Edge Capacity Planning for Real Time Compute-Intensive Applications

Marius Noreikis*, Yu Xiao* and Yuming Jiang†
*Department of Communications and Networking, Aalto University
Espoo, Finland
{marius.noreikis, yu.xiao}@aalto.fi
†Department of Information Security and Communication Technology, Norwegian University of Science and Technology
Trondheim, Norway
yuming.jiang@ntnu.no

***Abstract*—Cloud computing is a major breakthrough in enabling multi-user scalable web services, process offloading and infrastructure cost savings. However, public clouds impose high network latency which became a bottleneck for real time applications, such as mobile augmented reality applications. A widely accepted solution is to move latency sensitive services from the centralized cloud to the edge of the internet, close to service users. An important prerequisite for deploying applications at the edge is determining initial required edge capacity. However, little has been done to provide reliable estimates of required computing capacity under Quality-of-Service (QoS) constraints. Differently from previous works that focus only on applications' CPU usage, in this paper, we propose a novel, queuing theory based edge capacity planning solution for real-time compute-intensive applications that takes into account usage of both CPU and GPU. Our solution satisfies the QoS requirements in terms of response delays, while minimizing the number of required edge computing nodes, assuming that the nodes are with fixed CPU/GPU capacity. We demonstrate the applicability and accuracy of our solution through extensive evaluation, including a case study using real-life applications. The results show that our solution maximizes the resource utilization through intelligent combinations of service requests, and can accurately estimate the minimal amount of CPU and GPU capacity required for satisfying the QoS requirements.**

***Index Terms*—Capacity planning, queuing theory, edge computing, GPU, augmented reality**

I. Introduction

In recent years public cloud computing became highly attractive to application developers by providing virtually unlimited and highly available computing resources on demand. However, the centralization of computing capacity has certain drawbacks. For example, the round-trip time between a centralized cloud and a mobile client is relatively high, ranging from hundred milliseconds to seconds [13]. The high delay makes it challenging to satisfy the QoS requirements for real-time applications, such as mobile Augmented Reality (AR) and autonomous driving. To satisfy the QoS requirements in terms of the tolerable response delays, researchers proposed to deploy real time applications at the edge of the internet [22]. Researchers also proved that moving latency sensitive computation to the edge fulfills the most demanding QoS requirements [10], [19].

A typical example of an edge node is a small computing device located in wireless access networks. With an increasing demand for compute-intensive multimedia data processing in real time, in addition to central processing units (CPU) an edge node typically contains also graphics processing units (GPU). The diversity of processors increases the complexity of edge capacity planning. A new type of resource (GPU in this case) must be taken into account in both theoretical capacity planning models and simulators used for capacity estimation.

Numerous research efforts were made for task scheduling [15], and offloading and mobility management [16] for edge clouds. However, research on edge capacity planning is still very limited, especially considering diverse computing resources, such as CPU and GPU. Present works on public cloud capacity planning [2], [4], [11] considered only the CPU utilization. Besides, regarding the methods of capacity planning, benchmarking was commonly used in previous works [8], [17]. While benchmarking can be applied to the capacity planning of multiple resources, it requires extensive simulations whenever QoS requirements change or a new functionality is introduced to the application in question.

In this work, we propose an edge capacity planning solution that takes into account both CPU and GPU usages for different types of applications. Our solution aims at minimizing the required number of edge nodes under QoS constraints, assuming that each edge node is with fixed CPU/GPU capacity. We classify computing tasks (e.g. detecting objects in a photo) generated by service requests into 3 resource utilization types based on their levels of demand for CPU/GPU resources. Instead of benchmarking, we apply queuing theory, more specifically, the *M/M/k* model, to estimate the edge node counts for different combinations of task categories and QoS constraints. We evaluate the applicability and accuracy of our solution through a case study using real-life applications.

Contributions of this work are threefold:

1) We formulate the problem of QoS-oriented capacity planning as an estimation of minimal edge node counts with fixed CPU/GPU capacity each, and show how to apply the M/M/k model to minimize the node counts while satisfying the QoS requirements.

TABLE I
RESOURCE DEMAND AND QOS REQUIREMENTS FOR SYSTEM'S TASK CATEGORIES. LOW/MEDIUM/HIGH UTILIZATION: <20% / 20∼70% / >70%; LOW/MEDIUM/HIGH LATENCY: <500MS / 500MS∼2S / >2S;

Task category	Resource demand obtained via profiling			Resource utilization Type	Mean required response delay / task execution time	Expected requests frequency λ ($\frac{1}{s}$)
	CPU utilization	GPU utilization	Net. latency			
Localization	high	high	high	1	5s	0.2
Object recognition	low	high	low	2	250ms	2
Video streaming from server to client (Streaming)	low	-	medium	3	1s / 60s	$\frac{1}{300}$
Search	medium	-	medium	3	1s	0.2
Browsing	low	-	medium	3	1s	0.2

2) We propose to utilize weighted average approach for accurately estimating edge node counts when several tasks of different categories are executed in parallel.
3) We validate accuracy of the proposed solution by comparing the performance of applications between different edge capacity plans. The results prove that our solution can estimate the minimal edge node counts with high accuracy.

The rest of the paper is structured as follows: we highlight the motivation of this work and introduce the M/M/k model in Section II. We then introduce the design of the proposed capacity estimation model and the node count estimation algorithm, followed by its implementation details in Section III. We illustrate the evaluation of the proposed methods and tools in Section IV. We summarize the related work in Section V and discuss the limitations and future studies in Section VI. Finally, Section VII concludes the paper.

II. MOTIVATION AND SYSTEM MODEL

In this paper we consider a scenario of a practical AR based navigation and information system. Its main functionality includes positioning and navigation services, providing textual and video content to a user, and detecting real world objects in AR view. In this scenario, venue visitors use a mobile application to navigate indoors and to get relevant real-time information about nearby points of interest through an AR view. Table I summarizes the core system functionality including the following task categories:

1) *Localization*, where a server computes an indoors location of a person by processing a supplied query photo. This task involves heavy computer vision based processing and requires both CPU and GPU resources [6].
2) *Object recognition*, where a server recongnizes objects in the images captured and supplied by a user. Typically object recognition is done on a video feed, meaning that a new image for recognition is supplied in almost real time. GPU-based Convolutional Neural Network (CNN) implementations are used in order to achieve required accuracy and low response delays. Thus, this task category heavily utilizes GPU resources.
3) Video *streaming* from a server to a client, which is used to e.g. show videos about the recognised objects and typically requires substantial network resources.
4) Content *search* to issue queries against an information database. Typically it shows moderate CPU usage.
5) Web *browsing* for serving static information to a user.

Throughout the paper we will refer to the task categories by their names written in *italics*. Every task category has different resource and QoS requirements. For example, real-time *object recognition* cannot tolerate a mean response delay longer than 250ms, *localization* requests must be fulfilled within 5 seconds, and *browsing* and *search* requests should not take longer than a second. *Streaming* should also start within a second after a streaming request is received. In certain systems same task categories may require different QoS. For example, certain search queries must be completed faster than others or have a different expected request frequency. In this case, each different set of QoS requirements would result in a new task category, e.g. *Regular Search* and *Fast Search*.

Several challenges must be solved before deploying such a system. Firstly, the algorithms for processing *localization* and *object recognition* tasks are highly compute-intensive, and require both CPU and GPU resources. Secondly, the system must satisfy strict QoS requirements in terms of response delays. Finally, it must ensure high QoS for multiple simultaneous users. To solve the aforementioned challenges, accurate edge capacity planning must be conducted to satisfy required QoS requirements, and to minimize edge acquisition costs before deploying the application.

We formulate the edge capacity planning problem as an optimization problem that minimizes the required amount of edge nodes with fixed CPU/GPU capacity each while satisfying QoS requirements of the system to be deployed. We define two types of edge computing nodes. One is equipped with CPU only, while the other provides both CPU and GPU resources. We assume identical CPU capacity for both types of edge nodes. The GPU capacity is also identical for the second type of nodes.

We propose to use queuing theory for edge node count estimation. This prevents extensive benchmarking and helps to implement a more general capacity planning solution. Queuing

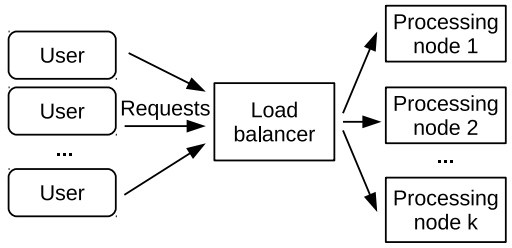

Fig. 1. Edge layer structure. All the user requests first arrive at the load balancer, before they are assigned to any edge node for task execution. Transmission latency between edge nodes is negligible, and commonly a request size is $\ll$ memory size of the load balancer.

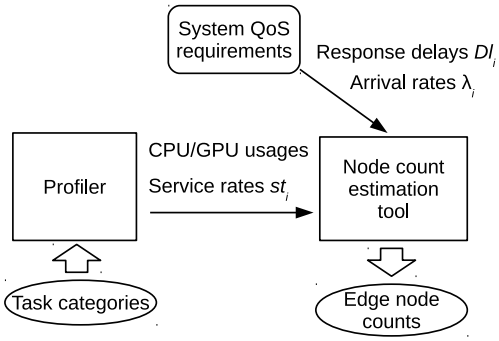

Fig. 2. Architecture of the edge capacity planning solution

theory has been widely applied in computing and telecommunications fields [9], [14]. A typical queuing model consists of 4 parameters: 1) requests arrival process, 2) processing times distribution (service process), 3) number of processors or servers, and 4) queuing buffer size and type [24]. Arrival and service processes usually follow Poisson [12] (exponential) distribution (denoted by M), deterministic model (denoted by D) or general distribution (denoted by G). The third parameter, usually denoted by k, defines the number of *processors*. In our case k represents the number of *edge nodes*. The buffer size defined by the fourth parameter can be either finite or infinite.

In our solution, we assume that for each category of requests, the request arrival rates and request processing times are exponentially distributed and independent. Therefore, we select the *M/M/k* model [24]. In this model, the requests arrival rates follow a Poisson distribution with parameter λ, and the service rates follow a Poisson distribution with parameter μ. The model assumes that when all the nodes are busy, the incoming requests will be queued in an infinite buffer. In practice, we propose to implement load balancing on a dedicated edge node, which acts as an infinite queue (see Figure 1). The arrival rate λ can be calculated from non-functional application requirements and historic user behavior data, such as the number of hourly venue visitors. Table I indicates request arrival rates (λ) that are obtained from non-functional system requirements. However, μ is calculated from service time st, which refers to the time needed to process a particular request. In practice it is challenging to calculate the value of st, thus we propose to utilize profiling of the developed microservices that handle different task categories to estimate the mean values of st.

As shown in Table I, real-life applications consist of multiple task categories. Therefore, our capacity planning solution must take into account cases when several users issue different requests to a server simultaneously. In such cases multiple task categories are handled in parallel. In our previous work [17], we showed that executing requests that belong to different task categories in parallel helps to better utilize edge resources, and to minimize required node counts. However, we note that such combinations of task categories must be carefully selected based on their resource usage patterns.

We empirically define three levels of resource utilization, namely, low($<$20%), medium(20$\sim$70%) and high($>$70%). Based on that, we define 3 resource utilization types (or Types in short) based on the levels of CPU and GPU usages. Every defined task category belongs to one of the 3 Types:

Type 1 Task categories that require medium or high CPU and medium or high GPU usage. Processing a single task belonging to these categories consumes at least 20% of the total CPU and GPU capacity of an edge node. Example task categories include image based localization [6].

Type 2 Task categories that require medium or high GPU usage but low CPU usage. Processing such types of tasks requires less than 20% of the total CPU capacity but needs notably higher percentage of GPU utilization. Examples include CNN based object recognition [18], 3D graphics rendering and image processing.

Type 3 Task categories that do not require GPU usage. Such task categories include web browsing, video streaming, and database search.

In Section III we indicate how such subdivision into Types is utilized with the proposed queuing model, while in Section IV we show the benefits of combining different Types for minimizing required edge node counts.

III. Design and Implementation

In this section we present design and implementation of the proposed edge capacity planning solution. The solution consists of tools for obtaining resource utilization and estimating required edge node counts, given the conditions that the CPU/GPU capacity of a single edge node is fixed and known beforehand. More specifically, we present 1) a *Profiler* that is used to obtain resource requirements for each task category, and 2) the node count estimation tool, which applies queuing theory to calculate the total number of edge nodes required to fulfill the QoS requirements of a system to be deployed. The output of the *Profiler* is used as input for the node count estimation tool (see Figure 2).

A. Profiler

We developed the *Profiler* by following an approach described in [17] to analyze resource usage patterns of each task category and to determine the mean response delays for

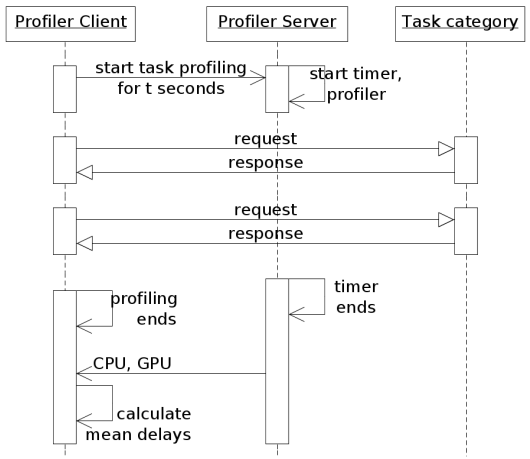

Fig. 3. Sequence diagram of the profiling tools deployed on client and server nodes

requests of the category. Figure 3 illustrates the working mechanism of the *Profiler* tool. It executes on both client and server sides. Let us assume that the system must support a set of task categories T_i, $i \in [1, n]$. Each task category has different demands for CPU and GPU resources. First, microservices that handle requests belonging to different task categories have to be deployed on a server (i.e. an edge node), where the profiling will take place. Alongside the microservices, we also deploy the server-side profiling tool that records resource usages on the server side. On a client (*e.g.* mobile device), we deploy the client-side profiling tool that issues requests to the server and obtains usage data after the requests are completed. In this way, the server-side tool can accurately profile the CPU and GPU usages with negligible interference from the client-side tool. Differently from the previous work [17], our tool also measures the service time spent on processing requests that belong to different task categories st_i(s), $i \in [1, n]$. Note that the measurement of response delay includes also the network latency. In order to obtain the most accurate results the *Profiler* should run on the same types of edge nodes which will later be used to deploy the system.

B. Node count estimation with queuing theory

The node count estimation algorithm is the key part of the node count estimation tool and comprises two parts: (1) estimating the number of edge nodes required for handling different combinations of tasks that all belong to the same Type (i.e. Type 1, 2 or 3), and (2) estimating the total number of edge nodes required for supporting all combinations of tasks belonging to different types of task categories, in other words for supporting the overall application in question.

1) Node counts for task combinations: We start with different combinations of tasks that all belong to Type 1. We calculate the number of required nodes, given a service time st, mean processing delay Dl, and arrival rate λ. For the selected $M/M/k$ model we utilize Eq. 1 to calculate the mean response delay $E[D]$.

$$E[D] = \frac{C_k(A)}{\mu k - \lambda} + \frac{1}{\mu} \tag{1}$$

Here, $C_k(A)$ is an Erlang C [5] formula where k stands for node count, $A = \lambda/\mu$ refers to offered traffic, and $\mu = 1/st$ is a service rate. Variable λ represents the arrival rate of requests and is obtained from QoS requirements multiplied by the number of simultaneous system users. Here we assume that the number of users using the system is known in advance, for example from service usage history or from predicted user flows. However, we allow each task category to define the expected arrival rate independently. The value of k can be updated whenever the arrival rates change.

We use the output from the *Profiler* to find out the mean completion time (service time) st_i for a request of a particular task category T_i and its CPU and GPU requirements. Assuming that the maximum response delay for i-th task category is Dl_i, we ensure that $st_i < Dl_i$. This means that the particular edge node can process a task within its maximum tolerable response delay. We also ensure system stability, where arrival rate must be lower than service rate, i.e. $k\mu > \lambda$. Otherwise it would lead to infinite queues over a long period of time. Thus, we set the initial $k = \lfloor \lambda/\mu \rfloor + 1$. Since we need to take into account required response delays Dl_i, we must ensure $E[D] \leq Dl$. Therefore we use dimensioning where we increase k by 1 until the response delay constraints are satisfied, or $k_{max} = 10^5$ is reached. If k_{max} is reached, it means that current servers cannot satisfy the required delay constraint and the response delay constraint should be relaxed or a different type of edge nodes must be used. The first k with which $E[D] \leq Dl$ represents the required node count.

Similar computations are done with task combinations that fall into Type 2, where k refers to the number of required GPU enabled nodes. Task combinations of Type 3 require a slightly different way to estimate k. For Type 3 we may have several CPU intensive tasks that are executed simultaneously. Thus, in order to use Eq. 1, we need to calculate new service rates (μ) and arrival rates (λ). In this work we adopt the weighted average approach. We assume that the service would be running for an arbitrary long period of time T, where different task categories have service times st_i and arrival rates λ_i, $i \in [1..n]$. In this case we can calculate a mean service time $E[st]$ with Eq. 2.

$$E[st] = \frac{\sum_{i=1}^{n} \lambda_i T st_i}{\sum_{i=1}^{n} \lambda_i T} = \frac{\sum_{i=1}^{n} \lambda_i st_i}{\sum_{i=1}^{n} \lambda_i} \tag{2}$$

We obtain mean service rate $E[\mu] = 1/E[st]$ and calculate mean arrival rate $E[\lambda]$ with the same assumptions using Eq. 3.

$$E[\lambda] = \frac{\sum_{i=1}^{n} \lambda_i T}{T} = \sum_{i=1}^{n} \lambda_i \tag{3}$$

Note that Eq. 2 and Eq. 3 also apply if there is only one task category. Based on these equations we design and implement Algorithm 1 to calculate the required node counts for combinations of task categories that all belong to a single

Type (i.e. Type 1, 2, or 3). The algorithm takes task category combinations, arrival rates and required QoS in terms of response delays as input and for each combination estimates the number of required nodes.

Algorithm 1 Calculation of the number of required nodes

Input: Combinations of task categories that belong to the same Type T, number of combinations n, arrival rates λ, maximum task response delays Dl

Output: resource usage and server requirements for each task combination R. k defines the number of required edge nodes

$R \leftarrow [\ ]$ ▷ Results
for $i \in (0..n)$ **do**
  Obtain from *Profiler* st_i(s) and usages of $CPU_i(\%)$, $GPU_i(\%)$
  $\lambda \leftarrow \sum_{j=1}^{n} \lambda_{i,j}$
  $st \leftarrow \frac{\sum_{i=1}^{n} \lambda_{i,j} st_{i,j}}{\lambda}$
  $\mu \leftarrow 1/st$ ▷ Service rate
  $A \leftarrow \lambda/\mu$
  $k \leftarrow \lceil \lambda/\mu \rceil - 1$
  $E[D] \leftarrow \inf$
  while $E[D] > Dl_i$ and $k \leq k_{max}$
    $k \leftarrow k + 1$
    $E[D] = C_k(A)/(\mu k - \lambda) + 1/\mu$ ▷ mean response time
  $R_i \leftarrow (k, st, CPU_i, GPU_i)$
return R

2) Required node count for all tasks: In the previous step we developed an algorithm to estimate required edge node counts for different combinations of tasks that all belong to a single Type. Now we move to the method for estimating the edge node counts to cover all task categories that belong to different Types.

A requirement of having a GPU greatly increases the price of a server. We aim to minimize the overall acquisition costs of edge nodes, therefore we firstly estimate the number of GPU enabled nodes needed for the whole system. As shown in [17], certain types of requests can be combined to improve resource utilization. We utilize combinations of task types in the following way: when an edge node is processing tasks that all belong to the Type 2 task categories, the node's CPU is mostly idle and can be used to process tasks that belong to the Type 3 task categories. However, Type 1 cannot be combined with other Types, since it would highly influence the response delays of all the tasks involved. Thus, we first need to estimate the required node counts for Type 1 requests, followed by Type 2 and finally Type 3 (as those can be combined with Type 2). We start by obtaining the number of nodes with GPUs for handling Type 1 tasks, which we denote by C_1. Similarly, we denote the GPU enabled node count for Type 2 tasks by C_2. We utilize a straightforward Eq. 4 to calculate the total number of required edge nodes that contain GPUs:

$$C_{GPU} = C_1 + C_2 \tag{4}$$

For Type 3, we denote the required node count by C_3. Since Type 3 task categories can be processed together with Type 2 ones on the same nodes, we obtain the number of required edge nodes that only contain a CPU by Eq. 5.

$$C_{CPU} = max(0, C_2 - C_3) \tag{5}$$

It may happen that we even do not need any additional nodes to support task categories of Type 3 if we have enough unutilized CPU processors dedicated for task categories of Type 2.

Finally the total number of required edge nodes to support the whole system is obtained with Eq. 6.

$$C_{total} = C_{GPU} + C_{CPU} \tag{6}$$

IV. Evaluation

In this section we evaluate the proposed node count estimation algorithm. We firstly identified how the proposed algorithm estimates response delays for each single task category. Secondly, we investigated the algorithm performance when calculating response delays for various task combinations. Thirdly, we examined how accurately the proposed method estimates the maximum number of simultaneous users, given a number of edge nodes. Such evaluations gave us a valuable insight on the applicability of the *M/M/k* model and weighted average approach for node count estimation. Finally, we evaluated the developed algorithm while planning the deployment of a practical real time system.

A. Experiment setup

We evaluated our node count estimation algorithm with the task categories presented in Table I. The implementation of each task category is described below.

We implemented the *localization* functionality by following the same approach of an image-based indoor localization system presented in [6]. The *object recognition* tasks were performed by utilizing YOLO [18], a real-time CNN based object detection system. For video *streaming* we used a Big Buck Bunny[1] video, and streamed it at a rate of 2500kbps. For *search* we utilized a full text search within a relational PostgreSQL[2] database filled with 25000 records. Finally, for the *browsing* task category we served a 1.5MB web page. We utilized Docker[3] containers to deploy each task category. We utilized microservices architecture where every task category is deployed as a self contained microservice. Such containerized microservices architecture allows easy scaling and deployment of the services on any cloud infrastructures.

In this work we focus on edge node count estimation. Accordingly, we simulated an edge computing environment within a public cloud, ensuring that the network latency between two different computing nodes agrees with edge computing scenario. We utilized Amazon Web Services (AWS) public cloud and used p2.xlarge instances as server edge

[1]https://peach.blender.org/
[2]https://www.postgresql.org/
[3]https://www.docker.com/what-docker

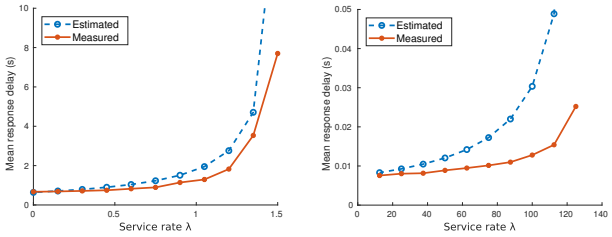

Fig. 4. Calculated and measured mean response delays for localization (left) and browsing (right) requests with different arrival rates λ

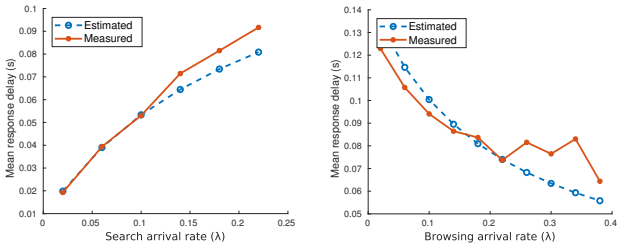

Fig. 5. Calculated and measured mean response delays when *browsing* and *search* requests are processed simultaneously. Fixed *browsing* arrival rate ($\lambda = 3$) and variable *search* arrival rate (left). Fixed *search* arrival rate ($\lambda = 1.5625$) and variable *browsing* arrival rate (right).

nodes and a c4.xlarge instance as a client. The p2.xlarge instance contains one NVIDIA K80 GPU, 4 core Intel Xeon CPU and is amongst the cheapest GPU enabled public cloud instances. According to [17], multiple low capacity nodes outperform fewer but more expensive and higher capacity ones, therefore p2.xlarge was the most suitable instance type for our system[4]. The c4.xlarge instance contains 4 core Intel Xeon CPU but lacks GPU. All instances were in the same region and availability zone, thus in the same subnet with a single network hop between any two machines. The average measured network latency between the machines was less than a millisecond.

B. Average response delay

As discussed in Section II, we utilized an *M/M/k* queuing theory model to estimate the number of edge nodes required to serve a given number of users while satisfying the QoS requirements. In order to check whether our assumed model fits a real world scenario and whether an estimated delay is within reasonable bounds, we simulated two scenarios. In the first one, a client is sending requests belonging to a single task category, either a resource intensive *localization* or a light weight *browsing* request. In the second scenario, the requests cover a combination of different task categories, such as a combination of *search* and *browsing* requests.

In our experiments, clients continuously sent requests to the server, waited for an exponentially distributed time T between subsequent requests. We utilized the python

[4]The reason to use an instance with higher capacity would only arise, if the current low-capacity node could not fulfill the required QoS even for a single request.

TABLE II
NUMBERS OF SIMULTANEOUS USERS SUPPORTED BY DIFFERENT NUMBERS OF EDGE NODES WHEN PROCESSING LOCALIZATION REQUESTS

Edge node count	Estimated No. users	Measured No. users
1	5	6
2	12	12
3	19	20

TABLE III
NUMBERS OF SIMULTANEOUS USERS SUPPORTED BY DIFFERENT NUMBERS OF SERVERS WHEN PROCESSING A COMBINATION OF SEARCH AND BROWSING REQUESTS

Edge node count	Estimated No. users	Measured No. users
1	30	32
2	64	65
3	97	101

`random.expovariate(`λ`)`[5] function to obtain T values. If a server is already processing a request when a new one arrives, the newly arrived request is placed into an unbound first-in-first-out (FIFO) queue. After a server becomes available, it takes a request from the head of the queue. We recorded response delays on the client side for each request until we observed at least 1000 responses. We disregard the first and last 10% of the requests to prevent errors introduced by system warm-up and cool-down periods, respectively. We then calculated a mean response time and compared it with values calculated with Eq. 1, where $k = 1$.

Figure 4 compares the estimated and measured response delays for localization and browsing requests with different arrival rates. In case of localization the figure clearly shows that both estimated and measured values follow the same trend and are reasonably close to each other. However, in the browsing case we can observe that the measured values are roughly twice lower than the estimated ones. This can be explained by analyzing response delays of browsing requests. During the experiment, the mean response delay was 0.0075 ($s = 0.001$). Therefore the service time for browsing is closer to a deterministic distribution rather than an exponential one. According to Tijms [21], the waiting time probabilities for a deterministic distribution are roughly half of those of an exponential distribution. For this particular case we could utilize an M/D/k queue, however, we note that deterministic service times may not be common in most applications. Furthermore, estimated values that show longer response delays provide a "pesimistic" estimation, which is useful to make sure that our algorithm provides node count estimates that can fulfill the worst case scenarios.

A combination of *search* and *browsing* tasks was chosen to evaluate how accurately our proposed algorithm estimates the response delays when different task categories belonging to the same type are involved. These task categories have different service time and QoS requirements. We started a client that sends requests and waits between subsequent ones according to two different arrival rates: $\lambda_{browsing}$ and λ_{search}. For the first case we fixed $\lambda_{browsing} = 3$ and varied $\lambda_{search} \in [0.1 \cdot \lambda_{browsing}, 1.1 \cdot \lambda_{browsing}]$. For the second case, we fixed

[5]https://docs.python.org/2/library/random.html#random.expovariate

λ_{search} and varied $\lambda_{browsing} \in [0.1 \cdot \lambda_{search}, 1.9 \cdot \lambda_{search}]$. The response delays of both cases are show in Figure 5, which proves that the estimated and measured response delays for this task category combination are similar and follow the same trend.

C. Maximum supported simultaneous users

Before evaluating the accuracy of the node count estimation algorithm for a given number of simultaneous users, we conducted a reverse evaluation, where we evaluated how many simultaneous users a certain number of computing nodes can support. This allowed us to run simulations more easily and efficiently as we only needed to reserve a few public cloud machines. At the same time, it provided useful insight into the algorithm performance.

We utilized *localization* tasks and a combination of *search* and *browsing* tasks to check whether the proposed algorithm accurately estimates the maximum number of simultaneous users supported by 1, 2 and 3 nodes, respectively. For localization requests, we fixed the mean service rate at $\mu = 1.5625$ and utilized $\lambda = \lambda_{localization} \cdot users\_count$ request arrival rates, where $users\_count$ represents the number of simultaneous localization service users. We kept increasing $users\_count$ and observed k values estimated by the node count estimation algorithm. The user counts for different k values are presented in Table II. In order to obtain an actual maximum number of supported concurrent users, we have developed a benchmarking tool by following the same approach described in [17]. Afterwards we ran the tool to obtain the numbers of simultaneous users that a system with 1, 2 or 3 servers supports. Table II shows the maximum number of simultaneous users estimated by our algorithm versus the maximum number of simultaneous users measured by the benchmarking tool. As the table shows, there is only a slight difference between the estimated and measured user counts, meaning that our algorithm is suitable for estimating node counts for a single task category. A similar experiment was conducted with a combination of *search* and *browsing* which are task categories belonging to Type 3. As Table III shows, the estimation error in the number of users is also small, proving that the same algorithm can reliably estimate the edge node counts when multiple task categories are involved.

D. Case study

Finally, we evaluated how well the proposed edge node count estimation algorithm handles real world use cases by planning the required capacity for a system composed of task categories introduced in Table I. For that we utilized QoS based cloud capacity planning framework from [17] and substituted node count estimation step with our proposed algorithm. We also focused solely on edge layer, thus we omitted a step of classifying task categories for either edge or public cloud layers. Eventually, we executed the following steps to plan the required system capacity:

Step 1. We utilized the *Profiler* to obtain resource usage patterns of each task category (see Figure 6). We configured the *Profiler* to run for 30 seconds, which is long enough to accurately record resource usage patterns.

TABLE IV
ESTIMATE OF THE NODE COUNTS REQUIRED BY EACH TASK CATEGORY TO SUPPORT 13 SIMULTANEOUS USERS.

Resource utilization type	Task categories	Required node count
Type 1 (CPU & GPU)	Localization	2
Type 2 (GPU)	Object recognition	3
Type 3 (CPU)	Streaming, search, browsing	3

Step 2. We identified task categories with complementary resource demands. According to the profiling data (see Figure 6) all of the task categories required CPU processing but *object recognition* and *localization* additionally made use of GPU. Regarding *object recognition*, utilization of processors was quite steady with a high GPU utilization and a low CPU utilization, while *localization* required both high CPU and high GPU utilization. Therefore we listed *localization* as Type 1 and *object recognition* as Type 2 task categories. Other task categories fell into Type 3, since they required no GPU processing.

The results suggest, that a node which is running *object recognition* can as well serve *streaming* and *browsing* requests in parallel. In case of *localization*, it falls under Type 1 where any combinations would influence the request completion time, thus the localization-related microservices would be deployed on separate nodes.

Step 3. At this point we determined the number of required nodes. We executed node count estimation tool with each combination of task categories as input that belong to the same type (i.e. Type 1, 2 or 3). We assumed a small scale scenario where there are at most *13* simultaneous users at a time. We chose this number to be able to verify algorithm accuracy with our edge environment that consisted of 6 nodes (including load balancer). After execution, the node count estimation tool provided the results shown in Table IV. We utilized Eq. 4 to obtain the number of required GPU equipped nodes $C_{GPU} = 2+3 = 5$ (2 for Type 1 and 3 for Type 2) and Eq. 5 to get the number of additionally required CPU nodes $C_{CPU} = max(0, 3-3) = 0$. We concluded that in order to deploy the system we need 2 dedicated nodes for Type 1, and 3 nodes for Type 2. We also noted that for Type 3, we can execute requests on the same 3 nodes used for Type 2. In other words, we did not require any additional nodes to run tasks that fall into Type 3.

Step 4. Lastly, we utilized a small scale public cloud setup to simulate an edge cloud infrastructure and verified the results of node count estimation. We deployed 5 server nodes, a load balancer, and a node that generates client requests. We configured our load balancer to forward *localization* requests to nodes 1 and 2, whereas all the other requests were being forwarded to nodes 3-5. We utilized the benchmarking tool to check the maximum amount of supported users. Figure 7 shows response delays of each task category for different num-

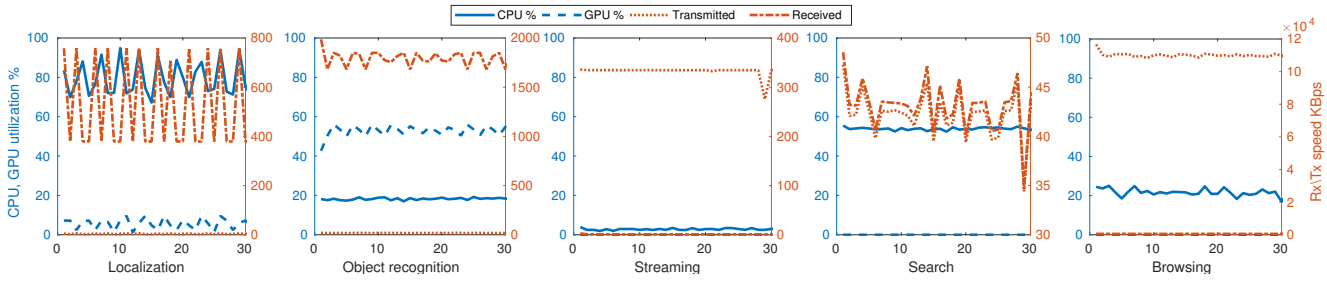

Fig. 6. Resource utilization for *localization*, *object recognition*, *streaming*, *search* and *browsing* task categories, respectively. X-axis represents time in seconds. Tests were run for 30 seconds.

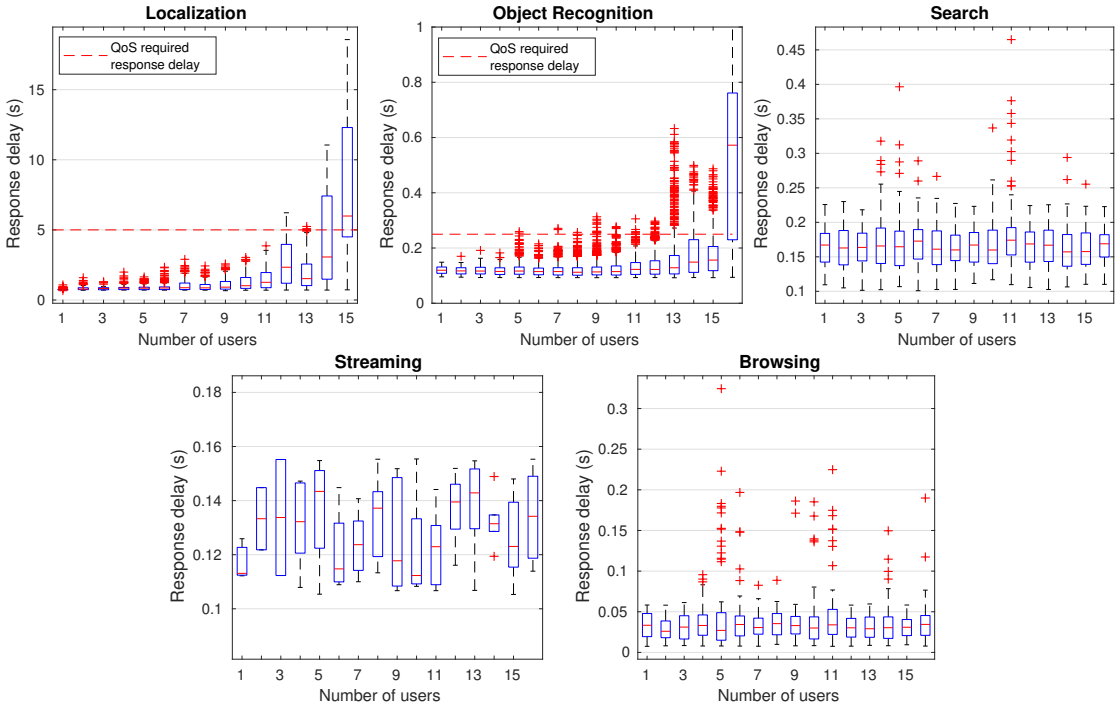

Fig. 7. Response delays for each task category with different numbers of simultaneous users when the system is running on the planned edge layer deployment. The marks on each box represent mean values, while the bottom and top edges of the boxes show 25th and 75th percentiles, respectively. Plus signs indicate outlier measurements, that were not included in the mean calculation. Horizontal dashed lines indicate mean required response delays (not visible for *streaming*, *search* and *browsing* requests as their response delays were well below the required ones).

bers of simultaneous users. It clearly indicates that *localization* and *object recognition* cannot fulfill the required QoS once the number of concurrent users exceeds 14 and 15, respectively. With this setup, edge nodes that served *localization* requests could support up to 14 simultaneous users ($14 - 13 = 1$ user difference from the estimated user count), while the other nodes that served *object recognition* (and at the same time all the remaining requests) could support up to 15 simultaneous users ($15 - 13 = 2$ users difference from the estimated count). The use case evaluation shows that our algorithm can accurately estimate the required capacity in terms of the number of the computing edge nodes within a small error margin.

V. Related work

This section briefly presents the related work in the field of cloud/edge capacity planning and points out differences from this work.

A. Capacity planning for cloud computing

Capacity planning for cloud computing has been widely studied. For example, Jiang et al. [11] introduced a cloud capacity planning and virtual machine provisioning solution based on workload predictions. Service Level Agreement (SLA) violations and costs of under provisioned resources were the main factors considered in the solution. Candeia et al. [2] proposed methods for increasing SaaS providers' profits by utilizing business driven capacity planning heuristics. Brunnert et al. [1] proposed to utilize resource usage profiles that are obtained during application development stage to estimate

the required server capacity. Carvalho et al. [4] proposed analytical models to estimate the minimum capacity that fulfills required SLAs. In addition, researchers distinguished different classes of SLA to better utilize cloud instances and proposed a quota-based admission control mechanism for rejecting lower SLA requests during peek times to ensure fulfillment of other classes' SLAs [4]. The researchers applied two queuing models, G/GI/c/K and M/M/c/K, for server capacity planning in the scenario of admission control. In this work, we propose to utilize a multi-server infinite queuing model M/M/k for edge capacity planning.

Goncalves et al. [8] showed that benchmarking based approach is suitable for obtaining optimal IaaS public cloud arrangements that minimize required instance counts. The process starts by deploying a certain workload on a cloud configuration of an initial capacity. Then, the workload is processed and depending on whether the processing satisfied a required SLA, the capacity is either decreased (if satisfied) or increased (if not satisfied), until an optimal capacity is reached. While the approach is able to estimate close to optimal capacity, the selection of initial capacity influences the number of different capacity deployments that have to be tested.

Even though some of the ideas can be applied in an edge cloud infrastructure, there are key differences and considerations when deploying an application on edge or hybrid clouds instead of centralized clouds. For example, edge clouds are typically not as scalable as public ones, thus initial capacity planning needs to ensure required QoS during peak usage times. Moreover, edge clouds are mostly utilized to accommodate real time applications that have stricter QoS requirements than applications deployed on public clouds.

B. Capacity planning for edge computing

Several researches focused on capacity planning for edge clouds. Zhang et al. [23] proposed a workload management system for video streaming services on hybrid clouds. The service primarily runs on edge nodes and utilizes a public cloud at peak times. The researchers showed that their solution can process up to 95% of requests solely on edge nodes. In our previous work [17], we proposed a benchmarking based capacity planning framework for hierarchical edge clouds. We have developed tools for analyzing application resource requirements and determining required edge node counts. However, in these works, the node count estimation was based on benchmarking of the system and while it showed accurate results there were a few limitations. New simulations had to be performed whenever QoS requirements changed or new task categories were introduced. In this work we propose a more generic solution that eliminates such need for simulations. We only require initial profiling of different categories of tasks to obtain their resource usages and mean execution times after microservices that implement the task categories are developed.

C. Multiple types of resources

While considerable amount of research focused only on CPU utilization, other researchers admitted that different resources must be taken into account. Carvalho et.al. proposed a capacity planning solution that takes into account both CPU and memory requirements [3], whereas Song et.al. [20] took into consideration CPU, network and memory resources. Brunnert et al. [1] proposed to distribute resource usage profiles alongside application binaries to support different types of resources. The application developers had to specify which resource and how much would the developed application need. In this work, we focus on CPU and GPU utilization as we consider these resources as the most important ones for real time applications. Differently from [1], we do not require prior knowledge of system performance or implementation details, as we infer resource usage profiles with automated profiling.

VI. Discussion

In this work we focused on edge capacity planning within Application Model [7]. We did not investigate mobility patterns of service users and spatial deployment schemes of wireless access points and computing nodes. Instead, we assumed that users experience similar edge network latency throughout the area. We also assumed that there is no restriction on the amount of nodes to be deployed. In practice, there may be various situations where edge network shows different network latencies. However, our proposed algorithm can still be applied in estimating node counts for the specific edge areas separately. Furthermore, being fast to compute, the algorithm is suitable for dynamic edge deployment scenarios, when one has to frequently estimate how many nodes to allocate from the edge layer node pool to support a specific service.

Our proposed algorithm estimates edge capacity for nodes with same capacity processors. As a future work we aim to investigate how the algorithm should be improved to take into account nodes of variable capacity. Furthermore, it is worth investigating applicability of the proposed model when other resources are involved, such as memory and network utilization. In this work we assume an infinite size of the buffer where incoming requests are stored when all servers are busy. However, as a future work we plan to investigate applicability of an *M/M/k/c* queue. In this queue, c represents the maximum number of requests that can be queued at a time (including requests in service).

VII. Conclusions

We presented a novel edge node count estimation solution for initial edge layer capacity planning for compute-intensive real time applications. The solution ensures required QoS in terms of response delays and minimizes system deployment costs by minimizing the amount of edge capacity in terms of the number of edge nodes with fixed capacity each. Our solution takes into account both CPU and GPU requirements and minimizes the number of required nodes by utilizing combinations of tasks with different resource demands. We applied the queuing model *M/M/k* and weighted averaging

approach to estimate required node counts for different task category combinations. Our experiments including case study using real life applications showed the applicability and high accuracy of the *M/M/k* model for estimating the minimal edge node count under the QoS constraints for real-time compute-intensive applications.

ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825496, and Academy of Finland under grant number No. 317432.

REFERENCES

[1] A. Brunnert and H. Krcmar. Continuous performance evaluation and capacity planning using resource profiles for enterprise applications. *Journal of Systems and Software*, 123:239–262, 2017.

[2] D. Candeia, R. A. Santos, and R. Lopes. Business-driven long-term capacity planning for saas applications. *IEEE Transactions on Cloud Computing*, 3(3):290–303, 2015.

[3] M. Carvalho, F. Brasileiro, R. Lopes, G. Farias, A. Fook, J. Mafra, and D. Turull. Multi-dimensional admission control and capacity planning for iaas clouds with multiple service classes. In *Proceedings of Cluster, Cloud and Grid Computing, 17th IEEE/ACM International Symposium on*, pages 160–169. IEEE, 2017.

[4] M. Carvalho, D. A. Menascé, and F. Brasileiro. Capacity planning for iaas cloud providers offering multiple service classes. *Future Generation Computer Systems*, 77:97–111, 2017.

[5] E. Chromy, T. Misuth, and M. Kavacky. Erlang c formula and its use in the call centers. *Advances in Electrical and Electronic Engineering*, 9(1):7, 2011.

[6] J. Dong, M. Noreikis, Y. Xiao, and A. Yla-Jaaski. Vinav: A vision-based indoor navigation system for smartphones. *IEEE Transactions on Mobile Computing*, 2018.

[7] S. S. Gill and R. Buyya. A taxonomy and future directions for sustainable cloud computing: 360 degree view. *arXiv preprint arXiv:1712.02899*, 2017.

[8] M. Gonçalves, M. Cunha, N. C. Mendonça, and A. Sampaio. Performance inference: a novel approach for planning the capacity of iaas cloud applications. In *Proceedings of the Cloud Computing, IEEE 8th International Conference on*, pages 813–820. IEEE, 2015.

[9] N. Grozev and R. Buyya. Performance modelling and simulation of three-tier applications in cloud and multi-cloud environments. *The Computer Journal*, 58(1):1–22, 2013.

[10] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan. Quantifying the impact of edge computing on mobile applications. In *Proceedings of Systems, 7th ACM SIGOPS Asia-Pacific Workshop on*, page 5. ACM, 2016.

[11] Y. Jiang, C.-S. Perng, T. Li, and R. N. Chang. Cloud analytics for capacity planning and instant vm provisioning. *IEEE Transactions on Network and Service Management*, 10(3):312–325, 2013.

[12] J. F. C. Kingman. *Poisson processes*, volume 3. Clarendon Press, 1992.

[13] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of Internet measurement, 10th ACM SIGCOMM conference on*, pages 1–14. ACM, 2010.

[14] E. Lisovskaya, S. Moiseeva, and M. Pagano. On the total customers capacity in multi-server queues. In *Proceedings of International Conference on Information Technologies and Mathematical Modelling*, pages 56–67. Springer, 2017.

[15] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief. Delay-optimal computation task scheduling for mobile-edge computing systems. In *Proceedings of Information Theory, IEEE International Symposium on*, pages 1451–1455. IEEE, 2016.

[16] P. Mach and Z. Becvar. Mobile edge computing: A survey on architecture and computation offloading. *arXiv preprint arXiv:1702.05309*, 2017.

[17] M. Noreikis, Y. Xiao, and A. Ylä-Jaäiski. Qos-oriented capacity planning for edge computing. In *Proceedings of Communications, IEEE International Conference on*, pages 1–6. IEEE, 2017.

[18] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[19] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. 3:637–646, 2016.

[20] W. Song, Z. Xiao, Q. Chen, and H. Luo. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Transactions on Computers*, 63(11):2647–2660, 2014.

[21] H. Tijms. New and old results for the m/d/c queue. *AEU-International Journal of Electronics and Communications*, 60(2):125–130, 2006.

[22] L. Tong, Y. Li, and W. Gao. A hierarchical edge cloud architecture for mobile computing. In *Proceedings of Computer Communications, IEEE 35th International Conference on*. IEEE, 2016.

[23] H. Zhang, G. Jiang, K. Yoshihira, and H. Chen. Proactive workload management in hybrid cloud computing. *IEEE Transactions on Network and Service Management*, 11(1):90–100, 2014.

[24] M. Zukerman. Introduction to queueing theory and stochastic teletraffic models. *arXiv preprint arXiv:1307.2968*, 2013.