec): **c_pvt**	0.00000694, *0.00000833*, 0.00000972, 0.0000111, 0.0000125, 0.0000139
Broker cost rate (in $ per hour) for a micro instance: **c_pub**	*0.02*, 0.025, 0.03

where Laxity is defined as the slack time a request has in addition to its service time before it reaches its deadline.

Laxity Factor (Lf) – is an integer characterizing the slack time of requests.

$$\text{Laxity} = \text{Request Service Time} \times \text{Laxity Factor}$$

Mean Arrival Rate (λ) – is the rate at which requests arrive on the system. A Poisson distribution is used to model the request arrivals [39].

System Parameters:

Linear regression (LR) is used as the default machine learning algorithm (MLA) with one experiment comparing the broker profit of a System using LR with one using Support Vector Machines (SVM).

The number of prediction for requests (P) is varied during the experiments.

The default public cloud provider cost is based on current Amazon charges; 2 cents [40] for every hour a micro instance is used. The system parameters are given in Table 5.

It was observed that increasing G from 1 to 1.2 resulted in a 5% increase in BP. However, increasing G beyond 1.2 did not result in any significant increase in BP. This indicates a 20% probability of a new request arriving and using the remaining time for the resource (refer to "Reactive auto-scaling" section). B_{spec} is maintained at 20 to 40%, which guarantees that 60 to 80% of the user requests will be executed.

Performance Metrics:

Broker *profit* (BP^a) – is the actual profit a broker earns per second in dollars, which is a performance metric of interest. A difference between the predicted and actual broker profit may be found in "Pricing" section. "Proactive auto-scaling" section also discusses the predicted metrics denoted by a superscript p which are calculated using the proposed algorithm during run-time.

Total User cost (UC^a) – is the amount charged to the user per second in dollars. A higher BP^a and a lower UC^a indicate good system performance.

Alternative systems

The hybrid system also referred to as System I in this paper, is compared with other alternative systems such as a system described in [41]. The authors of [41] describe a greedy approach that acquires resources as and when required. This system is modified by incorporating a mechanism to remove idling resources to improve its performance. This modified system uses a broker that scales up when it needs an additional resource and scales down if no more requests are pending on a particular resource. This enables the broker to increase its profits by ensuring idling resources are returned to the service provider. This system is called the non-proactive system or System II. A hybrid system that functions as System II during the training period (accepting all client requests) and switches to the proactive system (System I) at the end of the training period is an interesting topic for future research. In System III, users obtain the resources directly from the public cloud provider without broker (in the intermediary enterprise) intervention. With this system, users incur the additional overhead of starting and stopping resources as well as handling the matchmaking and scheduling operations. Thus, for System III, only total user cost is compared to that of System I and System II, as there is no broker involved and thus no broker profit is accrued.

The hybrid system introduced in this paper is also compared to a purely proactive system discussed by the same authors [15, 16] and a purely reactive system [17]. The proactive system mentioned in those papers uses a similar algorithm to the one used in this paper for the comparison. However, while performing the prediction, the system used in this paper introduces the GoS, as it is used in both the hybrid and reactive systems. The proactive system is referred to as System I–P, and the reactive system is referred to as System I–R. Both user cost and broker profit are compared for all three systems.

Experimental results

Each experiment was run long enough such that the system operated in a steady state. Each experiment was repeated a sufficient number of times such that an interval less than or equal to ±5% was achieved at a confidence level of 95% for each performance metric. All the experiments were performed using the prototype. However, a comparison of the measurement results with those of a simulated system showed close agreement. As explained in "Programming language/framework" section, a simulator is often preferred for running experiments because the results are obtained in a shorter period and the cost of resources acquired from an Amazon cloud for running experiments using the prototype system is avoided.

Figure 3, for example, presents a comparison between the broker's profits determined via measurements of the prototype system with that observed for a simulated system. The greatest difference between the two systems is 9% at f = 0.8. At lower values of *f*, the difference between the BP^a values determined for both systems is much smaller: for example, a difference of 3% is observed at f = 0.0.

Impact of arrival rate

A performance analysis of a system subjected to a single arrival rate is performed to investigate the impact of λ on profit. Figure 4, presents the effect of λ on the profit generated by System I and System II. BP of both systems is directly proportional to the arrival rate. This is because, at higher arrival rates, the system receives a larger number of requests per unit time, thus increasing the potential for a earning a higher profit/h. System I earns in between 2 and 6 times the profit earned by System II.

Impact of load factor

Figure 5, compares the profit that accrued by System I and System II for different values of f. In all cases, System I is able to generate more profit than System II. The profit is inversely proportional to f: as f increases, profit increases. For example, when f is 0, BP is the highest. Note that at f = 0, the system is subjected to only the higher arrival rate. As f increases, the proportion of time for which the system is subjected to the lower arrival rate increases. At f = 1 the system is subjected to only the lower arrival rate and the profits earned are lower. Overall, for any given f, BP for System I is 3–4 times higher than System II (see Fig. 5). Figure 6 presents a comparison of the total user cost incurred in all the three systems. As shown in the figure, System I leads to a lower total user cost in comparison to that of the other two systems. The experiments show that an increase in arrival rate implies a higher cost per hour for the user.

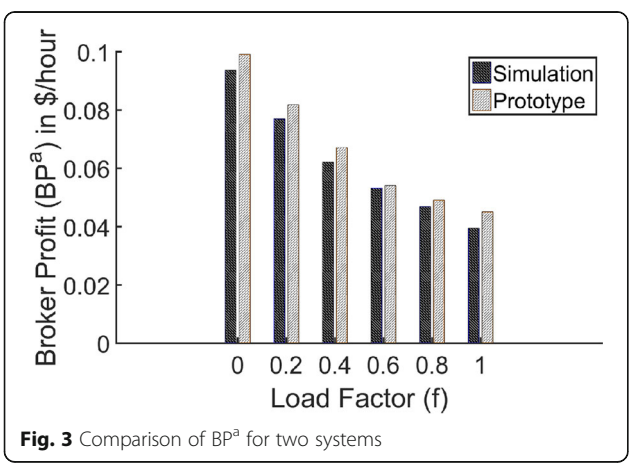

Fig. 3 Comparison of BP^a for two systems

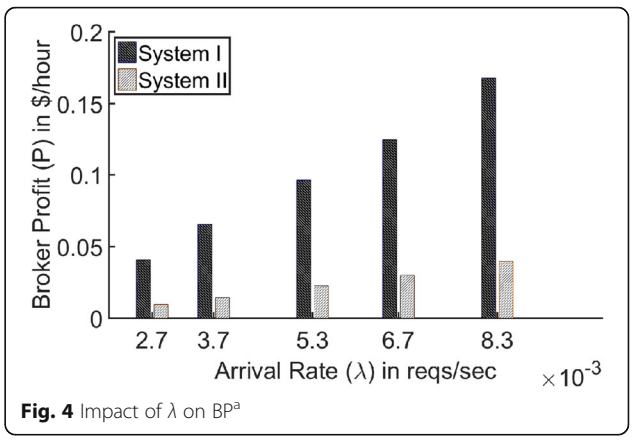

Fig. 4 Impact of λ on BP^a

This is because an increase in arrival rate implies that the broker is forced to address a larger number of request arrivals per unit time and thus has a greater opportunity to accrue a profit by executing a larger number of requests.

Impact of laxity factor

In Fig. 7, Systems I and II are subjected to different values of Lf. A Lf value of 0 implies that there is no additional slack time for the requests. This renders the auto-scaling ineffective because the system cannot schedule multiple requests without missing some of the deadlines. Consequently, the profit accrued is almost 0. However, as the laxity in the system increases, BP also increases. A 150% improvement in BP is observed as the Lf is increased from 1 to 4. Overall, System I performs 3 to 6 times better than System II. Laxity is a function of the workload (and set by users) and cannot be adjusted by the system.

Impact of service time

Figure 8 captures the impact of S on the performances of System I and System II. The figure shows that the profit increases as S increases. This is because the larger

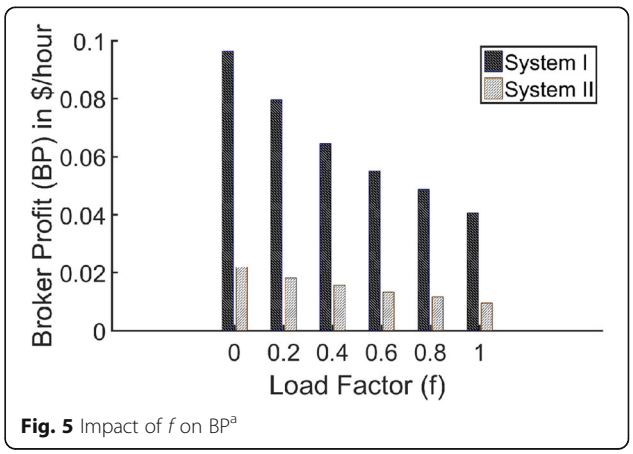

Fig. 5 Impact of *f* on BP^a

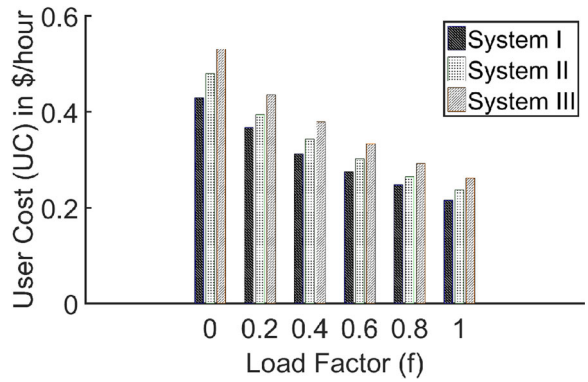

Fig. 6 Impact of *f* on UC[a]

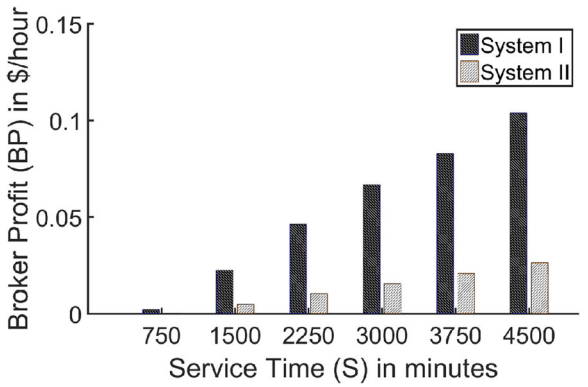

Fig. 8 Impact of *S* on BP[a]

the service time of a request, the higher the utilization of the resources. This enables the broker to exploit the pricing difference and earn a greater profit at a higher value of S. For a mean service time of 12.5 min, however, the system accrues almost no profit. Note that for the parameters used in this experiment it would take at least 40 min of requests running on a resource per hour for the system to break even and lead to a broker profit.

Additionally, when the service time increases, the user ends up paying more per request. This is a consequence of the pricing model, where the user is charged per second and the user cost increases as the service time increases. However, when using the default price of $0.02/h paid by the broker to the cloud provider and $0.00000833/s paid by SCE to the broker used in the experiment, there is an optimal period of cost savings that is applicable. Based on those prices, if the service time of the request is between 1 and 2399 s and between 3601 and 4799 s, the user will end up paying less for the request than when the resource is directly acquired from the public cloud.

System II follows a similar trend to that displayed by System I. For any given mean service time between 25 and 75 min, the broker profit for System I is 1.5 to 4 times that for System II.

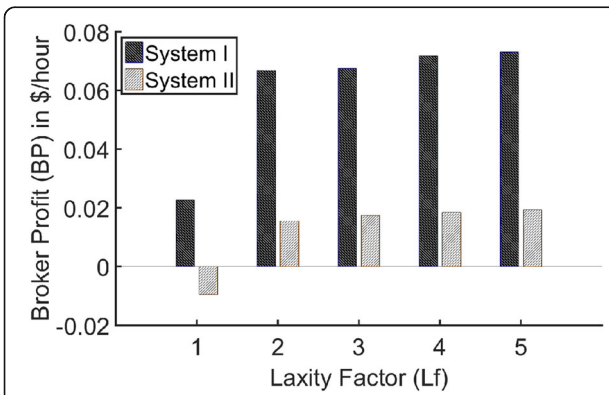

Fig. 7 Impact of Lf on BP[a]

Other parameters

Impact of Number of Predictions: Profit is observed to increase initially with an increase in the number of predictions used by MLE increases. However, it reaches a peak at 50 and then decreases as the number of predictions is increased further. Small values of P do not allow MLE to properly learn the request behaviour. Additionally, when P is too large, outdated information may get included in the prediction process, thus reducing the accuracy of predictions.

Impact of Machine Learning Algorithm: The improvement in BP_a achieved by SVM over LR is marginal, i.e., 2 to 4%.

Comparison with reactive and proactive approaches

This section compares the performance of the proposed system with that of the purely proactive and reactive systems discussed in "Alternative systems" section. The goal of this section is to analyse the impact of the hybrid auto-scaling framework on the performance metrics UC[a] and BP[a] and to compare the performance of this framework to that of other reactive/proactive frameworks described in previous papers by the authors of this paper. In Fig. 9, the BP[a] values of System I, System I–R and System I–P are compared for different values of *f*. As in Fig. 5, as the load factor increases, the broker profit decreases. The graph shows that in almost all cases, the hybrid system performs better than the two other systems. The hybrid system accrues 11 and 46% more profit than System I–R and System I–P, respectively, in the best case.

In Fig. 10, the UC[a] values for System I, System I–R and System I–P are compared for different values of *f*. As observed in Fig. 6, as the load factor increases, the user cost decreases. Additionally, the graph shows that in most cases, the hybrid system leads to a higher user

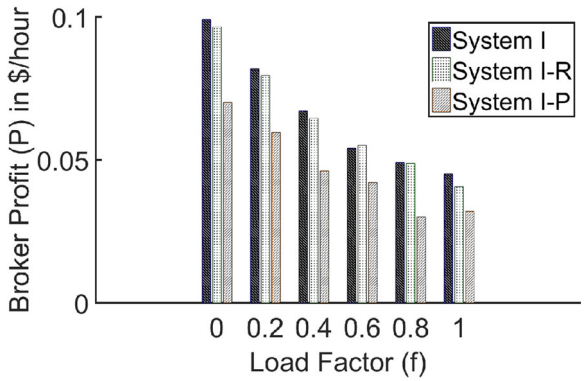

Fig. 9 BP[a] with change in *f*

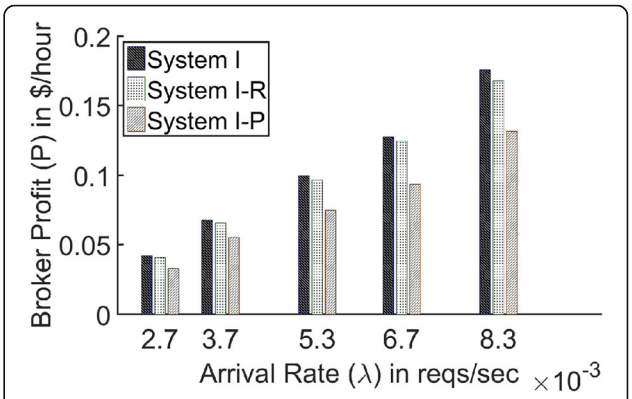

Fig. 11 BP[a] with change in λ

cost in comparison to that of the two other systems. The reason for the hybrid auto-scaler performing worse than System I–R and System I–P is that in the other cases for the reactive/proactive systems, the brokers accepted fewer requests, leading to the lower profit shown in Fig. 9. However, since the hybrid broker for System I accepted a higher number of requests per hour, the users had to pay more to the broker. Thus, the higher profit earned by the hybrid system leads to an increase in cost to the users by 7 and 122% in comparison to System I–R and System I–P, respectively. These numbers reflect the maximum percentage difference between System I and Systems I–R and I–P for all values of *f*. Note that for all values of *f* considered, the UC[a] value observed for the hybrid auto-scaler is lower than that observed for System II and System III.

Figure 11 compares BP[a] for System I, System I–R and System IP for different values of λ. As in Fig. 4, as the arrival rate increases, the broker profit increases. The graph shows that in all cases, the hybrid system performs better than System I–R and System I–P. The hybrid system accrues 5 and 37% more profit than System I–R and System I–P respectively in the best case.

Figure 12 presents a comparison of BP[a] values for System I, System I–R and System IP for different values of *S*. As in Fig. 8, as the mean service time increases, the broker profit increases. Figure 12 shows that in almost all cases, the hybrid system performs better than the other two systems. The hybrid system accrues 10 and 73% more profit than System I–R and System I–P respectively in the best case.

Figure 13 presents BP[a] values for System I, System I–R and System IP for different values of *Lf*. As shown in Fig. 7, as the arrival rate increases, the broker profit increases Fig. 13 shows that in all cases, the hybrid system performs better than the other two systems. The hybrid system accrues 13 and 49% more profit than System I–R and System I–P respectively in the best case. An interesting observation is that in some cases, System I–R performs better than System I–P and worse in other cases. However, the hybrid system, System I, always achieves a better performance compared to that of the other systems or close to the best performance achieved for a given case.

Profitability and cost analysis

Two important questions in the context of the proposed IE-based system concern the relationship between the workload, system and price parameter values that lead

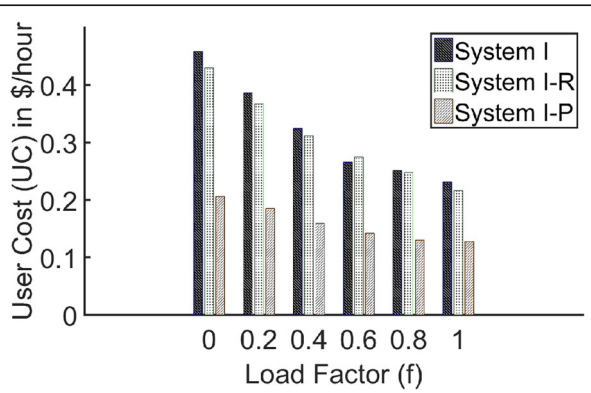

Fig. 10 UC[a] with change in *f*

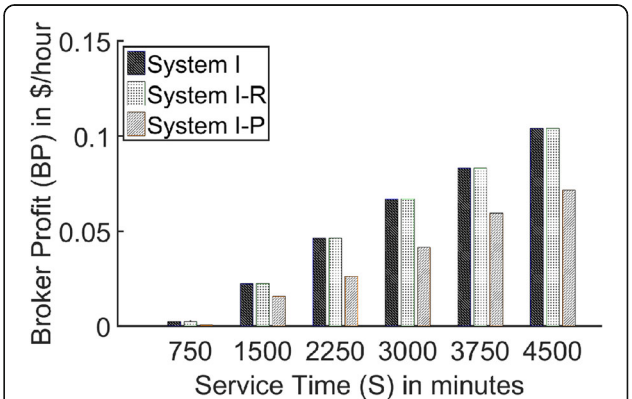

Fig. 12 BP[a] with change in *S*

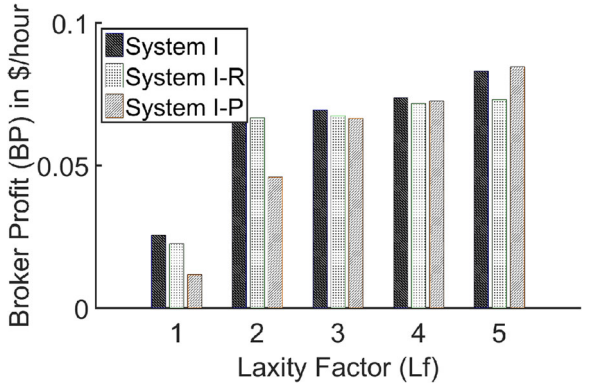

Fig. 13 BP[a] with change in *Lf*

to (i) a broker profit and (ii) a lower cost to the user in comparison to the situation in which the user acquires a resource directly from the public cloud provider. Analyses of broker profitability and user cost are presented in this section.

Profitability (for the broker/IE) analysis

Let, t_r be the time in seconds that a given resource r is used during the period of T_r hours, the time for which the resource was "rented" by IE from the cloud service provider).

IE (broker) earning during this period = $t_r \times c\_pvt$.

Cost incurred by IE for paying the public cloud service provider during this period = $T_r \times c\_pub$.

For IE to remain profitable, broker earning must exceed the cost incurred by broker:

$$t_r \times c\_pvt > T_r \times c\_pub$$

$$\Rightarrow \frac{t_r}{T_r} > \frac{c_{pub}}{c_{pvt}}$$

Dividing both sides by 3600 yields:

$$\Rightarrow \frac{t_r}{T_r \times 3600} > \frac{c_{pub}}{c_{pvt} \times 3600}$$

$$\Rightarrow U_r > \frac{cpub}{cpvt \times 3600}$$

Where U_r is the utilization of the given resource r. Thus, if U_r for every resource exceeds $\frac{cpub}{c_{pvt} \times 3600}$, IE is guaranteed to remain profitable. But this bound is conservative (i.e. IE may remain profitable even if U_r for some resources do not satisfy the inequality). Hence, to determine the conditions under which the overall profit is greater than zero, a new inequality that leads to a tighter bound is formulated:

$$\sum_{r=1}^{R} (tr \times c\_pvt) > \sum_{r=1}^{R} (T_r \times c\_pub)$$

$$\Rightarrow \frac{\sum_{r=1}^{R} (tr \times cpvt)}{\sum_{r=1}^{R} (Tr \times cpub)} > 1$$

Note that the broker will not remain profitable if this inequality is not satisfied.

Analysis of user cost

Let, s_r be the time in seconds that a given resource r is used by request s during the period of T_r hours.

For the price model to be attractive to the user: User cost (with IE) must be lower that the User Cost (direct) (the cost incurred when the resource is acquired directly from a public cloud provider)

$$s_r \times c\_pvt > \left(\lceil \frac{s_r}{3600} \rceil \times c\_pub \right)$$

The user can use this inequality to determine which system is better for executing the request: the IE based system or the public cloud provider. For example, the defaults values of *c_pvt* ($0.00000833/s) and *c_pub* are ($0.02/h) used by the IE based system remains more attractive as long as $s_r < 2400$ s. For a higher ratio of *c_pub* and *c_pvt*, with *c_pvt* = $0.00000694444/s and *c_pub* = $0.02/h for example, the IE based system will be more attractive to the user if $s_r < 2880$ s.

Conclusions

This paper describes a technique and the associated algorithms and framework for performing hybrid auto-scaling of resources in a cloud. A prototype implementation of the system is also discussed. The auto-scaling technique introduced in the paper combines a machine learning-based proactive approach with a reactive approach for scaling resources to adapt to changes in workload demands. As demonstrated via simulation experiments and measurements of the prototype, the proposed technique can generate a profit for the intermediary cloud provider hosting the broker while reducing the cost incurred by users. The key features of the technique are as follows:

- The number of resources in the pool used for the user requests does not need to be determined a priori and is controlled dynamically thereby eliminating the need for capacity planning.
- The framework allows users in an enterprise, for example, to submit AR as well as OD requests to a private cloud provided by the intermediary cloud provider where resources from a public cloud are used to handle the workload.
  - The benefit of using the proposed system is that even though the public cloud may not have AR support, the broker at the intermediary cloud provider performs the necessary operations to enable the handling of AR requests.
  - The novel auto-scaling technique introduced in the paper combines a proactive approach by utilizing machine learning algorithms and a reactive approach that scales resources based on a Grade of Service criterion.

Both simulations and measurements based on a proof of concept prototype implementation using a real cloud demonstrate the effectiveness of the proposed technique. A number of insights were gained by performing simulation experiments. Some of the important observations are presented

- The experimental results demonstrate that using the proposed hybrid broker can lead to a higher profit as compared to other proactive/reactive systems. The broker profit is observed to depend on parameters, such as load factor, arrival rate, mean service time, laxity factor.
- A higher mean arrival rate enables the broker to earn a higher profit. As a result, lower the load factor, higher is the difference between the profit generated by the hybrid system and that generated by System II.
- The broker profit is observed to be directly promotional to mean service time
- The broker profit seems to increase as the laxity factor increases. Most laxity factors considered in the experiments led to a broker profit. However, for a laxity factor of 1, the requests have no additional slack time to meet their deadlines. In such a situation, the profit earning mechanism fails.
- Changing the machine-learning algorithm has a small impact on the broker profit with a more advanced algorithm like SVM performing better than linear regression. However, the overhead incurred by using an algorithm with a larger time complexity increases the time taken by the broker to make auto-scaling decisions.
- The Hybrid system performs better than a purely proactive or reactive system in most cases when comparing BP[a].

A performance analysis of the proposed system using real workload traces is an interesting direction for future research. Extending the auto-scaling technique to storage and network resources warrants investigation. A hybrid system that functions as System II during the training period (accepting all client requests) and switches to the proactive system (System I) at the end of the training period is another interesting topic for future research. Such a system will allow the elimination of the training period required to use the proactive component.

Abbreviations

API: Application programming interface; AR: Advance reservation; AT: Arrival time; AWS: Amazon Web Services; B: Block Probability; BC: Broker Cost; c_pub: Broker cost rate; c_pub: Broker cost rate; c_pvt: User cost rate; c_pvt: User cost rate; CAPEX: Capital expenditure; CS: Total User Cost; DL: Deadline; DM: Decision Maker; DRPM: Dynamic Resource Pool Manager; EDF: Earliest Deadline First; EST: Earliest Start Time; f: Load Factor; FF: Firth Fit; GoS: Grade of service; IaaS: Infrastructure as a Service; IE: Intermediary Enterprise; Lf: Laxity Factor; LR: Linear Regression; MLA: Machine Learning Algorithm; MLE: Machine Learning Engine; MMS: MatchMakeSched; OD: On-demand; OPEX: Operational expenditure; P: Number of predictions; PA: Proactive Autoscaler; QoS: Quality of service; RA: Reactive Autoscaler; ReST: Representational State Transfer; RH: Request Handler; S: Mean Service Time; SCE: Single Client Enterprise; SDK: Software development kit; SFT: Scheduled Finish Time; SLA: Service level agreement; SST: Scheduler Start Time; ST: Service Time; SVM: Support Vector Machines; T: Type of request; UC: User cost; VPC: Virtual Private Cloud; XML: Extensible Markup Language; λ: Mean Arrival Rate

Acknowledgements

We are grateful to Telus and the Natural Sciences and Engineering Research Council of Canada (NSERC) for supporting this research.

Funding

No funding has been used for this research.

Availability of data and materials

Not applicable.

Declarations

This section has been included to adhere to the guidelines mentioned in the Journal of Cloud Computing website [42].

Authors' contributions

The work presented in this paper is based on AB's Ph.D. research and thesis, which is supervised by SM and BN. AE-H is a collaborator and industrial partner for this research. AB devised and implemented the algorithms and conducted the experiments. SM, BN and AE-H provided guidance and participated in system design. All the authors read and approved the final manuscript.

Authors' information

Anshuman Biswas is a Ph.D. candidate at Carleton University, Ottawa, Canada. He is working on devising efficient auto-scaling techniques in a cloud environment. Anshuman received his Master's degree in Computer Science from the University of Calcutta, Kolkata, India in 2009. He currently works at Turbonomic, as a software engineer, where he designs systems to automate operations inside a data-centric environment. He has previous experience working with Tata Consultancy Services Limited in India and Trend Micro in Canada.

Shikharesh Majumdar received the PhD degree in Computational Science from the University of Saskatchewan, Saskatoon, Canada. He is a professor and the director of the Real Time and Distributed Systems Research Centre, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada. His research interests include the areas of cloud and grid computing, operating systems, middleware, and performance evaluation. He has just completed his term as the area editor of the Simulation Modelling Practice and Theory journal published by Elsevier. He is a member of the ACM, a senior member of the IEEE, and was a distinguished visitor for the IEEE Computer Society from 1998 to 2001.

Biswajit Nandy has more than 25 years of experience in the area of data communication and network security. At Solana Networks, as Chief Technology Officer he is focused on technology development for network and security monitoring solutions. He is an Adjunct Research Professor with the Systems and Computer Engineering Department at Carleton University, Ottawa, Canada. His research areas are software defined networking, cloud computing and security solutions for SCADA and IOT based networks. He has authored more than 60 papers in conferences and journals and holds 22 granted US patents in the area of IP networking. Biswajit holds a PhD degree in Electrical and Computer Engineering from the University of Waterloo, Canada.

Ali El-Haraki has 29 years of solid experiences in IT and Telecommunications with strong expertise knowledge in designing and architecting networks. His roles include Due Diligence/Discovery Missions, ISMS, NGN Technologies and Strategy. Wide-ranging experience as both a technology architect and consultant, combined with knowledge of technology and global trends, enable Ali to lead projects that demand immediate results and solutions to complex problems.

As a result, Ali has built a reputation for success based on his extensive technical knowledge, his ability to translate customer needs into technological solutions. He has been involved with the successful

completion of numerous projects, including TELUS Next Generation Network, TELUS new IDC, TELUS' TS Private cloud and leading the Carleton University/ TELUS cloud computing research program for TELUS.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada. [2]TELUS, Ottawa, Canada.

Received: 20 February 2017 Accepted: 27 November 2017
Published online: 19 December 2017

References

1. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2010) A view of cloud computing. Commun ACM 53(4):50–58
2. Amazon. "Amazon EC2 Pricing" [Online]. Available: http://aws.amazon.com/ec2/pricing/. Accessed Dec 2017
3. Grossman R (2009) The case for cloud computing. IT Prof 11(2):23–27
4. Lehrig S, Eikerling H, Becker S (2015) Scalability, elasticity, and efficiency in cloud computing: a systematic literature review of definitions and metrics. In: Proc. of the 11th international ACM SIGSOFT conference on quality of software architectures, New York, pp 83–92
5. Foster I, Kesselman C, Lee C, Lindell B, Nahrstedt C, Roy A (1999) A distributed resource management architecture that supports advance reservations and co-allocation. In: Proc. of seventh international workshop on quality of service (IWQoS), London, pp 27–36
6. Buyya R, Garg SK, Calheiros RN (2011) SLA-oriented resource provisioning for cloud computing: challenges, architecture, and solutions. In: Proc. of the international conference on cloud and service computing (CSC), Hong Kong, pp 1–10
7. Lim N, Majumdar S, Ashwood-Smith P (2017) MRCP-RM: a technique for resource allocation and scheduling of MapReduce jobs with deadlines. IEEE Trans Parallel Distrib Syst 28(5):14
8. Mao M, Humphrey M (2011) Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: Proc. of high performance computing, networking, storage and analysis, Seatle, pp 1–12
9. Mao M, Humphrey M (2013) Scaling and scheduling to maximize application performance within budget constraints in cloud workflows. In: Proc. of 27th international symposium on parallel & distributed processing (IPDPS), Boston, pp 67–78
10. Melendez JO, Majumdar S (2012) Matchmaking on clouds and grids. J Internet Techonol 13(6):853–866
11. White J, Dougherty B, Schmidt DC (2012) Model-driven auto-scaling of green cloud computing infrastructure. Futur Gener Comput Syst 28(2):371–378
12. Sotomayor B, Montero RS, Llorente IM, Foster I (2009) Virtual infrastructure management in private and hybrid clouds. IEEE Comput Soc 13(5):14–22
13. Chieu TC, Mohindra A, Karve AA, Segal A (2009) Dynamic scaling of web applications in a virtualized cloud computing environment. In: Proc. of IEEE international conference on e-business engineering (ICEBE), Macau, pp 281–286
14. Mao M, Humphrey M (2012) A performance study on the VM startup time in the cloud. In: Proc. of IEEE 5th international conference on CLOUD computing (CLOUD), Honolulu, pp 423–430
15. Biswas A, Majumdar S, Nandy B, El-Haraki A (2014) Automatic resource provisioning: a machine learning based proactive approach. In: Proc. of international conference on cloud computing technology and science (CloudCom), Singapore, pp 168–173
16. Biswas A, Majumdar S, Nandy B, El-Haraki A (2015) Predictive auto-scaling techniques for clouds subjected to requests with service level agreements. In: Proc. of IEEE world congress on services (SERVICES), New York City, pp 311–318
17. Biswas A, Majumdar S, Nandy B, El-Haraki A (2015) An auto-scaling framework for controlling Enterprise resources on clouds. In: Proc. of 15th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid), C4BIE workshop, Shenzhen, pp 971–980
18. Hasan MZ, Magana E, Clemm A, Gudreddi SLD (2012) Integrated and autonomic cloud resource scaling. In: Proc. of network operations and management symposium (NOMS), Maui, HI
19. RightScale. "RightScale Cloud Management" [Online]. Available: http://www.rightscale.com/. Accessed Dec 2017
20. Dutreilh X, Rivierre N, Moreau A, Malenfan J, Truck I (2010) From data center resource allocation to control theory and back. In: Proc. of 3rd IEEE conference on cloud computing (CLOUD), Miami, FL
21. Kouki Y, Ledoux T (2013) SCAling: SLA-driven cloud auto-scaling. In: Proc. of the 28th annual ACM symposium on applied computing, New York, pp 411–414
22. Xu H, Li B (2012) Anchor: a versatile and efficient framework for resource management in the cloud. 24(6):1066–1076.
23. "Amazon CloudWatch" [Online]. Available: http://aws.amazon.com/cloudwatch/. Accessed Dec 2017
24. Wang W, Niu D, Li B, Liang B (2013) Dynamic cloud resource reservation via cloud brokerage. In: Proc. of 33rd international conference on distributed computing systems (ICDCS), Philadelphia, pp 400–409
25. Moore LR, Bean K, Ellahi T (2013) A coordinated reactive and predictive approach to cloud elasticity. In: Proc. of fourth international conference on cloud computing, GRIDs, and virtualization, Valencia, pp 87–92
26. Breitgand D, Henis E, Shehory O (2005) Automated and adaptive threshold setting: enabling Technology for Autonomy and Self-Management. In: Proc. of the 2nd international conference on automatic computing (ICAC '05), Seattle, pp 204–215
27. Urgaonkar B, Shenoy P, Chandra A, Goyal P, Wood T (2008) Agile dynamic provisioning of multi-tier internet applications. ACM Trans Auton Adapt Syst 3(1):39
28. Iqbal W, Dailey MN, Carrera D, Janecek P (2011) Adaptive resource provisioning for read intensive multi-tier applications in the cloud. Futur Gener Comput Syst 27(6):871–879
29. Amazon. "Amazon VPC" [Online]. Available: http://aws.amazon.com/vpc/. Accessed Dec 2017
30. Mazzucco M, Dumas M (2011) Reserved or on-demand instances? A revenue maximization model for cloud providers. In: In proc. of IEEE international conference on cloud computing (CLOUD), Washington DC, pp 428–435
31. Melendez JO, Biswas A, Majumdar S, Nandy B, Zaman M, Srivastava P, Goel N (2013) A framework for automatic resource provisioning for private clouds. In: Proc. of the 13th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid), delft, pp 610–617
32. Buyya R, Yeo CS, Venugopal S (2008) Market-oriented cloud computing: vision, hype, and reality for delivering IT services as computing utilities. In: Proc. of the 10th IEEE international conference on high performance computing and communications (HPCC), Dalian, pp 5–13
33. de Assuncao MD, di Costanzo A, Buyya R (2009) Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In: Proceedings of the 18th ACM international symposium on high performance distributed computing, New York, pp 141–150
34. Pentaho. "Time Series Analysis and Forecasting with Weka" [Online]. Available: http://wiki.pentaho.com/display/DATAMINING/Time+Series+Analysis+and+Forecasting+with+Weka. Accessed Dec 2017
35. Waikato UO. "Weka 3: Data Mining Software in Java" [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/. Accessed Dec 2017
36. Wang H, Jin J, Wang Z, Shu L (2011) On a novel property of the earliest deadline first algorithm. In: Proc. of eighth international conference on fuzzy systems and knowledge discovery, shanghai, pp 197–201
37. Farooq U, Majumdar S, Parsons EW (2009) Achieving efficiency, quality of service and robustness in multi-organizational grids. J Syst Softw 82(1):15
38. "JSON" [Online]. Available: http://www.json.org/. Accessed Dec 2017
39. Sulistio A, Buyya R (2004) A grid simulation infrastructure supporting advance reservation. In: Proc. of the 16th international conference on parallel and distributed computing and systems, Boston, pp 1–7
40. Amazon. "Amazon EC2 Pricing" [Online]. Available: http://aws.amazon.com/ec2/pricing/. Accessed Dec 2017
41. Yu J, Buyya R, Tham CK (2005) Cost-based scheduling of scientific workflow applications on utility grids. In: Proc. of first international conference on e-science and grid computing, Melbourne, pp 140–147
42. Springer. "Journal of Cloud Computing" 24 11 2017 [Online]. Available: https://journalofcloudcomputing.springeropen.com/submission-guidelines/preparing-your-manuscript/research-article. Accessed Dec 2017