Improving Availability of Cloud-Based Applications through Deployment Choices

Jim (Zhanwen) Li1, Qinghua Lu1,2, Liming Zhu1,2, Len Bass1,2, Xiwei Xu1, Sherif Sakr1,2, Paul L. Bannerman1,2, Anna Liu1,2
1NICTA, Sydney, Australia
2School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Abstract—Deployment choices are critical in determining the availability of applications running in a cloud. But choosing good deployment for various software application components into virtual machines is a challenging task because of potential sharing of components among applications and potential interference from multi-tenancy. This paper presents an approach for improving the availability guarantee of software applications by optimizing the availability, performance and monetary cost trade-offs of different deployment choices. Our approach explicitly considers different classes of application requests during the decision process. The results of our experimental evaluation show that the approach can effectively improve the availability guarantees with little or negligible increase in the performance and monetary cost of the deployment choice.

Keywords—availability, deployment, optimization, multi-class requests, cloud computing

I. INTRODUCTION

Cloud-hosted applications (such as Netflix, Heroku etc.) use cloud services, such as EC2 from Amazon [1], to deliver their services to end users. Virtual machines (VMs) are increasingly being used to improve the manageability of software systems and lower the total cost of ownership. They allow resources to be allocated to different applications on demand and hide the complexity of resource sharing from cloud users by providing a powerful abstraction for application and resource provisioning. However, sharing VMs among applications with different resource requirements introduces uncertainty in both performance and availability quality of services. The process of deploying application components into virtual machines and the placement of these virtual machines on physical machines has critical impacts on achieving the goal of maximizing the availability guarantees of that application in a cloud environment [10]. During this process, decisions regarding the following challenges arise:

• What is the best deployment strategy for the software components that should be followed in order to deal with the different probabilities of failures?
• How many replicas of each component should be deployed?
• Which components should be co-located on the same virtual machine?
• What other applications utilize which of the components, in what fashion, and with what availability requirements?

In our previous research on run-time resource management for multi-layered cloud-based service systems [23], we have addressed some of above problems in details. In this paper, we are concerned with the problem of static placement of software components where an executor is placed on a VM and it remains there for the duration of its life. This is a critical dependable problem for applications that run in clouds. We extend the existing works, and present a new model that enables the prediction of availability for an application with a particular placement assuming that components are used by different applications and that components can be co-located on the same node. We also allow a single component to be used in a different manner by different applications. Our main contribution is a new algorithm to determine the placement of components on virtual machines based on the definitions of explicit request classes that can be issued from different applications. Availability, of course, is not the only quality that determines optimal placement strategy. Therefore, our algorithm also considers other important factors such as performance and cost. The results of our experimental evaluation show that the proposed placement algorithm can significantly improve the availability guarantee over existing placement strategies under different workload assumptions and resilience management strategies.

The rest of the paper is organized as follows. The next section describes our placement model (Section II). We then provide some examples to show how the various factors might interact (Section III and IV). This is followed by a description of our placement algorithm (Section V) and evaluation of its results (Section VI).

II. COMPONENT PLACEMENT MODEL

Applications in clouds consist of a variety of interacting components which, in turn, depend on a variety of services. We use the term “executor” to refer to either a component of an application or one of the services that run on VMs. Figure 1 illustrates the placement problem in these terms. Applications are represented as a graph of executors. Each of these executors is placed on one (or more) virtual machines (VMs).

The placement problem is concerned with deciding which executors should be placed on which VMs. The other important actor in the setup of this problem is the cloud provider. In practice, cloud providers are responsible for failures in their cloud infrastructure and service providers are responsible for failures in their applications – even if these failures are caused by a failure in the cloud infrastructure.
More precisely, requests come into the application from UserClasses. A UserClass is a collection of humans or other applications that use the task executors. A request is processed by different tasks which are executed by one or more task executors. The task executors are hosted on virtual machines comprising a collection of resources. Figure 2 provides a UML representation of this structure. For a given collection of executors, each UserClass will have its own execution sequence through that collection, possibly utilizing only a subset of the executors. In this paper, we use the term of “Node” to refer to a virtual machine.

Figure 1: Representation of the placement problem

Failure of Task Executors. Task executors may fail for a variety of reasons. The availability of each task executor is a function of the mean time to recovery (MTTR) and the mean time to failure (MTTF) [28]. In this paper, the availability of an executor ($A_i$) is the availability of the individual executor itself.

Failure of Infrastructure Resources. Failure of cloud-based computation resources will cause the executors deployed on them to become unavailable. Infrastructure resource failure could be caused by errors on the individual nodes (e.g., resource overloading and operating system crashing) or failures of the cloud infrastructure (e.g., hardware errors, power issues, API errors, network problems and operation errors). The cloud infrastructure does not automatically offer redundant computation resources to each task executor for fail-over services. Consequently, failures of cloud-based resources lead to failures of application request processing.

Multi-tenancy of Task Executors. When multiple task executors share resources on the same node, failure of one executor may impact the availability of other executors on the node. For example, suppose $t_1$ and $t_2$ are task executors placed on the same VM. Further suppose for some request classes that only $t_1$ is executed. Failure of $t_2$ may impact the availability of $t_1$ for the classes of requests that use $t_1$ and not $t_2$. For example, if $t_2$ fails in a busy state, sufficient CPU cycles may not be provided to $t_1$ to enable it to respond to requests.

In this paper, we define the impact of an errant task executor $t$ on other executors placed on the same node as a correlation factor, $\rho$, whose value is in the range 0 to 1. Availability of an executor $t_1$ ($A_{t_1}$) subject to error interference from executor $t_2$ ($A_{t_2}$) is formulated as $A_{t_1}(A_{t_2}/\rho)$. The smaller the correlation factor, the better is the error isolation. Conversely, when $\rho = 1$, failure of a task executor will always lead to unavailability of other executors running on the same node.

IV. A MOTIVATING EXAMPLE

Figure 3 illustrates a service system handling two classes of requests, Class A and Class B, using a notation from [12][26], where requests from Class A access webservice1 (WS1), database1 (DB1) and database2 (DB2), while Class B requests access webservice2 and database2 (DB2), as shown below.

In practice, there are several possibilities for placing task executors of the illustrated application architecture into virtual machines. Figure 4 shows four typical configurations for the example application architecture, namely: (a) all task executors are consolidated into one node; (b) one node per task, and (c) task executor groupings. The notations $A_{e,t}$ and $A_{e,b}$ respectively represent the availability of processing requests from Class A and Class B, and $C_{e,t}$ represents the monetary cost over resources.

Figure 3 Example of a service system

The deployment architecture shown in Figure 4(a) uses only one node where each errant task executor interferes with both classes of requests, such that errors occurring on WS1 and DB1 may interfere with the requests from Class B, even though they are not required on Class B’s execution path. failure of the node results in downtime of
the whole application. This deployment choice increases resource utilization but at the cost of weak isolation between the processing channels of different requests.

In the deployment architecture shown in Figure (4b), each node hosts only one task executor. So an errant task executor only affects the requests that use it. Failure of a node does not affect other task executors. This deployment choice reduces interference between the processing channels of the different request operations. However, it requires allocating a larger number of nodes.

The deployment architecture shown in Figure (4c) uses two nodes to partially isolate the processing requests from the two classes. In this deployment, errors occurring on WS1 and DB1 only affect requests from Class A, while errors on WS2 interfere with both classes. Therefore, errors on Class A’s request execution are limited to impact the requests from Class A, while failures on executing Class B’s requests may interfere with requests from Class A. This deployment choice uses fewer nodes than option (b) and limits some interference between the two classes.

This example illustrates that the deployment of task executors yields to variance in the availability guarantees of processing the application requests. Deployment choice can improve the availability guarantee by adding redundant resources. Placement optimization can help to further improve the availability guarantee as well. On the other hand, deciding a good deployment choice of executors is not simple in enterprise service systems that incorporate many services with nested requests for other services. It involves nonlinear programming; it cannot be done by simple packing strategies because of joint factors affecting deployment. This situation calls for the crucial need for developing optimization techniques to handle request-oriented aware placements for increasing the availability guarantees of deploying software applications in cloud environments.

V. OPTIMIZATION FOR PLACEMENT

Our approach views the deployment process of task executors as an assignment problem where the goal of an optimal deployment is to:

1) Allocate resource reservations on nodes to task executors,
2) Divide request traffic between multiple replicas of task executors, where applicable
3) Minimize resource costs and failure costs

One approach to increase the availability guarantees is k-resilience[28]. In our work, we assume that every task executes in a user class request path is executed. Extending our approach to consider k-resilience is left for future work.

1) Decision Variables (Capacity Allocation)

The optimization determines a deployment matrix \( \mathbf{\alpha} \) giving the capacity allocation \( \alpha_{nt} \) of task executor \( t \) to node \( n \) for all executors and nodes:

\[
\mathbf{\alpha} = (\alpha_{nt}), \quad \alpha_{nt} = \text{capacity reserved for task executor } t \text{ on node } n
\]

Each task executor is a deployable unit in a node. It is assumed that any number of replicas of each task executor may be deployed, so \( \alpha_{nt} \) may be non-zero for multiple nodes. Each node has an available capacity \( \Omega_n \), and each task executor has required capacity of \( \omega_t \). The units of \( \mathbf{\alpha} \), \( \Omega_n \) and \( \omega_t \) are units of utilization as measured on a chosen “standard processor” defined by Cloud infrastructure providers, i.e. the computation unit defined by Amazon Web Service (AWS) [1], such that a faster processor or multicore processor has a larger capacity. For a task executor \( t \), with multiple \( \omega_{t} \)-0, there are multiple replicas of a task \( t \) that serves it, the requests to which are divided in the proportion given by the corresponding \( \alpha_{nt} \). From \( \mathbf{\alpha} \), we can derive the useful matrices of indicator variables:

\[
P_{nt} = 1 \text{ if } \alpha_{nt} > 0, \text{ indicating task executor } t \text{ is deployed on node } n, \text{ else zero.}
\]

\[
S_{nt} = 1 \text{ indicating at least one task executor } t \text{ required by request in Class } c \text{ is deployed on node } n, \text{ else zero.}
\]

\[
S_n = 1 \text{ if } \max_t (\alpha_{nt}) > 0, \text{ indicating node } n \text{ is used, else zero.}
\]

2) Capacity and Memory Constraint
Deployment of task executors must reserve sufficient resources from the allocated node to meet its SLA requirements. Capacity is allocated up to a node’s “available capacity” \( \Omega_n \). Then the allocations \( \alpha \) must satisfy
\[
\sum \alpha_{nt} \leq \Omega_n , \quad n=1, N
\]
(1)

The required capacity of the task executors \( \omega_t \) are determined by capacity planning on the basis of the required throughputs \( \lambda_\tau \) and the execution demands of the executor \( d_{ct} \). One class \( c \) user request generates a demand by task executor \( t \) for computation units \( d_{ct} \). These parameters can be obtained via tracking filters [33] or statistical analysis [30] on the basis of performance models [12][24]. So sufficient capacity for task executor \( t \) requires \( \Sigma \alpha_{nt} \geq \Sigma_{c} f_{c} d_{ct} \), but, since cost increases with \( \alpha \), the equality is always optimal. Thus
\[
\omega_t = \Sigma_{c} f_{c} d_{ct} = \Sigma \alpha_{nt}
\]
(2)
For each task executor \( t \) with \( \alpha_{nt} > 0 \) in a solution, task executor has a replica on the node.

To fit tasks requiring memory \( M_t \) for each replica of task into a memory of size \( M_n \) for node \( n \), requires
\[
\sum P_m M_t' \leq M_n , \quad n=1,N
\]
(3)
We assume the required performance can be achieved when each task executor obtains the required capacity and memory, such as [22].

3) Formulation of Request Processing

Availability of a class of request processing is determined by the choice of \( \alpha_{nt} \) in deployment matrix \( \alpha \). A non-zero \( \alpha_{nt} \) forces corresponding indicators \( P_m \) and \( S_{nc} \) to be non-zero. We introduce \( \tau_{nt} \) as a placement indicator. \( \tau_{nt} = 1 \) when a task executor \( t \) (which is not used by request in class \( c \)) shares a node \( n \) with other task executors that are used by requests from class \( c \), else zero. \( \rho_{nt} \) is a correlation factor for measuring the impacts of availability of a task executor to others in the same node. In practice, the correlation factor is subject to performance overheads [10] and dependencies between task executors. The availability of a class of request processing is then given as:
\[
A_c = \prod\prod A_{nt}^{s_{nc}} \prod A_{nt}^{p_{m}} \prod A_{nt}^{\tau_{nt} \rho_{nt} c}
\]
(4)
The first term \( \prod A_{nt}^{s_{nc}} \) is the availability of the nodes that execute the tasks required by requests of class \( c \); the second term \( \prod A_{nt}^{p_{m}} \) is the availability of the required task executors, and; the third term \( \prod A_{nt}^{\tau_{nt} \rho_{nt} c} \) is the availability interference between task executors in terms of resource sharing.

Assuming each task executor could have an unlimited number of replicas on any node, the number of feasible deployments of \( n \) task executors on \( m \) nodes is given as \( \binom{m}{n} \) when \( n \geq m \), if capacity constraints are not considered. This is a very large search space for seeking optimal deployment matrix \( \alpha \) subject to the availability relationship in Eq.4. Using traditional Karush–Kuhn–Tucker conditions [21] to search, near optimal value is exhausted for this exponential programming. Linearization helps to improve capability of optimization. Firstly, Eq.4 can be rewritten as:
\[
A_c = e^{\sum_{n} S_{nc} \ln A_n + \sum_{c,t} p_{m} \ln A_t + \sum_{n} \tau_{nt} \rho_{nt} c \ln A_t}
\]
(5)
Because all variables in Eq.5 are binary type, we approximately linearize this equation as below, if the downtime of a node or a component is less than 30 hours per month (whose availability equivalently is over 95%, which is satisfied by most enterprise service systems).
\[
A_t = 1 + \sum_{n} S_{nc} \ln A_n + \sum_{c,t} p_{m} \ln A_t + \sum_{n} \tau_{nt} \rho_{nt} c \ln A_t
\]
(6)

4) Objective Function in Optimization and Model Linearization

a) Objectives Function of Optimization

To deploy \( T \) task executors on \( N \) nodes requires the optimal choice of \( \alpha_{nt} \), for all combinations of node \( n \) and task executor \( t \), to minimize the total cost given as:
\[
\text{Cost} = \Sigma_{c} C_c (1-A_c) + \Sigma_{n} C_n S_n
\]
(7)
subject to the above constraints. This is an objective function. The first term accounts for the cost paid for unavailability of request processing, in which \( C_c \) is the penalty cost for the failure of processing requests from class \( c \), and the second term represents the costs of operating a host \( n \). In the experiments shown in this paper, the penalty \( C_c \) is in the unit of dollar per hour. It is weighted by the contribution of these classes in terms of the size of workloads and the number of required task executors.

b) Mathematical modelling for indicators

We use the BigM method [7] to convert the above indicators \( P_m \) and \( S_{nc} \) described in logical relationships to mathematical constraints, as follows: For each \( P_m \):
\[
\text{BigM}. \ P_m \geq \alpha_{nt}
\]
(8)
BigM is not less than \( \min(\Omega_n , \alpha_{nt}) \).

For each \( S_{nc} \) and \( S_n \):
\[
\text{BigM}. \ S_{nc} \geq \Sigma_{nt} P_m
\]
(9)
\[
\text{BigM}. \ S_n \geq \Sigma_{t} P_m
\]
(10)
BigM must be greater than the maximum value of the right hand side.

For each \( \tau_{nt} \) defined in Eq.4 can be formulated with host selection \( S_{nc} \) and \( P_m \) in a linear relationship, given as:
\[
S_{nc} + P_m - 1 \leq \tau_{nt}
\]
(11)
When both \( S_{nc} \) and \( P_m \) are equal to one, \( \tau_{nt} \) is one, else zero. Because the objective is to minimize the failure costs subject to the availability of requests, any one of \( S_{nc} \) and \( P_m \) is zero, \( \tau_{nt} \) will then be forced to zero to reduce the costs of optimization.
This model considers availability of request processing, failures due to interference between executors, and costs for resource consumption. We name it the ACR model (Availability, Correlation Factor, Resource Consumption) where the placement decision maker takes into account the request availability, resources costs and interference between task executors in the same node as represented in the Objective Function (Eq.7).

5) Extensions of the Optimization Model

The core of this optimization is a mixed integer linear programming (MILP) model. Moderate-sized instances of this problem can be solved by commercial software such as CPLEX [6]. The above ACR model has several variants in terms of different objectives and conditions:

- **AR** (Availability and Resource Consumption): This model simplifies the problem for better scalability. It considers the costs of resources and the end-to-end availability of request processing, but ignores the interference between task executors. The corresponding MILP model thus has such changes as:
  1. Constraints regarding to $\tau_{ne}$ (Eq.11) are taken away, and; 2. the availability model of request processing in Eq.4 is rewritten as

$$A_c = \prod_{i} A_n^{\tau_{ni}} \prod_{t \in EC} A_t^{\tau_{nt}}$$  \hspace{1cm} (12)

And the corresponding linearization then is rewritten as:

$$A_c = 1 + \sum_{t} S_{nt} \ln A_t + \sum_{r \in EC} p_{nt} \ln A_t$$  \hspace{1cm} (13)

- **AC** (Availability and Correlation Factor): This placement model mainly aims for maximizing the end-to-end availability of request processing with consideration of failure interference between executors. Therefore, the costs of resource consumption are not taken into account. So the objective function is modeled as:

$$Cost = \sum_c C_c (1-A_c)$$  \hspace{1cm} (14)

It applies Eq.6 to formulate $A_c$ in the MILP.

- **Avail** (Availability only): This model aims to maximize the end-to-end availability of request processing. To simplify the problem, it does not take into account the interference between executors. Therefore, the constraints Eq.10 and Eq.11 for $S_i$ and $\tau_{ne}$ are not included in the MILP model. $A_c$ is formulated as Eq.13. Since this decision-making does not take into consideration the costs over resource consumption, it uses the objective function Eq.14.

VI. EXAMPLE AND EVALUATION

This section examines the effect of the placement optimizer by simulating the deployment of a number of moderate-sized service-oriented applications, with a layered service structure, onto four types of AWS EC2 instances (small (S), medium (M), large (L) and extra large (XL)). Each application consists of a set of task executors, providing services to multi-classes of application requests. A task executor is hostable by arbitrary types of EC2s, without limitations of the number of replicas for workload sharing, and an executor may handle requests from multiple classes at the same time. In particular this evaluation aims to answer the following questions:

**Question for the Evaluation:**

1. Are the optimization strategies effective, in the sense that the availability of request processing is improved in comparison to current placement strategies?
2. Do the availability-aware placement strategies increase the costs of resource consumption?
3. How to select the adequate placement strategy from available set of deployment choices?

In our evaluation, the demands for CPU and memory for each task executor are determined based on recorded resource utilization of 90% workloads in an enterprise costumer environment [16], respectively in a range of [0.1–3] compute units and [0.2–3GB] memory usage. The availability of each task executor is specified within a range between 0.99–0.999, which is satisfied by most of service applications [25], and the correlation factor $\rho$ is set at 0.1 for task executors which are running in the same node.

Our evaluation compares with a min-cost max-flow based placement approach which has shown to be effective for minimizing the costs of resource consumption in commercial practical products [27]. We take the results returned by this approach as a baseline in evaluation, labeled as “BL”. In the experimental results, the availability of a class of requests is calculated in terms of the returned deployment matrix with Eq.4.

### A. A Small Example for Evaluation

This case study shows the results of deploying two independent service-oriented applications (modeled by simplified version of LQN [11]), shown as model 1 and model 2 in figure 6(a) and 6(b), onto EC2s. This is a small-scale example but it demonstrates the capabilities of our approach.

In the figure, rectangles labeled $M1_{Tex}$ and $M2_{Tex}$ represent task executors in the service systems, and $M1_{Cx}$ and $M2_{Cx}$ are concurrent user classes delivering requests for execution. Each task executor offers the required computation, and may access lower layer task executors through request-reply interactions or asynchronous messaging, indicating by call arcs (arrows) between task executors. The task executors used by different classes of requests are shown in Table 2.

We compare the end-to-end availability of each class of requests given by the placement optimizer subject to different conditions. Table 3 gives the returned availability and the corresponding resource costs on the basis of the returned deployment matrix $\alpha$. The ‘RC’ column in the table denotes the costs on hosts in the unit of $$/\text{hour}. The highest availability for each class of request and the smallest costs used by the placement are highlighted with bold fonts.
Table 2. Execution paths of the service-oriented applications shown in Fig. 6

<table>
<thead>
<tr>
<th>Classes of Requests</th>
<th>Required Task Executors</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1_CA</td>
<td>M1_TE1→M1_TE3→M1_TE5→M1_TE7</td>
</tr>
<tr>
<td>M1_CB</td>
<td>M1_TE1→M1_TE3→(M1_TE4 and M1_TE5)→M1_TE7</td>
</tr>
<tr>
<td>M1_CC</td>
<td>M1_TE1→M1_TE3→M1_TE2→M1_TE3→(M1_TE4 and M1_TE5)</td>
</tr>
<tr>
<td>M1_CD</td>
<td>M1_TE3→M1_TE2→M1_TE5→M1_TE6</td>
</tr>
<tr>
<td>M2_CA</td>
<td>M2_TE1→M2_TE4→(M2_TE5 and M2_TE6)→M2_TE7</td>
</tr>
<tr>
<td>M2_CB</td>
<td>M2_TE1→(M2_TE2 and M2_TE3)→M2_TE5→M2_TE6</td>
</tr>
</tbody>
</table>

The results show that availability-aware deployments improve the end-to-end availability for each class of request processing, from 0.93 to 0.97, in comparison to the min-cost max-flow based resource-oriented placement strategy (BL), which does not take account of availability. Results show that AC gives the highest availability to all classes of requests while it uses the highest resource costs. ACR and AR return approximately the same availability, and their resource costs equal to BL. This illustrates that increasing the application availability does not necessarily lead to extra costs on resource consumption. In the following section, we will deploy large systems with more comprehensive tests to answer the above evaluation questions.

Table 3: The availability of each class of request processing returned by different placement strategies

<table>
<thead>
<tr>
<th></th>
<th>m1_CA</th>
<th>m1_CB</th>
<th>m1_CC</th>
<th>m1_CD</th>
<th>m2_CA</th>
<th>m2_CB</th>
<th>RC ($/h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>0.936</td>
<td>0.961</td>
<td>0.943</td>
<td>0.946</td>
<td>0.928</td>
<td>0.930</td>
<td>1.04</td>
</tr>
<tr>
<td>ACR</td>
<td>0.959</td>
<td>0.961</td>
<td>0.957</td>
<td>0.966</td>
<td>0.951</td>
<td>0.953</td>
<td>1.04</td>
</tr>
<tr>
<td>AR</td>
<td>0.958</td>
<td>0.960</td>
<td>0.957</td>
<td>0.966</td>
<td>0.951</td>
<td>0.953</td>
<td>1.04</td>
</tr>
<tr>
<td>Avail</td>
<td>0.964</td>
<td>0.967</td>
<td>0.971</td>
<td>0.981</td>
<td>0.981</td>
<td>0.978</td>
<td>1.3</td>
</tr>
<tr>
<td>AC</td>
<td>0.965</td>
<td>0.967</td>
<td>0.972</td>
<td>0.981</td>
<td>0.982</td>
<td>0.978</td>
<td>1.3</td>
</tr>
</tbody>
</table>

B. Many Applications, and Effectiveness Analysis

In the evaluation, we simulate deploying 10 copies of heterogeneous service-oriented applications, with a layered processing model similar to Figure 6, but each application is somewhat different and, as before, input parameters (demands of CPU, memory and availability) are randomly chosen in the recorded information from the enterprise computing environment shown in [16]. Each service application handles two to five classes of requests, where each class is using three to eight task executors. In total, there are 30 concurrent classes of requests which are distributed across 80 task executors in this deployment experiment. We used 30 EC2 instances (8 small, 8 medium, 7 large and 7 extra large) available for use. We approximate the availability of each virtual machine (node) is 0.99 based on AWS SLA [1][2][3] and using the pricing model for EC2 given in [1].

The evaluation examines the effect of availability-aware placement by evaluating the returned availability and their resource consumption for the various deployment choices. In this section, we only compare the Avail, BL, AR and AC, because the limited capability of the MILP solver CPLEX on finding the near-global optimal deployment for ACR that requires several minutes due to the size of the optimization problem.

Figure 7 shows a box-and-whisker diagram giving the median, quartiles, and highest (peak) and lowest availability of 30 classes of request processing. The values shown in the diagram are a statistical summary of the availability of the processing requests. The box plots show that Avail, AR and AC are able to deliver over 0.91 availability to all classes of requests, and the median availability is over 0.95. They significantly improve the availability over BL.

Because BL does not take account of availability issues, the returned availability is the lowest. The peak availability is about 0.95, while the median and quartiles are in the range of 0.9 to 0.92. The lowest availability is under 0.90.

AR has a good median behavior, and quartiles and peak are almost the same as Avail. Results indicate that AR is able to achieve as good availability as Avail and it considers the resource costs for deployment process. The difference between AR and Avail is Avail gives slightly larger median while its lowest availability is smaller.

AC achieves the highest availability among these deployment approaches. It can be seen that more than three quarters of classes achieve over 0.95 availability for their request processing, and the lowest availability is as high as 0.93. The difference between Avail and AC shows that considering failure interference in deployment decision-making improves availability of request processing.

Table 4 gives the corresponding costs of resource consumption and the number of EC2s used in each type. Each virtual machine hosts 7~15 task executors, and AR and BL only use 2/3 of resource consumption as Avail and AC.
Table 4. Resource consumption of different placement strategies

<table>
<thead>
<tr>
<th></th>
<th>BL</th>
<th>AR</th>
<th>Avail</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costs on Resources ($/hour)</td>
<td>2.6</td>
<td>2.6</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>Number of EC2s in each type</td>
<td>3 XL</td>
<td>3 XL</td>
<td>5 XL</td>
<td>5 XL</td>
</tr>
<tr>
<td></td>
<td>4 L</td>
<td>4 L</td>
<td>4 L</td>
<td>4 L</td>
</tr>
</tbody>
</table>

As a summary of the results above, AR is able to significantly increase the availability over BL, which is approximately as good as AC and Avail, while keeping its costs over resource consumption as small as BL. AC achieves the best availability for request processing, however, its resource consumption is the largest. Avail uses the same resources as AC, but it cannot achieve the same level of availability due to ignorance of interference.

C. Summary of the Evaluation

- Availability-aware placements are effective in improving the end-to-end availability for multi-class request processing. The average availability increases from 0.91 to 0.95 in comparison to the baseline approach, implying that these placements are able to perform better failure isolation between the processing channels of the different request.
- Increasing the application availability does not necessarily lead to extra costs on resource consumption. It has been shown that AR is able to achieve high availability with the same resource consumption as the baseline approach.
- Error interference affects availability of task executors running in the same node. Considering interference correlations helps in choosing good placement decisions that can improve the availability of application. However, it introduces a larger number of variables in the model where the scalability of the optimization problem cannot be guaranteed in this situation.
- Though ACR considers all factors in the optimization, it cannot scale well due to limitations of the optimization solver, so it is impractical. Our evaluation showed that AR is a more effective approach for practical use among these placement strategies. AR minimizes the resource consumption to as small as the baseline approach, and maximizes the availability for applications to as high as AC and Avail.

Adding redundant services is another approach for improving the availability of a system, such as k-resilience. This approach has been widely used in clouds for data backup and system failover [4]. A key approach for increasing the effects of k-resilience with less resource is to reduce the failure rates of each replica. Our approach optimizes the placement in each node, maximizing availability for the applications with no or negligible increase on the performance and monetary cost of the placement choice. The experimental results above have demonstrated that the request oriented aware placement is an effective approach for increasing availability of cloud-deployed applications. It provides a good basis to combine k-resilience techniques in order to achieve further higher availability for applications through the duration of their life.

VII. RELATED WORK

There has been a large amount of recent research on placement optimization for cloud-based applications. [5][6] optimizes multi resource sharing for concurrent tasks to deliver performance fairness for users. [8] optimizes database placement subject to workloads. And [32] and [17] respectively use heuristic packing and flow model (a variant version of the BL strategy) to schedule tasks, reducing the volume of data transfer across data centers or clusters. These approaches attempt to find adequate resource placement choices for applications to achieve better performance for users, but availability of computation is ignored.

Cloud-based resource-centric availability attracts many studies on high-availability aware placement. [23][15] introduces efficient placement of virtual machines on cloud infrastructure with consideration of resource availability. Schlichting et al. [19][20] used heuristic packing to search good placement of cloud-based multi-tier applications for high-availability with ensured performance. These placement strategies target improving the availability of some resources and services, but do not take account of the availability of different classes of application requests. An effective approach is required for improving end-to-end availability of request processing.

Request-oriented computation is a basis of many applications in clouds, [18] aims to bridge the gap between cloud-based resources and performance of requests delivered by users. Availability-aware object placement for multi-object operations can be seen as a simplified version of request oriented aware placement. [31] discusses the correlation of data placement and the availability of multi-object operations. It demonstrated that the high availability of the data host is not enough for giving high availability of multi-object operations. [34] formulated this problem as a high order nonlinear objective function for seeking good placement for object replicas. Different to these co-allocation problems on object placement, an errant task executor may cascade failures to other task execution in connection. The problem of task executors relates not only to resource availability, but also strongly relevant to the issues of software quality and resilience. Downtime of an executor may affect different classes that request the service. These factors affecting application availability have been taken into consideration in our approach.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented a request oriented aware placement approach to improving the availability of deployed software applications in cloud platforms. The placement problem is represented as a MILP problem where it considers several edges of conditions on performance and availability, including workload distribution across different replicas, failure of infrastructure resources, failure of executors, multi-tenancy of task execution, and different classes of application requests, etc.
In principle, our approach can be easily adapted to cope with existing system design and resilient management strategies, giving advantages to improve the availability of different classes of requests in a multi-tenant computing environment that consists of many applications. It is a step forwards, improving availability of deployed software applications in cloud environments. A tool has been implemented for application deployment on top of the AWS cloud infrastructure.

Our current approach linearizes exponential programming as an MILP. To further improve the scalability of our implementation, simplifications of the optimization (e.g., preliminary clustering of applications) are required to cope with the limitations of available MILP solvers such as CPLEX. Dealing with these limitations will be considered in our future work.

REFERENCES