2012 19th Asia-Pacific Software Engineering Conference

An Approach to Estimating Cost of Running Cloud Applications based on AWS

Huihong He[1], ZhiYi Ma[1+], Xiang Li[2], Hongjie Chen[1], Weizhong Shao[2]
Key Laboratory of High Confidence Software Technologies, Ministry of Education,
School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
[1]{hehh09, mzy, chhj}@sei.pku.edu.cn, [2]{lx0414, wzshao}@pku.edu.cn

Abstract—**estimating the cost is important for cloud application developers to services in clouds, and becomes even important when it needs remaining a certain service level at the same time. Though currently much work has been down to predict cost and performance of cloud applications, most of them perform either before application design or after the application construction, which leads to either imprecise estimation or irreparable design fault. In this paper, we propose an approach to estimate the cost of running typical applications in Amazon Web Service (AWS) cloud during design phase. We propose an UML Activity-extended model (AeModel) to describe execution of application service and introduce an extraction algorithm to extract information contained in the AeModel automatically. We propose a cost model on AWS, which can help developers to estimate operating cost during design phase and satisfy performance needs, with an algorithm to produce suitable purchase solutions automatically. We perform case studies using a web-based business application to show effectiveness of our approach, and find that our approach can help developers lessen cost by adjusting application models.**

Keywords-Amazon Web Service Cloud; Pricing Model; design phase; performance; cost estimation

I. INTRODUCTION

Cloud computing [1] [2] is becoming an attractive alternative for small- and medium-sized companies who are especially with limited budget. Compared with traditional computing environment where applications run on on-premise infrastructures, companies need not pay a lot before lunching services or for coping with peak demand. Besides, companies can avoid keeping IT staff to manage the facilities.

Although cloud charges the application providers on a pay-as-you-go basis, we notice that currently different clouds offer different pricing models. The differences lay in two ways: valuable services are different, and services of the same type are charged differently. Since AWS cloud is considered as a pioneer in cloud computing, which sets up a mature and enormous pricing to charge for ingredient services respectively, we found our research in AWS platform. The AWS pricing includes various charge items, charge alternatives and preferential tactics. To economize the cost in AWS as much as possible, developers need to answer the following questions:

- How many resources should be applied for?
- How to choose charge alternatives and preferential tactics?

Developers need deep understanding of AWS pricing to resolve those questions. Moreover, developers soon find themselves faced massive resource offerings and complicate relationships between resource offering and pricing offerings. From a cloud application developer's perspective, problem becomes even more challenging when the application has Quality of Service (QoS) requirements. The developers have to design application more delicately to meet requirements of the both. Hence, an effective process is urgently needed to reflect automatically whether the design satisfies needs and guide developers to adjust application design.

In this paper, we propose an approach trying to answer those two questions: provide a resource allocation and recommend suitable purchase plan according to the allocation. Besides, our approach performs during application design stage rather than after construction. Contribution of our work lies in three respects as follows: first, our approach supports to perform performance related cost estimation during design phase, by taking the proposal Activities-extended business model as well as budget and performance needs. In this way, our approach can help developers to reveal inadequate design early and thus lesson the cost of adjustment. Second, our approach is almost automatic performed to generate series of purchase choices for developers, except the decomposition of performance related budget may want domain experts' involvement. Therefore, our approach effectively relieves the burden of developers to performing cost estimation. Last, AWS pricing model, as far as we know, is first time trying to incorporate resource provision onto consideration, which explicitly and comprehensively depicts AWS pricing. AWS pricing model contributes to increase automation of our approach, and to provide developers with comprehensive understanding of AWS pricing.

The rest of this paper organizes as follows: in section 2, we discuss some related work. Then in section 3, we present an outline of our approach to gain a better understanding. After that, we give a detailed description of our approach. And then, a case study of using our approach to produce purchase plan or adjust design are presented in section 4 and a discussion is presented in section 5. Finally, we summarize our contributions and describe avenues for future work.

II. RELATED WORK

Much research has done on cost-base d scheduling to meet performance expectations of workflow-based applications, such as [3] [4] [5]. However, these work settled on traditional computing technology e.g. grid computing, and thus cannot take the advantages of cloud computing such as infinity provision and pay-as-you-go model. Recent work such as [6] [7]

+ corresponding author

The work supported by the National Natural Science Foundation of China (No. 61272159), Beijing Natural Science Foundation (No. 4122036), the National Basic Research Program of China (No. 2011CB302604), National Science Foundation of China project on Group Creative Project (No.60821003), and National High Technology Research and Development Program of China (No. 2012AA011202)

proposes scheduling algorithms to optimize execution of workflows and consumption. Nevertheless, these work do not consider how to integrate the algorithms well with inherent schedulers of cloud platforms, so it cannot take the benefit of elasticity of cloud computing. Our approach, on the other hand, makes full use of Amazon Elasitc Load Balancing (ELB) service, where the requests to services are handled by load balancers by default.

Research of cost models on clouds has also become a hot spot recently. Works like [8] [9] [10] proposes different cost model by using the cloud pricing schema. Hongyi Wang etc. in [8] offer an amortized cost model of task long running, where the cost simply multiplies by the price per virtual machine hour and the total running time of the task in hours. Hong-Linh Truong in [10] proposes cost models associated with different types of application, and these cost models are composable. We notice that all of above works simply view cloud pricing as schema, ignoring charge alternatives and preferential tactics which can be used to further optimize the deduced cost. We amend this problem by considering charging alternatives, so developers can enjoy the cost optimization. Besides, because of the cost models charge based on single request, without considering as a whole for concurrency and parallelism, they are not applicable in clouds for it charges based on virtual machine. Our work, on the other hand, studies how applications running on clouds by using the underlying platform services. We map responses into threads by extracting information from Activity-extended model, and view capacity of calculation in holding threads rather than CPU ability.

III. THE PROPOSED APPROACH

In this section, we introduce our approach. We first present an outline of our approach in III-A. In III-B we present AWS pricing model for pricing AWS services. In III-C we present Activity-extended model, using it to model business logic of application services as well as interaction with underlying AWS services. Besides, the extraction algorithm is also presented. We then present in III-D our cost model which takes as input pricing model, service model and performed conclusions. Besides, a purchase algorithm and a purchase plan are also presented.

A. Overview of Approach

Figure 1 depicts the internal organization of the proposed approach and the role of each part. We view cloud application as a series of services, each of which is depicted by AeModel. When the business models are submitted, it automatically translates into uses of *Amazon underlying services*. *Amazon underlying services* include two types of services: *business services,* such as EC2, SQS, and *management services,* such as ELB. *Business services* consume resources provided by *Resource Provision* in a lower layer module *Resource Management*, while *management services* control the *Consumption pattern* such as consuming rate which are also positioned in *Resource Management*.

Apart from the Activity-extended model, *budget-constraint performance expectation* against the service also needs. Since budget is a cumulative argument while performance is usually event-based, before going to the next step, it requires breaking up budget into performance–related budget. Our work aims at two types of applications: application is continuous online, invoked by concurrent requests, which are usually long running; application is online only when invoked, usually of high parallelism. For the first application, budget disassembles into normal load budget and peak time budget. For the second application, budget breaks up into per-task budget. Each dissembled budget matches one performance expectation.

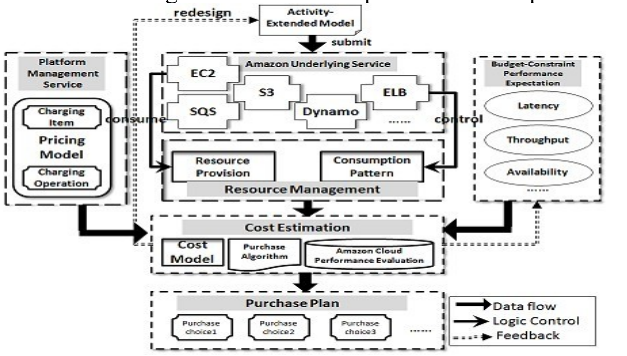

FIGURE 1 ARCHITECTURE FOR COST ESTIMATING APPROACH

During the decomposition, the performing data and curve of the same type application are also required, which serves to describe running characteristics of target application. The decomposition is a heuristic process, which wants domain experts to involve into. In this paper, we divide the budget according to sum of requests to services of the first application per unit time. We distribute according to sum of requests for the second type.

Amazon Pricing Model plays an important role in deciding the unit price of resources, according to the charge alternatives and amount of request. Under different alternatives, the unit price varies; the more resource wanted, the more preferential treatment developers enjoy may enjoy.

Cost Estimation takes the above information as input, with knowledge about performance of Amazon underlying services. The knowledge may be collected from existing work that devotes to exploring AWS performance. *Cost Estimation* module calculates per unit time consumption during normal load and peak time, to meet the refined budget-constraint performance expectation. *Purchase Algorithm* then performs to optimize operating cost via using charging alternatives in *Amazon Pricing Model*.

Different *purchase choices* are generated by *Cost Estimation*, each of which applies for resources under different charge alternatives to meet budget need. It is worth noting when the budget cannot meet although performance expectation satisfies, the expectation will be lower until there is a *purchase choice* falling into the budget. The developers may choose the suitable one as *purchase plan*, guiding them to applying for resource in future.

B. AWS Pricing Model

Figure 3 shows the main classes in the abstract syntax of AWS Pricing Model. We view AWS pricing as a *PricingService,* which consists of sub *PricingServices*, where *namespace* distinguishes which Amazon AWS service it attaches to. A *PricingService* includes a suit of *Charges* and *Actions. Charge* encapsulates *Resource* in associating resource with required amount and deciding unit price. *Resource*depicts valuable resource in AWS cloud. It includes four types of resource: *CloudComponentClass* refers to the elements constituting AWS service, such as “Instance” of EC2 service; *CloudComponentProperty* refers to the properties influencing the behaviors of AWS service, such as “writeCapacityUnits” of DynamoDB service; *CloudComponentOperation* refers to the operations released by AWS service, such as “CreateQueue” “ChangeMessageVisibility” of SQS service; *DataTransfer* refers to the data transferred between end users and AWS cloud and among AWS services. We consider *DataTransfer* as a special resource, for it is not bound to any specific AWS services but implied in every service. To simplify *DataTransfer,* we omit the price charged between computing instances located in different available regions [11]. *Resource* is charged differently under different charge mode and operation type. We consider four types of resource charge mode in our model, which are described as follows:

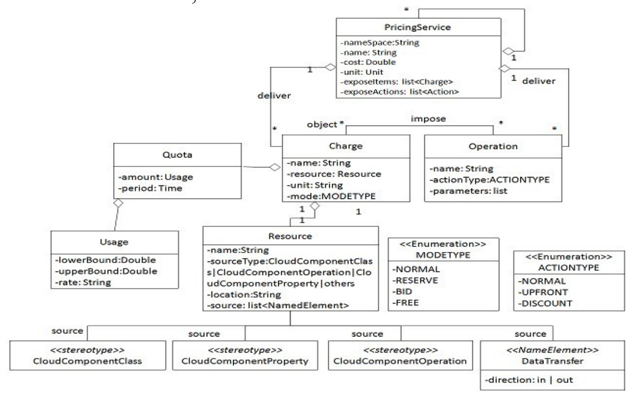

FIGURE 2 PRINCIPAL CLASSES AND RELATIONSHIP IN THE ABSTRACT SYNTAX OF AMAZON PRICING MODEL

- **NORMAL**: charges based on on-demand pattern, enjoying no discount.
- **RESERVE**: applies in advance, enjoying a lower unit price.
- **BID**: introduces stock trading pattern for resource deals.
- **FREE**: at the beginning of every cycle which usually is a month, a certain amount of resources are free.

Beyond that, there are three types of action in released operations to depict the payment methods. It should be noticed that action type is used in concert with charge mode type, which describes as Table 1. Preferential tactics in AWS are realized through composition of *Charge* and *Operation*. For example, when *Charge* is under *free mode*, *discount Operation* imposing on *Charge* indicates that the *Resource* associated with this charge is free for quantity specified in *Charge*.

TABLE 1 COMBINATION OF ACTION TYPE AND CHARGE MODE

Action Type	Charge Mode	Remark
NORMAL	NORMAL	Resource are used on demand, and settled at every end of cycle.
UP-FRONT	RESERVE, BID	When resources are under reserve or bid mode, they are required for upfront payment.
DISCOUNT	FREE	The free resources are used under discount operation.

C. Activity-Extended Model & Extraction Algorithm

AeModel is a core component of our approach. it can be used to model response when a service is called. We extend activities in UML [12] by introducing the following elements:

- *CloudExecutionUnit*: extends *Partition* element. Besides represent organizational units in a business model, it can be used to describe a group of codes working in concert to carry out the activities of organizational units. Inspired by the server-side architecture design in [13], in our work, we consider an instance of *CloudExecutionUnit* as Java Servlet [14] or alike web technology, which runs as a thread and is recognized as mainstream technology to build applications on clouds. Property “*isRemote*” tells whether the unit is performed on cloud. When the value is *true*, it means that this unit carries out on cloud.
- *CloudDataBuffer*: extends *CentralBufferNode* element. This element is used to hold intermediate results, which is matching to SQS service in our work.
- *CloudDataStore*: extends *DataStoreNode* element. In our work, DynamoDB service and S3 service complete this element. The former service stores semi-structure data, while the latter stores unstructured data.
- *MessageAdd, MessageRemove* and *MessageAssociation*: extend *ObjectFlow* element to describe the interaction between *CloudDataBuffer* and *Action*, as well as the connection between *CloudDataBuffer* and *ObjectNode.*
- *DataStore and DataQuery*: extend *ObjectFlow* element to describe the interaction between *CloudDataStore* and *Action*.

The information contained in Activity-extended model will be automatically extracted. We set up an algorithm to perform information extraction in Algorithm 1.

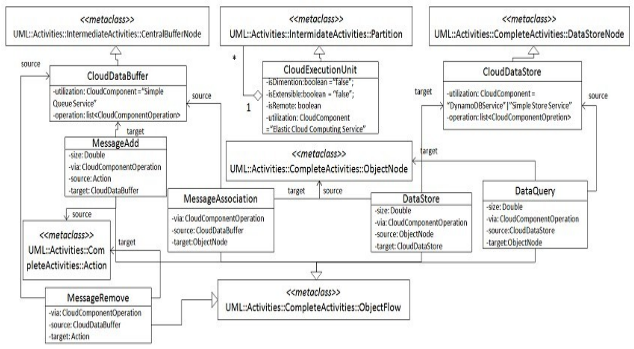
<<metaclass>> UML::Activities::IntermediateActivities::CentralBufferNode
<<metaclass>> UML::Activities::IntermediateActivities::Partition
<<metaclass>> UML::Activities::CompleteActivities::DataStoreNode
CloudDataBuffer
CloudExecutionUnit
CloudDataStore
MessageAdd
<<metaclass>> UML::Activities::CompleteActivities::ObjectNode
MessageAssociation
DataStore
DataQuery
<<metaclass>> UML::Activities::CompleteActivities::Action
MessageRemove
<<metaclass>> UML::Activities::CompleteActivities::ObjectFlow
source
target

FIGURE 3 SEMANTIC MODEL FOR ACTIVITY-EXTENDED MODEL

```
    Algorithm 1: General Operations of Extraction of Activity-extended model
    Data: AEmodel:Activity-extended model
    Data: m_extraLength: length of extra data for holding message in CloudDataBuffer
    Data:s_extraLength: length of extra data for storing data in CloudDataStore
1   numOfThreads ←0; messageTransferIn←0; messageTransferOut←0;
2   messageOperations←0; dataTransferIn←0; dataTransferOut←0;
3   requestIn←0; responseOut←0;
4   /* Extracting Calculating Information
5   if number of forkNode of AEmodel == 0 do
6       numOfThreads ← 1;
7   executionUnitList←∅;
8   foreach forkNode of AEmodel do
9       actionList ← getTargetActions(forkNode);
10      foreach action of actionList do
11          CEUnit ← getCloudExecutionUnit(action);
12          if CEUnit does not exist in executionUnitList do
13              executionUnitList ← executionUnitLis ∪ CEunit;
14  if numOfThreads == 0 do
15      numOfThreads ← length(executionUnitList);
16  /* Extracting message transfer Information
```

```
17  foreach databuffer of AEmodel do
18      length ← length(databuffer.messageAssociation.objectNode);
19      messageAddList ← getMessageFlow(databuffer.messageAdd);
20      messageRemList ← getMessageFlow(databuffer.messageRemove);
21      foreach messageAdd in messageAddList do
22          messageTransferIn ← length + m_extraLength;
23          messageOperations ← messageOperations+1;
24      foreach messageRemove in messageRemList do
25          messageTransferOut ← length + m_extraLength;
26          messageOperations ← messageOperations+1;
27  /* Extracting data transfer information
28  foreach datastore in CloudDataStore do
29      dataTransferIn ← length+s_extraLength;
30  foreach datastore in CloudDataStore do
31      length ← length(dataStore.ObjectNode);
32      dataTransferIn ← length+s_extraLength;
33  /* Extracting In&Out of cloud transfer information
34  foreach CEUnit of AEmodel & isRemoted is false do
35      foreach action of CEUnit do
36          foreach objectFlow of action do
37              if objectFlow.target == action do
38                  responseOut ← responseOut +length(object);
39              end
40              if objectFlow.source == action do
41                  requestIn ← requestIn +length(object);
```

The extraction is a four-stage translation. Algorithm 1 first checks fork nodes in sent Activity-extended model to gather calculating information. If there is no fork node, it suggests executing the model consumes only one thread from the beginning to the end. If the model contains fork nodes, Algorithm 1 then checks every fork node to calculate maximum number of concurrent threads (line8–line13). In the second stage, the volume of message transfer is then calculated by using the extended object flows (line17–line26). Calculating volume of data transfer is similar to that of message transfer (line28-line32). Finally, extracting volume transferred into and out of cloud is implemented by examining every *CloudExectionUnit* (*CEUnit*) running on client side (line 34). The volumes are calculated by checking object flows associated with *actions* in that *CEUnit* (line 34-line 41).

D. Cost Model & Purchase Algorithm

Cloud applications in our work are viewed as composed by a series of services, each of which is requested via Internet. As stated in Ⅲ-A, applications are classified into two types: invoked by concurrent requests; invoked by single call at a time. The unit price and charge mode used in the reminder of this chapter are from AWS pricing model introduced in Ⅲ-B. If the application is invoked at a pattern of single call at a time, the cost of running application is:

$$\cos t_{operating}(\text{app}) = \sum_{i=1}^{n} \cos t(s_i) \times request(s_i) \qquad (1)$$

where the application (app) consists of *n* services (s), and requests are statistical data. The periodic cost of running is gained by summing the cost of performing every service, which relies on cost of one time execution times the number of requests during the given period. The cost of one time execution is calculated by:

$$\cos t(s_i) = \cos t_{data}(s_i) + \cos t_{computing}(s_i) + \cos t_{message}(s_i) + \cos t_{transfer}(s_i) \qquad (2)$$

If the application is invoked by concurrent requests, the total cost of running should consider the cost of operating at normal load and peak time respectively, leading to (3). Besides, calculating cost of one time execution is different, for optimizing consumption caused by ELB cannot be ignored. As a result, the cost of computing should be calculated separately, which leads to (4). $\cos t_{op\_peace}(app)$ and $\cos t_{op\_peak}(app)$ in (3) are calculated by (4). Compared with (2), cost of computing is calculated alone.

$$\cos t_{operating}(app) = \cos t_{op\_peace}(app) \times T_{peace} + \cos t_{op\_peak}(app) \times T_{peak} \qquad (3)$$

$$\text{cost}_{\text{op}}(app) = \sum_{i=1}^{n} (cost_{data}(s_i) + \cos t_{message}(s_i) + \cos t_{transfer}(s_i)) \times request(s_i) + \cos t_{computing} \qquad (4)$$

In (2), the cost of computing to perform i^{th} service is gained by:

$$\cos t_{computing}(s_i) = up_{\text{mode}}(type) \times count(type) \times \max(\sum_{i=1}^{n} T_{execu}(a_i)) \qquad (5)$$

up_{mode}(type) is the unit price for a specific instance under a certain charge mode. $count(type)$ is the number of applied instance of a specific type. $\max(\sum_{i=1}^{n} T_{execu}(a_i))$ is the execution path of maximum duration in s_i. In (4), $\cos t_{computing}$ at peacetime and peak time is achieved by:

$$cost_{computing} = up_{\text{mode}}(type) \times count(type) \qquad (6)$$

The $count(type)$ in (5) and (6) is calculated by using the performed conclusions and extracted calculating information by Ⅲ-C, leading to:

$$count(type) = \frac{\sum_{i=1}^{n} request(s_i) \times numOfThreads(s_i)}{\max_{thread}(type)} \qquad (7)$$

$\sum_{i=1}^{n} request(s_i)$ corresponds to the total requests at one time at normal load and peak time respectively. $numOfThreads(S_i)$ is the maximum number of consumed threads of service s_i calculated in Algorithm 1in Ⅲ-C. $\max_{thread}(type)$ is the maximum number of concurrent threads which experts provide.

The rest of cost calculation is relatively simple. The cost of data store during performing service s_i is given by:

$$\cos t_{data}(s_i) = volume(data) \times T_{duration} \times up(DBService) \qquad (8)$$

where $volume(data)$ is the volume of data transferred into *CloudDataStore*, stored in variable *DataTransferIn* in Algorithm 1. Holding messages in SQS is free, while the operations imposing on queues are charged, leading to:

$$\cos t_{message}(s_i) = count(operation) \times up(SQS) \qquad (9)$$

The $\cos t_{transfer}(s_i)$ consists of two parts: transferring data into and out of cloud, and transferring data between AWS services in the cloud, depicted as follows:

$$\cos t_{transfer}(s_i) = (volume(request) + volume(response)) \times up(EC2) + (volume_{in} + volume_{out}) \times up(SQS) + (volume_{in} + volume_{out}) \times up(DBService) \qquad (10)$$

The cost of computing is influenced by a variety of factors, e.g. the type of instance, the charge mode and tenancy. We introduce algorithm 2 to automatically generate a series of purchase by utilizing cost model, Amazon EC2 pricing model in Ⅲ-B and performed conclusions.

Algorithm 2:*General Operations of Purchasing Computing Resource*

```
1   Data: cpm: EC2 pricing model
2   Data: bud_peace<$req_{peace}$,cost>:the budget of peacetime
3   Data: budget_peak<$req_{peak}$,cost>:the budget of peak time
4   Data: peakload<type, threads>: holding the peak loads of every type of instance
5   Data: $req_{peace}$: number of requests to application at one time at peacetime
6   Data: $req_{peak}$: number of requests to application at one time at peak time
7   Data: $T_{peace}$: the duration of peacetime in specific period
8   Data: $T_{peak}$: the duration of peak time in specific period
    /* Initialization
9   pur_peace<type,count,mode> ← ∅;
10  pur_peak<type,count,mode> ← ∅;
11  purchaseList<purchase, cost> ← ∅;
12  allocation<type, count> ← ∅;
13  trial ← 10;
    /* generate purchase choices
14  for i from 1 to trial do
    /*purchasing during peacetime
15      allocation ← allocate($req_{peace}$ ,peakload);
16      pur_peace ← purchase(allocation, bud_peace);
    /*purchasing during peak time
17      allocation ← allocate($req_{peak}$ ,peakload);
18      pur_peak ← purchase(bud_peak, allocation);
    /*calculate the cost
19      cost ← calculate(pur_peace, pur_peak, $T_{peace}$,$T_{peak}$)
20      purchaseList ← purchaseList ∪ <pur_peace ∪ pur_peak, cost>
```

We adopt Knapsack Problem Algorithm [15] in building *allocate* and *purchase*. Because of space limit, detail of how they work doesn't present here. In *allocate*, available types of instance are seen as kinds of resources, and *peakload* is seen as features of resources. Besides, (7) is taken as distribution means. The number of requests acts as minimum threshold. *purchase* is similar to *allocate*, where the output of *allocate* is taken as features of resource, (5) or (6) as distribution means respectively and budget as maximum threshold. Beyond that, we adjust purchase to hold second-best solutions, which allows developers deciding the size of result set. It by default holds the top ten solutions according to climbing cost sort. So far, we have the total cost of the produced purchase choice if the application is single call pattern. If not, it then can be gain by calculate which adopt (3).

IV. CASE STUDIES

We use a train tickets website to evaluate our approach. The website delivers the same business services as *12306* [16], which is a newly launched website in China. Figure 4 illuminates a successful response by using AeModel, and the transferred messages and stored data in the process.

Now, the website owners want to run it on AWS cloud for cost saving purpose. The owners make a budget for 10,000 $ for the first year. Based on operation record watched from *12306*, there is annual travel peak around the Spring Festival, which lasts 40 days. Also, during the travel peak, there are about 1,000,000,000 requests put into the website each day [17], while during the peacetime, there are around 10,000,000 requests into the website per day [18]. There are about 10,000,000 users registering at this website, and every day about 5,000,000 tickets are updated in database [19].Assuming the website mainly receives requests during daytime (12 hours), we then observe these data: there is 23,150 request per second during peak time and 232 requests per second during peacetime.

Meanwhile, the performed conclusion about performance of AWS cloud is needed. We examine existing work that explores AWS performance. We select the performance data from [20] [21] under similar workload. According to [20], SQS add 4 messages into a queue and remove 2.5 messages from a queue per second on average. SQS delivers between 4 and 5 transactions per thread per second. According to [21], under similar workload, a small instance handles 1500 concurrent requests and receives a *connection_error* of 9.44. A medium instance allows double requests as small instance while enduring the same connection rate.

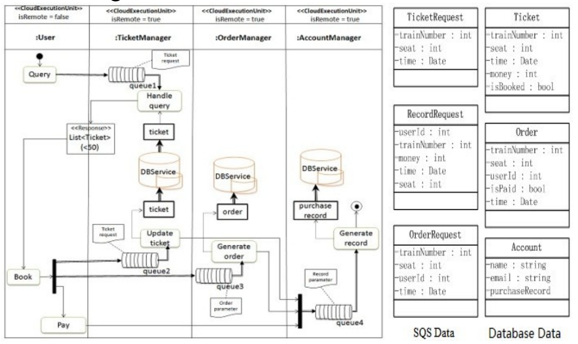

FIGURE 4 EXECUTION AND DATA MODEL OF A SUCCESSFUL RESPONSE

When the execution model in Figure 4 is put into Algorithm 1, we get the following data:

- *numOfThreads:*3; there is a fork node connecting to 3 actions located in three *cloudExecutionUnits*.
- *messageOperaions:*8; every queue is of Single-In-Single-Out pattern.
- *dataTransferIn:*97 bytes; sum of lengths of data into the *DBService*.
- *responseOut:*850 bytes; the length of ticket object is 17 bytes, and it returns 50 items of ticket object when responds to querying request.

Then, Algorithm 2 is performed to purchase computing resources of EC2 services. Table 2 enumerates 6 out of 10 solutions returned by Algorithm 2. Because of space limited, we display three types of purchases which contain on-demand mode only, reserved mode only or both as Table 2 suggests.

TABLE 2 OVERVIEW OF PURCHASE CHOICES

Allocation solution	*Count*	*Charge Mode*		
		On-demand	**Mix**	**Reserved**
Small	47 instances at peak time; 1 instance at peacetime	All instances are on-demand instance	One is heavy utilization reserved instance continuous running for one year; The others are on-demand instances only continuous running during *peak time*	One is heavy utilization reserved instance continuous running for one year; The others are light utilization reserved instances continuous running during *peak time*
Medium	24 instances at peak time; 1 instance at peacetime			

The overview of computing cost of each purchase choice is shown in Table 3, as well as the cost optimization.

TABLE 3 COST OF COMPUTING OF PURCHASE CHOICES

Mode / Allocation	*Cost of Computing*			*Optimization*
	On-demand	**Mix**	**Reserved**	
Small	$4233.6	$3832.92	$3474.12	17.9%
Medium	$4934.4	$4133.04	$3774.24	23.5%

Optimization	14.2%	7.3%	15.9%	29.6%

Table 3 suggests the cost of computing can save up to 30% compared with which doesn't use our approach. The rest of cost is estimated as Table 4 shows. The total cost is shown in Table 5, none of which satisfies the budget constraint. We observe the cost of message processing is far higher than other parts, and has an overwhelming part of total cost which is up to 92.5%. After an investigation is performed, we find budget excess is caused by abuse of SQS service during design. As a result, it suggests the developers modifying the process model to reduce the use of SQS service.

TABLE 4 THE REST OF COST ACCORIDING COST MODEL

Message	*Transfer*			*Database*
	In&Out	**Data**	**Message**	
$333,312.0	$4404.96	$4,404.96	$294.96	$14,248
Sum	$356,664.88			

TABLE 5 COST OF OPERATING WEBSITE FOR DIFFERENT PURCHASE CHOICE

Mode / Allocation	*Cost of Operating*		
	On-demand	**Mix**	**Reserved**
Small	$360,898.48	$360,497.80	$360,139.00
Medium	$361,599.28	$360,797.92	$360,439.12

V. DISSCUSION

Although the case demonstrates the effectiveness of our approach well, we notice there are three aspects can influence the effectiveness of cost estimation. First, we do not include use of ELB into our cost model, for we consider it as cost of managing applications. The developers should pay attention to it; otherwise the cost of using ELB will easily be omitted. Second, the result of budget decomposition acts like arbiter during resource allocation and resource purchase in cost estimation phase. So far, the decomposition still wants experts' participant to carry out. The more accurate the decomposition can be, the more accurate the result of cost estimation can be. Last, the performed conclusions play a key role in allocation of cost estimation phase. Since the infrastructures of cloud providers upgrade as well as cloud services update time to time, the collected conclusions should be continuously updated. The more precise the conclusions are, the more precise the result of allocation can be.

Our approach can also apply to other platforms such as Google App Engine (GAE). Notice that the pricing model is specific to AWS, so it needs to replace it with GAE pricing model. Other parts can be directly reused.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose an approach to estimate the cost of running application in AWS cloud during design phase. We first propose Amazon pricing to comprehensively depict the pricing on Amazon cloud. We extends UML activity model to describe service process and propose an extraction algorithm, where AeModel can then automatically produce input of cost estimation with the algorithm. We propose Amazon cost model which composes of a suit of formulas, with an algorithm serving to purchase the best resource allocation solutions automatically. Taking as input pricing model, AeModel, the performed conclusions and the decomposed budget, the cost model produces a series of purchase choices for developers. We show by case studies that our approach can guide stakeholders to select the most suitable choice as purchase plan, or further help developers to adjust decomposition of budget or application desins.

In the future, we plan to explore on estimating cost of management of cloud application, which supplement our current work so as to provide cloud application developers with a comprehensive control of applications development. Besides, we would like to look into how the results of cost estimation may guide developers to adjust application model, creating an effective circle and thus improve the quality of application.

REFERENCES

[1] M.Armbrust, A.Fox, R.Griffith, etc., "Above the Clouds: A Berkeley View of Cloud Computing", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28,2009.

[2] R.Buyya, C.Yeo, S.Venugopal, etc., "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality ofr Delivering Computing as the 5th Utility", Future Generation Computer Systems, vol.25, pp.599-616,2009.

[3] Eun-Kyu Byun, Jin-Soo Ki, Yang-Suk Kee, etc., "Efficient Resource Capacity Estimate of Workflow Application for Provisioning Resources", 4th IEEE International Conference on e-Science(eScience'08).

[4] J.Yu and R.Buyya, and C.K.Tham, "Cost-based Scheduling of Scientific Workflow Applications on Utility Grids", In Proceedings of the 1st IEEE International Conference on e-Science and Grid Computing, 2005.

[5] J.Yu and R.Buyya, etc., "A Budget Constrained Scheduling of Workflow Applications on Utility Grids using Genetic Algorithms", Workshop on Workflows in Support of Large-Scale Science, 2006.

[6] Tram Truong Huu, Joha Montagant, "Virtual Resource Allocation for Workflow-based Applications Distribution on A Cloud Infrastructure", 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing(CCGrid), 2010, pp. 612-617.

[7] Khawaja S Shams, Dr,Mark W.Powell, Tom M.Crockett, etc., "Polyphony: A Workflow Orchestration Framework for Cloud Computing" ,

[8] Ewa Deelman, etc., " the Cost of Doing Scince on the Cloud: The Montage Example", International Conference for High Perofrmance Computing, Networking, Storage and Analysis, 2008.pp.1-12

[9] Hongyi Wang, etc. "Distributed Systems Meet Economics:Pricing in the Cloud", Proceeding HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing.

[10] Hong-Linh Truong, Schahram Dustdar, etc., "Composable Cost Estimation and Monitoring for Computational Applications in Cloud Computing Environments", International Conference on Computational Science, ICCS 2010.

[11] Amazon EC2 Pricing. http://aws.amazon.com/ec2/pricing

[12] OMG Unified Modeling Language Superstructure, Version 2.4.1

[13] Shigeru Hosono, He Huang, etc., "A Lifetime Supporting Framework for Cloud Applications", 2010 IEEE 3rd International Conference on Cloud Computing.

[14] Java Servlet *from* Wikipedia, available at: http://en.wikipedia.org/wiki/Java_Servlet.

[15] Knapsack Problem Algorithm *from* Wikipedia, available at: http://en.wikipedia.org/wiki/Knapsack_problem

[16] Online Train Tickets Website, available at: http://www.12306.cn/mormhweb/

[17] http://finance.ifeng.com/stock/roll/20120108/5415577.shtml

[18] http://alexa.chinaz.com/?domain=12306.cn

[19] http://www.china.com.cn/news/2012-01/08/content_24353225.htm

[20] Simon L, Garfinkel, "Technical Report TR-08-07: An Evaluation of Amazon's Grid Computing Services:EC2, S3 and SQS", 2007

[21] Liang Zhao, Anna Liu, Jacky Keung, "Evaluating Cloud Platform Architecture with the CARE Framework", 2010 17th Asia Pacific Software Engineering Conference(APSEC).pp.60-69