ATOM: Model-Driven Autoscaling for Microservices

Alim Ul Gias
Department of Computing
Imperial College London
London, UK
a.gias17@imperial.ac.uk

Giuliano Casale
Department of Computing
Imperial College London
London, UK
g.casale@imperial.ac.uk

Murray Woodside
Dept. of Systems and Computer Eng.
Carleton University
Ottawa, Canada
cmw@sce.carleton.ca

***Abstract*—Microservices based architectures are increasingly widespread in the cloud software industry. Still, there is a shortage of auto-scaling methods designed to leverage the unique features of these architectures, such as the ability to independently scale a subset of microservices, as well as the ease of monitoring their state and reciprocal calls.**

We propose to address this shortage with ATOM, a model-driven autoscaling controller for microservices. ATOM instantiates and solves at run-time a layered queueing network model of the application. Computational optimization is used to dynamically control the number of replicas for each microservice and its associated container CPU share, overall achieving a fine-grained control of the application capacity at run-time.

Experimental results indicate that for heavy workloads ATOM offers around 30%-37% higher throughput than baseline model-agnostic controllers based on simple static rules. We also find that model-driven reasoning reduces the number of actions needed to scale the system as it reduces the number of bottleneck shifts that we observe with model-agnostic controllers.

***Index Terms*—microservices, autoscaling, layered queueing network, performance optimization**

I. Introduction

Microservices define a cloud-native architecture for software development [1] that is increasingly accepted in the software industry thanks to their synergy with DevOps [2]. Microservices have evolved from service oriented architecture with the aim of delivering a set of scalable services by decentralizing business logic among fine-grained services [3]. This property results, among other benefits, in greater control of performance since scaling needs for a system can be addressed by adding capacity only to the sections of an application that actually need the extra capacity. In addition, using containers for deploying microservices inherently improves the underpinning performance management thanks to fast start-up times [4] and ease of replication and reconfiguration.

In this paper, we have considered an autoscaling scenario from the context of microservices. For autoscaling cloud applications, rule based approaches are common in the industry [5]. Such approaches usually scale stateless services horizontally and stateful services vertically. However, recent studies [6], [7] have shown that where both horizontal and vertical scaling are applicable, they can provide different performance gains based on the current workload. Thus, depending on the current workload, a scaling decision should be based on the potential performance improvement after applying both vertical and horizontal scaling, either separately or in a combination. This can be achieved using the concepts of queueing models. However, using such concepts for microservices involve multiple research challenges. For accurate performance estimations of microservices, a queueing model should both abstract its explicit properties, like fractional CPU share, and the implicit properties inherited from service oriented architecture, like operating simultaneously as a server and client.

Researchers have proposed multiple methods for autoscaling cloud applications based on meta-heuristics [8], application profiling [9], and analytical methods [10]. However, these methods are not particularly suitable for microservices as they do not consider such platform specific metrics like container start-up time [11] and message queue status [12]. Moreover, it remains the problem on how to tune the capacity estimation process for microservices. Recently, a number of rule based autoscalers have been proposed for microservices [12]–[15]. Some of these scalers utilize message queue metrics [12], which can provide better insights of the system state than traditional container level metrics used in [13]–[15]. However, prior work focuses only on horizontal scaling, whereas we consider both vertical and horizontal scaling.

To aid in this autoscaling context, we present ATOM, a model-driven autoscaler for microservices. For any particular workload, ATOM assesses the effect of the workload on each individual microservice and potential performance improvement after applying various horizontal and vertical scaling configurations. The scaling configurations include the number of replicas for each of the microservices and the allocated CPU capacity (we also used the term "CPU share" interchangeably) for those replicas. To abstract a microservices application and reason on the potential performance gain, ATOM leverages a Layered Queueing Network (LQN) [16], [17] model. Such models are instantiated by host demands determined by monitoring the traffic between microservices. By applying various configurations on a model, ATOM aims to provide a scaling strategy that maximizes the revenue of the system with minimal CPU shares. We have formulated this scaling scenario as a non-linear mixed-integer program using the weighted sum method [18]. To search for an optimal solution for a given workload, considering the non-linear nature of the problem and the presence of integer variables, we have used a genetic algorithm (GA).

Overall, our contributions can be summarized as follows:

- an autoscaling controller that maximizes the revenue of

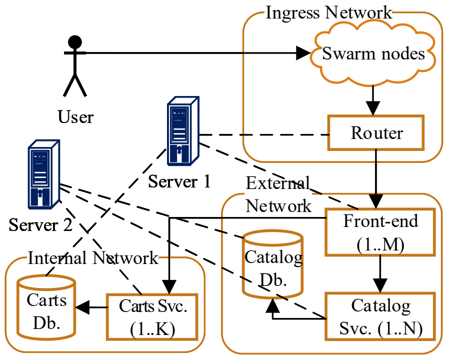


Fig. 1. The architecture of the sock shop application

a microservices application without violating the service level agreement (SLA)

- LQN models to estimate the performance of a microservices application with different number of replicas and CPU shares
- a GA based method that searches a sample space of LQN models to provide an optimal scaling strategy
- an experimentation, using real measurements from a microservices application Sock Shop[1], demonstrating that ATOM outperforms the rule based approaches for heavy workloads

The rest of the paper is organized as follows: Section II presents a motivating example for our work. Section III presents the performance modeling aspect of microservices with LQN. Section IV discusses about our proposed autoscaler ATOM. Section V presents the experiments and results. Section VI discusses the state of the art autoscalers for cloud applications. The paper is finally concluded with future research directions in section VII.

II. Motivating Example

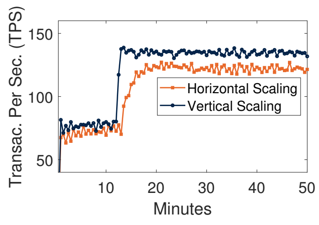
A. Running case

In this example we want to illustrate that scaling strategies for microservices should not be solely based on static rules, such as ‘always scale a stateless microservices horizontally” or “always apply either vertical or horizontal scaling to a specific microservice”. Instead, if a service can be both vertically and horizontally scaled the scaling decision should be driven by workload characteristics. To illustrate this point, we have used a microservices application named Sock Shop. It is an e-commerce website that allows one to view and buy different types of socks. The architecture of the Sock Shop running case, deployed on two Docker[2] swarm nodes, is illustrated in Figure 1. The front-end, catalog and carts services can have up to M, N and K replicas, respectively, which are load balanced by a router service. In addition, the catalog and carts services persist data on private database services. These database services and the router service are stateful, whereas the other services are stateless.

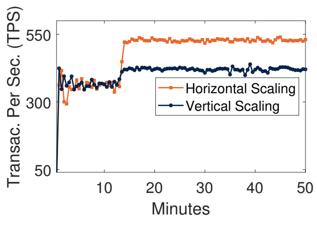
[1]Microservice Demo: Sock Shop [microservices-demo.github.io]
[2]Docker: Enterprise Container Platform [www.docker.com]

TABLE I
Two cases where the front-end microservice is the bottleneck

Case	System Workload					Front-end Config.	
	Request Distribution			Conc. Users	Think Time	CPU Share	Replica
	Home	Catalog	Carts				
A	57%	29%	14%	1000	7 sec	0.2	1
B				4000		1.0	



(a) Case A



(b) Case B

Fig. 2. Effect of vertical and horizontal scaling over the front-end microservice in two different cases

We have considered a browsing workload in this application, which creates a scenario where a user can visit the website and add or remove items in the cart. In this scenario, we have investigated two cases where the front-end microservice is the bottleneck and requires additional capacity. The workload and system setup used in the two cases are summarized in Table I. The workload is specified by the mix of requests, i.e, the number and types of concurrent users issuing synchronous requests, and their think times, the times in-between a request completion and issue of the next request. The system configuration includes the CPU share and the replica number of the front-end microservice. A CPU share of 0.2 means that the microservice can at most utilize 20% of a CPU core, even if the rest of the CPU remains idle. The cases, A and B, represent a light and heavy workload respectively.

B. Comparing scaling rules

For the evaluation of the two cases, vertical scaling was applied by doubling the CPU share of the front-end microservice (the bottleneck); horizontal scaling was applied by doubling the number of replicas of the front-end. The results in Figure 2, for Cases A and B, show clearly that it is better to tailor the choice of vertical and horizontal scaling strategy to the workload rather than always depending on one strategy or the other.

In the light-workload case (A), Figure 2a shows that both strategies gave increased throughput, but vertical scaling responded more quickly and reached a higher steady-state throughput (about 10% higher). The more rapid response is due to a faster ramp-up of capacity, and the higher steady-state capacity is due to the well-known inefficiency of multiple servers sharing a light or moderate load, compared to a more powerful single server [19]. This inefficiency is captured in the queueing model used by ATOM for decision-making.

In the heavy-workload case (B), Figure 2b shows that vertical scaling is much less effective at increasing steady-state capacity, about 20% less. The two strategies respond about

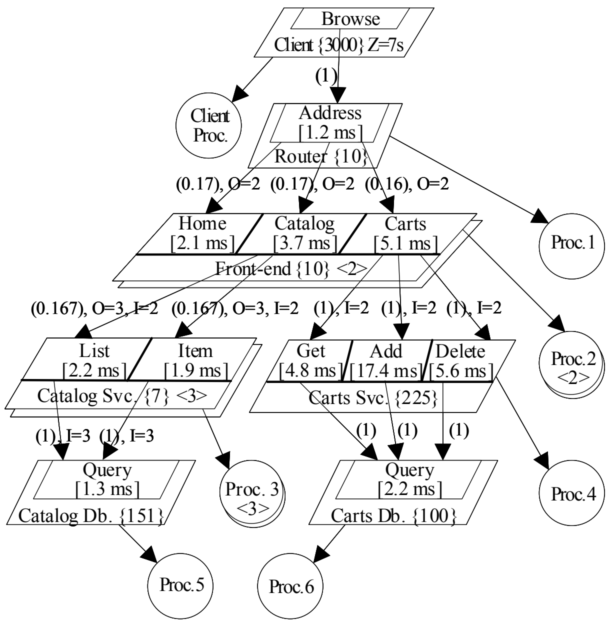


Fig. 3. An LQN model of the Sock Shop application. The boxes, nested boxes and circles represent tasks, entries and processors respectively. Calls are shown by arrows, which are associated with a call mean, fan-in and fan-out. The values between square, curly and angle brackets specify service demands, multiplicity and number of replicas respectively.

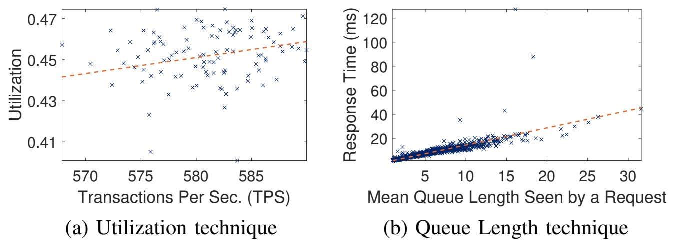


Fig. 4. Service demand estimation using two different techniques

equally quickly in heavy load, so horizontal scaling is overall much more effective. Vertical scaling gave lower throughput in heavy traffic because the front-end microservice code is not internally parallel, thus it cannot leverage the availability of the extra CPU cores. This is not uncommon in software, as demonstrated by the Sock Shop implementation itself.

Summarizing, our results suggest that, when a service can both be vertically or horizontally scaled, an optimal runtime autoscaler would reason on which of the two scaling actions to take based on workload, individual microservice features, and overall software system architecture. In the next sections, we adopt a queueing-based approach to address this problem.

III. Performance Model

This section illustrates the use of Layered Queueing Network (LQNs) for performance modeling of microservices, together with service demand estimation methods required to instantiate such models in concrete systems.

A. *LQN Modeling*

LQN models can naturally abstract scenarios where a microservice can sometimes also work as a client to another microservice. This property often leads to the use of LQN in modeling distributed systems [20]–[22] and can motivate the adoption of these models in microservice autoscalers. For ease of illustration, the LQN model we have derived for the running case is shown in Figure 3. In the LQN model, microservices are abstracted by tasks[3]. The reference task (Client) represents the system workload. The entries of the task, which are equivalent to service classes in a queueing network, hosts the features or APIs exposed by the microservices to their clients (entries). The tasks are placed in a processor, which in this case represents the CPUs in the host servers.

In a LQN model, the performance impact of vertical or horizontal scaling can then be assessed either by scaling the service demands of the activities in a processor according to the CPU share (vertical scaling) or, for horizontal scaling, by increasing the level of replication of a task. This inherently increases the number of service queues for that task. An autoscaler can then reason on expected performance by analyzing the LQN with standard solvers [23], [24] and obtaining corresponding metrics such as throughput, utilization and response times expected for each microservice after performing the scaling action.

B. *Service Demand Estimation*

The tasks of an LQN model can include one or more activities. An activity consumes a CPU time which is represented by its service demand, which needs to be estimated to parameterize an LQN. A common method is to use a linear regression model based on the utilization law $U_i = \sum_k X_k D_{i,k}$, where U_i means the utilization of resource i, X_k means the throughput of class k and $D_{i,k}$ means the service demand of a job of class k in resource i [25]. To get the service demand, initially some utilization and throughput samples are collected from system measurements. The utilization function is then fitted by ordinary Least Square (LSQ) regression, possibly with non-negativity constraints, to obtain the service demands.

The regression method assumes some variability in the observed throughputs, which is not always present in observations on microservices. To illustrate this, in Figure 4a, we have plotted the estimation data of *View item* feature of the Sock Shop application. The data do not show strong correlation between the variables, which in turn means that accurate estimates of demand cannot be found this way. The latter issues can occur frequently for microservices as they are finely grained and contains a simple business logic. This makes microservices less resource intensive than traditional applications. Thus estimating their service demands based on utilization becomes more difficult.

More accurate estimates can be found using more fine-grained observations on the system, of the response time of individual operations or groups of operations versus as a func-

[3]LQN Notation and Solvers [www.sce.carleton.ca/rads/lqns]

TABLE II
WORKLOAD PATTERNS FOR MODEL VALIDATION

#	Request Distribution			Concurrent Users	Think Time
	Home	**Catalog**	**Carts**		
1	57%	29%	14%	1000, 2000, 3000	7 sec
2	34%	33%	33%	1000, 2000, 3000	
3	57%	29%	14%	1500, 2500, 4000	10 sec
4	34%	33%	33%	1000, 2000, 3000	

TABLE III
PERCENT ERROR BETWEEN MODEL AND SYSTEM MEASUREMENT

Service Name	TPS % Error			Util. % Error		
	min	**max**	**avg**	**min**	**max**	**avg**
Front-End	0.05	7.68	2.26	1.06	9.98	5.05
Carts Svc.	0.04	6.08	2.17	0.37	4.08	1.89
Catalog Svc.	0.09	7.83	2.79	0.66	4.01	2.04
Catalog Db.	0.43	7.75	2.81	0.05	1.7	0.95
Carts Db.	0.06	4.29	1.71	0.58	7.45	3.5

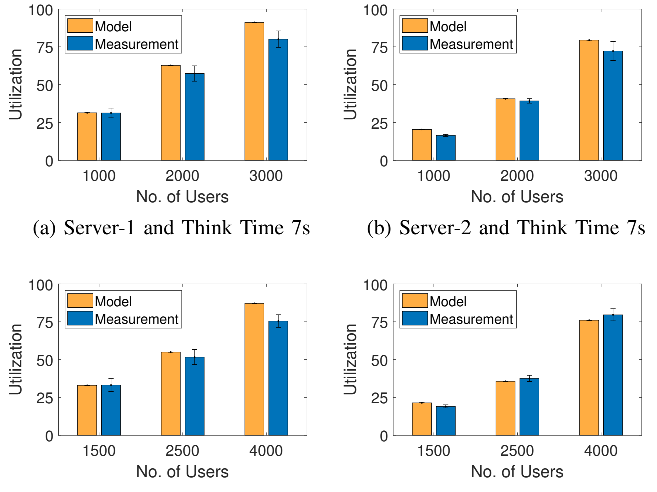
tion of the queue length at their arrival [26]. This technique is based on the formula for estimating response time using the mean-value analysis arrival theorem [26]. When applied to the previous case, the results are shown in Figure 4b. From the figure, it is seen that the variability in data is significantly higher than the data for utilization samples, simplifying the estimate of demands. It is also evident that the estimate will be insensitive to anomalies in the data. This suggests that the technique is applicable for microservices.

C. Validation

To validate the LQN model output against real system measurements, we have performed multiple experiments. We have considered a subset of the Sock Shop application from Figure 1 that contains business logic. The workloads for the experiments are presented in Table II. The request mix and the number of concurrent users have been selected such that they create light, normal and heavy load in the system. The think times are set as suggested in literature for such studies [27], [28]. For workload 2 and 4, we have deployed the microservices in Docker compose mode, creating a scenario with a single host server. For workload 1 and 3 we have deployed the microservices in two Docker swarm nodes, creating a scenario with multiple host servers, where server 1 hosts the front-end and cart services and server 2 hosts the other three services. Within these servers, to avoid the approximation error of multiserver queueing nodes, we have kept a single CPU online. For workload generation, we have used the Locust[4] tool with a distributed load testing configuration. The models are solved using the LQN simulator (LQSIM).

The percent errors, considering the TPS and utilization, between the model and system measurement are presented in Table III. We presented the minimum, maximum and average errors for both TPS and utilization. From the table, it is seen that all the average errors are less than 5.05%, and even the maximum error, observed in the utilization of front-end microservice, is only 9.98%. These errors are well accepted for performance modeling [21], [29]. To further investigate the model accuracy, we have considered the total CPU utilization of the two servers participating in Docker swarm. From the results, as presented in Figure 5, it is seen that, for all the workloads, the model estimations are close to the system measurement. To further demonstrate accuracy of individual service, we have used a heavy workload (workload 1 in Table II) with 3000 users. From the results, as presented in Table IV,

[4]Locust - A modern load testing framework [www.locust.io]



(a) Server-1 and Think Time 7s (b) Server-2 and Think Time 7s
(c) Server-1 and Think Time 10s (d) Server-2 and Think Time 10s

Fig. 5. Utilization of different servers according to model estimation and system measurement for workload pattern 1 and 3

it can be seen that for all of the components, the throughput and utilization values are within the acceptable range of 5% to 10% [21], [29]. These results indicate that the model can be used for further performance analysis.

IV. ATOM: AUTOSCALING MICROSERVICES

In this section we discuss the details of our autoscaler ATOM. Initially, we present how a user can use ATOM, the system considered by ATOM and its components. Subsequently, we formulate the problem that is addressed in ATOM. Finally, we present the optimization method incorporated in ATOM to find solutions for the problem and suggest optimal scaling configurations.

A. Context and Architecture

To use ATOM, the user needs a deployed microservice application and its LQN model. The LQN model can be obtained in two ways. If the UML design of the application is available, it can be used to automatically generate a model following a transformation technique similar to [30]. This is a likely scenario for an in-house application, deployed in a private cloud. If the application is deployed in a public cloud, the service provider may not have access to a model-based specification. However, due to the fine-grained nature of microservices, a suitable model may be developed in principle

TABLE IV
FEATURE AND MICROSERVICE SPECIFIC TPS AND UTILIZATION FROM MODEL AND MEASUREMENT FOR WORKLOAD 1 WITH 3000 USER

Service Name	Feature Name	Model TPS	Measured TPS	% Error	Model Util.	Measured Util.	% Error
Front-end	Home	236.3	221.3	6.3	75.2	65.9	9.3
	Catalog	120.2	110.9	7.7			
	Cart	58	55.6	4.1			
Carts Svc.	Get	19.1	18.5	3.1	16	14.2	1.8
	Add	19.1	18.5	3.1			
	Delete	19.7	18.5	6.1			
Catalog Svc.	List	60.2	55.5	7.8	19.2	15.4	3.8
	Item	60.1	55.5	7.7			
Catalog Db.	Query	120.2	110.9	7.7	12	12.6	0.6
Cart Db.	Query	58.1	55.6	4.3	48.2	44.3	3.9

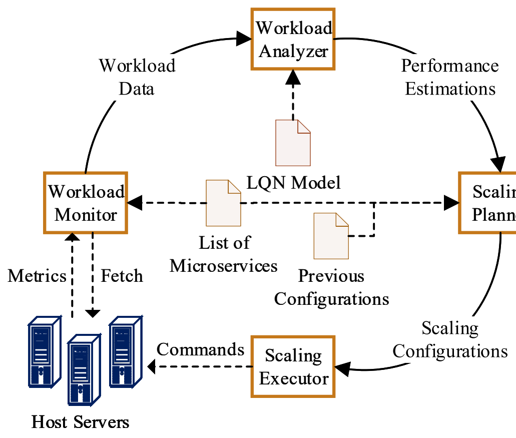


Fig. 6. Overview of ATOM's components and their interactions

by only monitoring the communication among the microservices. As mentioned earlier, service demands are important parameters for LQN models. In this work, we have assumed that service demands are estimated offline, we plan in future work to add an online demand estimation method.

ATOM considers a system which executes different transactions using microservices. Each transaction type combines a set of features of the microservices and yields different revenue. Thus, the transactions can be prioritized based on their revenue. ATOM acts periodically. At a moment when it is invoked, suppose the system has previously been optimized but has experienced an increase or other change in the workload. ATOM aims to estimate a scaling strategy for each of the microservices so that the revenue of the system is still optimal.

The components of ATOM, as shown in Figure 6, are aligned with the MAPE-K (monitor, analyze, plan and execute with a shared knowledge base) [31] control model. The workload monitor counts the user requests for each feature of the system, using a set of time intervals within a monitoring window [32] to obtain a set of samples for each feature. These samples are forwarded to the workload analyzer after a monitoring window is completed. Based on the monitoring data, the workload analyzer needs to update two things in the LQN model: the concurrent number of users in the system (N) and the request mix. Since we are using a workload generator, the system that we consider is a closed loop system. Thus, we assume that we know the value of N. To calculate the request mix, the workload analyzer initially determines the average request count for each type of feature in a monitoring window. From these average request counts, the request mix is determined as the fraction of requests for different system features. The analyzer then updates the LQN model with the value of N and new request mix. To update the model, the value of N is set as the multiplicity of client task and the request mix is used to update the call parameters among the entries of the model.

The analyzer uses a search strategy, described in Section IV-B and IV-C, to vary the number of replicas (representing horizontal scaling) and the host demands (representing vertical scaling) and find the best combination. This result is forwarded to the scaling planner. Initially, the planner translates the received result from modeling to system perspective using a map between the LQN model and the microservices. After that, the planner decides the scaling strategies (number of replicas and CPU share of each replica) for each microservices. The planner also includes an option to use the previous scaling configurations as a reference to prevent a drastic change in the current monitoring window. This can be achieved either from the perspective of TPS or total allocated CPU capacity. If the option is chosen, the planner does not change a configuration in the current monitoring window unless the TPS improves or total CPU share does not change more than a threshold value. After that, the planner creates a scaling script for the scaling command executor. Finally, the executor component runs the scaling commands specific to each microservice. All these components share a knowledge base that includes the LQN model, list of microservices and their mapping to LQN and scaling configurations from the previous iteration.

B. Problem Statement

ATOM provides the scaling decisions by solving multiple LQN models. To generate these models, ATOM needs to decide the number of replicas for each microservice and the CPU shares for all the replicas of each microservice. Considering there are N microservices, these decisions are represented by the following variables.

- $r^{(t)} = [r_i^{(t)}], r_i^{(t)} = 1, \dots, Q_i, \forall i = 1, \dots, N$. Replica vector: denotes the number of replicas for each of the microservices i for the time interval t, where Q_i is the upper bound of replicas that a microservice can have.
- $s^{(t)} = [s_i^{(t)}], s_i^{lb} \leq s_i^{(t)} \leq s_i^{ub}, \forall i = 1, \dots, N$. CPU share vector: denotes the CPU share for each of the replicas of a microservice i for the time interval t. The share is bounded by upper and lower bound s_i^{ub} and s_i^{lb}.

Our objective is to determine the values of $r^{(t)}$ and $s^{(t)}$ that maximize the revenue $B^{(t)}$ and minimize total allocated CPU capacity $C^{(t)}$ for the next time interval t. The revenue is defined as the number of transactions completed per unit time. The transactions are weighted by a numerical coefficient expressing their business value. These weights vary according to the priority of a feature of the application. The features are associated with the service classes of different microservices. We consider that a microservice i has E_i number of service classes. The number of transactions per unit time for a given period t and a particular service class j of microservice i is denoted by $X_{ij}^{(t)}$. The weight of the transactions is denoted by ψ_{ij}. In calculating the revenue, it is common to ignore the transactions of a particular set of services. Thus we define a set of revenue calculation services $I = \{i \in \mathbb{N} | i \in \{1...N\} \wedge \phi(i) \neq 0\}$, where $\phi(i) = 0$ if a microservice i is ignored. We also define the set of replication values and CPU share vector as $R = \{r \in \mathbb{N}^N | \forall i = 1 \dots N, r_i = 1 \dots Q_i\}$ and $S = \{s \in \mathbb{R}^N | \forall i = 1, \dots, N, s_i^{lb} \leq s_i \leq s_i^{ub}\}$, Considering these variables, we can calculate $B^{(t)}$ according to (1).

$$B^{(t)} = \sum_{i \in I} \sum_{j=1}^{E_i} \psi_{ij} X_{ij}^{(t)} \tag{1}$$

The total CPU share $C^{(t)}$ can be defined simply as the sum of CPU shares of all the replicas of the microservices. This can be calculated as $C^{(t)} = \sum_i r_i s_i$. With these definitions of $B^{(t)}$ and $C^{(t)}$, we express the objective function for our optimization problem as a weighted sum in (2).

$$\Theta^{(t)} = \max_{(r^{(t)}, s^{(t)})} \tau_1 \hat{B}^{(t)} - \tau_2 \hat{C}^{(t)} \tag{2}$$

$$\text{s.t:} \quad W_{ij}^{(t)}(r^{(t)}, s^{(t)}) \leq W_{ij}^{\max}, \forall i \in I, j = 1, \dots, E_i \tag{3}$$

$$C_k^{(t)}(r^{(t)}, s^{(t)}) \leq C_k^{\max}, \forall k \tag{4}$$

$$U_i^{(t)}(r^{(t)}, s^{(t)}) \leq U_i^{\max}, \forall i \tag{5}$$

$$r^{(t)} \in R, s^{(t)} \in S$$

In (2), $\hat{B}^{(t)}$ and $\hat{C}^{(t)}$ represents the normalized value of $B^{(t)}$ and $C^{(t)}$. These are normalized since it is suggested for a weighted sum method if the magnitudes of the objectives differ significantly [18], which is our case. The relative importance of revenue and total CPU share are defined by the weights τ_1 and τ_2. The constraints for the optimization problem are presented in (3)-(5). The function $W_{ij}^{(t)}$, in (3), estimates the response time for a service class j of microservice i for a time period t. The estimated response time cannot be higher than the maximum limit $W_{ij}^{\max}$ mentioned in the SLA. The function $C_k^{(t)}$, in (4), calculates the total allocated CPU share of a server k for a time period t. This value must not exceed the server's capacity $C_k^{\max}$. In (5), $U_i^{(t)}$, represents the utilization value of a microservice i for a time period t, which must not exceed its limit $U_i^{\max}$.

C. Optimization Method

To find the optimal scaling strategy, ATOM needs to generate a set of solution candidates and evaluate their quality. The steps for solution candidate generation is presented in Algorithm 1. The algorithm requires three parameters: the LQN model, time limit for algorithm execution and the constraint tolerance value.

Initially, the algorithm generates a random set of configurations ($r \in R$,$s \in S$). This initial set should satisfy the constraints in (4). For a given r and s, the total CPU share C_k allocated in a server k can be calculated as $\sum_i r_i s_i z_{ik}, \forall i$ where $z_{ik} \in \{0, 1\}$ represents whether the replicas of a microservice i is placed in server k. After that, a time bounded searching starts to find an optimal solution. This bound should be less than the monitoring window to execute a scaling strategy before receiving the data of the next monitoring window. Thus, for a 5 minute monitoring window, we have used a 2 minute time bound.

Algorithm 1 Solution Candidate Generation

```
Input: LQN Model $\Gamma$, $timeLimit$ and $tolerance$
Output: Set of solution candidates $G$
1:  Init: generate initial $config$ set
2:  while $time \leq timeLimit$ do
3:      $currentCandidates \leftarrow \varnothing$
4:      for each $(r, s) \in config$ do
5:          $\Gamma' \leftarrow updateReplication(r, \Gamma)$
6:          $\Gamma'' \leftarrow updateCalls(r, \Gamma')$
7:          $\Gamma''' \leftarrow updateHostDemand(s, \Gamma'')$
8:          $(fval, c) \leftarrow solveModel(\Gamma''')$
9:          if $c \leq tolerance$ then
10:             $currentCandidates \cup \{(r, s), fval\}$
11:         end if
12:     end for
13:     $config \leftarrow generateConfig(currentCandidates)$
14:     $G \cup currentCandidates$
15: end while
```

The search process is as follows. For each r and s pair, the LQN model is updated. This update process has three steps. At first, the $updateReplication$ function updates the replica number of a task-processor pair according to the replicas of the corresponding microservice. After that the fan-in and fan-out values of the tasks and the calls parameter among the entries is updated by the $updateCalls$ function. Considering a chain of tasks $A \rightarrow B \rightarrow C$, the fan-in of task B is $F_{in}^B = r_A$ and the fan-out is $F_{out}^B = r_C$, where r_B and r_C is the number of replicas of task B and C. According to the current request mix, if the value of a call parameter from B to C is $D_{B,C}$, it is updated as $\frac{D_{B,C}}{r_C}$. Finally, the $updateHostDemand$ function scales the host demands of the activities of a task by dividing the host demands by the CPU share of the corresponding microservice for that task.

After completing the update process, the model is solved. Since we need to solve a large number of models at runtime, we need to get the solutions quickly. Thus, we have invoked an analytic solver LQNS [23], with option for Bard-Schweitzer single step mean value analysis (MVA) [33], allowing faster model solving. We have checked the solution provided by LQNS to verify that the estimated r and s satisfy the response time and utilization constraint in (3) and (5) considering the given $tolerance$. The response time W_{ij} is obtained from the residence time for the corresponding call to the entry (representing a service class j) of a task (representing a microservice i). The residence time is defined as the sum of queueing and service time for a call to an entry. In LQN context, the service time is an output that represents the time a server remains busy to serve a request, including the nested service calls. The utilization value U_i^t for a microservice i is estimated from the utilization value of its corresponding processor in the model. If the constraints are satisfied, the throughput values of the tasks, corresponding to the features of the microservices, and the configuration (r and s) is added to the set of current candidate solutions. This set is used to generate a set of new configuration for the next iteration.

Since it is infeasible to evaluate all the scaling configurations at runtime, we will end up with a subset of solution candidates. In such cases, a meta-heuristic approach is commonly used to generate a sufficiently good set of solution candidates. Here, we have used the genetic algorithm (GA) for this purpose. GA can provide optimal solutions for highly non-linear problems [34] and is a powerful approach for problems with integer variables [35]. This is important in our case as we have a non-linear mixed-integer program. The integer variables in our case is the number of replicas. The non-linearity originates from the reason that simply increasing the number of replicas and CPU share may not provide an optimal scaling strategy. It is important that how these numbers are increased depending on the characteristics of the current workload. In ATOM, we used the MATLAB implementation of genetic algorithm[5]. At every iteration, we have used GA to generate a set of scaling configurations, to be evaluated in the next iteration, based on the current set of solution candidates. After this generation, the current candidate solutions are merged with the set of all solution candidates. The algorithm continues until it reaches the time limit. Finally, these solution candidates are forwarded to the scaling planner.

Since we are running GA at runtime with time constraints, it is possible that the scaling strategy provided by it can be further improved. This can be both from the perspective of revenue maximization and CPU share minimization. To address this, the planner uses two quick fixes. Firstly, to minimize CPU shares, it checks whether a microservice was allocated less CPU share in the previous monitoring window. If so, it replaces the current configuration for that microservice with the previous one. If this change does not affect the TPS significantly the previous configuration is chosen for that microservice, otherwise the current configuration is kept. Secondly, to increase the TPS, ATOM reduces the number of replicas while increasing the CPU share of each replica, keeping the total CPU share same. It then checks again whether the TPS is affected significantly and if not, it keeps the modified configuration. This improves the TPS since reducing the number of replicas also reduces the parallelization overhead. Before sending the scaling strategy, the planner also checks for a drastic change in total CPU share, as mentioned in section IV-A, if that option is chosen. After that, the planner creates the scaling configurations from the best solution candidate, which are executed by the scaling executor.

[5]Genetic Algorithm - MATLAB & Simulink - [www.mathworks.com/discovery/genetic-algorithm.html]

TABLE V
DOCKER SWARM SERVER CONFIG. FOR PERFORMANCE EVALUATION

#	Deployed Microservices	Online CPU	CPU Freq.	Memory
1	Router, Front-end, Carts Db.	4	1.2 GHz.	64 GB
2	Catalog Svc., Carts Svc., Catalog Db.		0.8 GHz.	16 GB

TABLE VI
WORKLOADS CONSIDERED IN EVALUATING THE AUTOSCALERS

Name	Request Distribution			Concurrent Users	Think Time
	Home	Catalog	Carts		
Browsing Mix	63%	32%	5%	1000, 2000, 3000	7 sec
Shopping Mix	54%	26%	20%	1000, 2000, 3000	
Ordering Mix	33%	17%	50%	1000, 2000, 3000	

V. EVALUATION

In this section we present the evaluation of ATOM. We initially provide the details of our experimental setup and the baseline autoscalers that we have developed to compare with ATOM. After that, the performance comparison is presented according to different elasticity and performance metrics.

A. Experimental Setup

For ATOM's performance evaluation, we have deployed the Sock Shop application in Docker swarm mode in two servers. The details of the servers are provided in Table V. We reduced the CPU frequency in those servers to create high resource contention. The CPU frequency that has been presented in Table V is the reduced CPU frequency. We have developed two rule based autoscalers to compare their performance with ATOM. Such autoscalers are common in industry [5] for example Amazon Web Services (AWS)[6] and Kubernetes[7].

Both autoscalers monitor the CPU utilization of the microservices. If the CPU utilization level reaches the current limit, the autoscalers double the amount of CPU share currently allocated to the microservices. The first scaler (UH) allocates the resource horizontally and the second one (UV) vertically. For example, if the CPU utilization reaches 35%, a value near to the limit of 40%, UH creates 2 replicas each having 0.4 CPU share and UV increases the CPU share of the replica to 0.8. UH operates only on the stateless microservices whereas UV operates on both stateful and stateless microservices. Thus, for UH scenarios we have allocated a full CPU core for each of the stateful microservices.

Here we have configured the Sock Shop application to handle 500 concurrent users in a browsing workload. After this initial setting, we have evaluated UH, UV and ATOM considering an increase in workload in multiple combinations of request mix and concurrent number of users (N). The request mix is similar to the suggested mix in TPC-W benchmark [27]. The values of N are chosen to create scenarios ranging

[6]Amazon EC2 Auto Scaling - [aws.amazon.com/ec2/autoscaling]

[7]Horizontal Pod Autoscaler - [kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale]

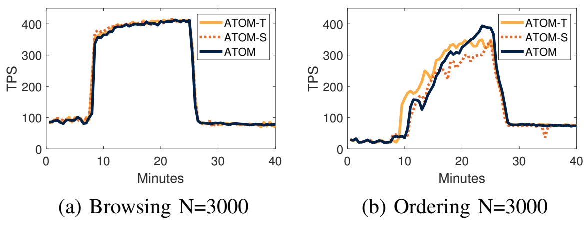


Fig. 7. Performance comparison of ATOM with its conservative versions: ATOM-S and ATOM-T

from moderate to heavy increase in the current workload. The details of the workload is provided in Table VI.

Before comparing ATOM with the baseline scalers, we have compared different variations of ATOM. The variations are based on ATOM's likelihood to change the CPU shares at each monitoring window. As mentioned in section IV-A, this has been controlled by the planner by being conservative about changing the system configuration on each monitoring window. It provides two options for achieving this: not allowing a large change in CPU share or discarding the current configuration if the potential improvement in TPS is not significant. We call these versions a conservative version of ATOM. The conservative version that changes CPU shares by looking potential improvement in TPS is named ATOM-T and the one that allows only a small change to the total allocated CPU capacity is named ATOM-S.

For performance comparisons, we have chosen two workloads from the light browsing mix and heavy ordering mix with $N = 3000$. It is seen from Figure 7 that all the versions of ATOM produce nearly identical improvement in TPS for the browsing workload. In case of ordering mix, ATOM-T yields slightly higher average TPS whereas ATOM-S yields slightly less TPS than ATOM. Although the TPS for ATOM increases slowly at the beginning, later it increases sharply and achieves a higher value than ATOM-T. For ATOM, such sharp increases can occur, possibly with a sharp and short decrease period before, whereas for ATOM-T and ATOM-S the improvement is more steady. An issue for the conservative versions is that it can be difficult to determine the threshold value that gradually improves the scaling strategy rather than completely stopping the improvement. This is bad if the system experiences the increase in workload for a long period. Thus, for the latter experiments we have used ATOM.

B. Results

For all the workloads in Table VI, we have run the experiments for 40 minutes where the first 25 minutes represent the increase in workload. The effect on the TPS by ATOM and the baseline autoscalers is presented in Figure 8. From the figure, it is seen that there is a delay in the scaling action for ATOM. In these experiments, the monitoring window has been set to 5 minutes for all the scalers. In addition to this delay, ATOM has on average a 2.5 minute delay for its optimization and planning, whereas the other autoscalers initiate scaling

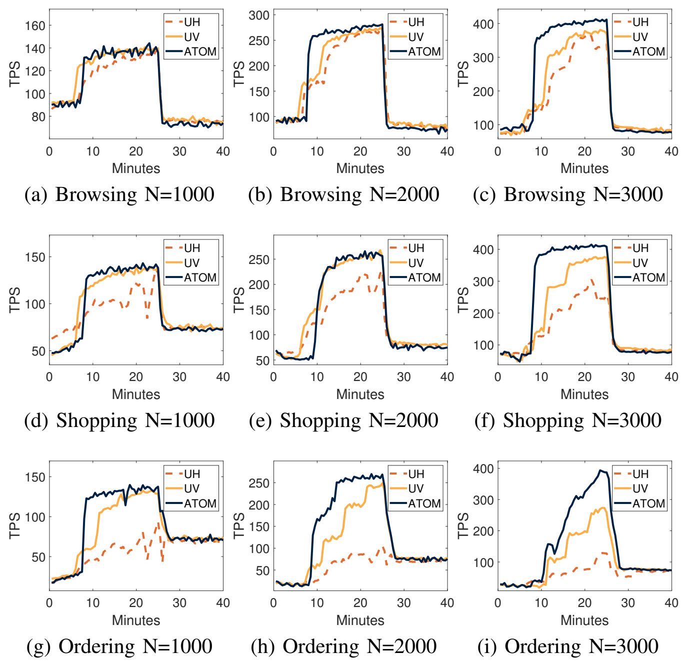
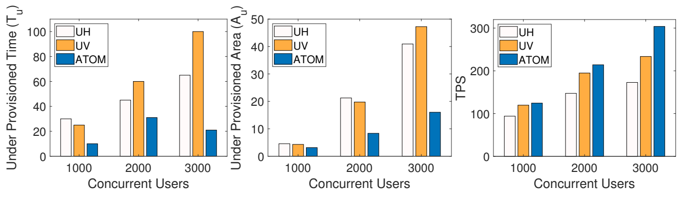


Fig. 8. The effect over time on the TPS by ATOM and baseline autoscalers under different workload

actions immediately after the monitoring window. However, from the figure it is seen that the delay does not effect the performance of ATOM significantly compare to others. This is because, although there is a step by step improvement in TPS for the baseline autoscalers, the TPS improves significantly in case of ATOM in the first step. This is expected as ATOM monitors the workload and then provide an optimal scaling configuration for that workload. On the other hand, the baseline scalers monitor the current utilization and cannot estimate the maximum amount of required CPU share from this information only. Thus, they improve the TPS in a step by step basis.

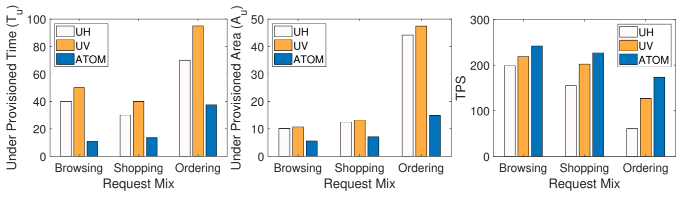
For a quantitative evaluation, we have used the following metrics: total under-provisioned time (T_u) [36], [37], total under-provisioned area (A_u) [36], [37] and TPS. The metrics indicate how an autoscaler performs when there is a increase in the workload. In the context of microservices, T_u can be defined as $\sum_i T_u^{(i)}$, where $T_u^{(i)}$ is the total time for a microservice i in an under-provisioned state. Similarly, A_u can be defined as $\sum_i A_u^{(i)}$ where $A_u^{(i)}$ is the total under-provisioned area for an microservice i. The under-provisioned area represents the extent of under provisioning for a time period t. It is defined as the product of an under-provisioned period and the difference of required and allocated CPU capacity for a microservice in that period. TPS is defined as the total number of transactions completed per second. We have considered the TPS during the increased load period since the TPS is affected in that period and the autoscalers aim to resolve this issue. For T_u and A_u a lower value is expected, which is the opposite case for TPS.

In Figure 9 we compared the performance of the scalers,



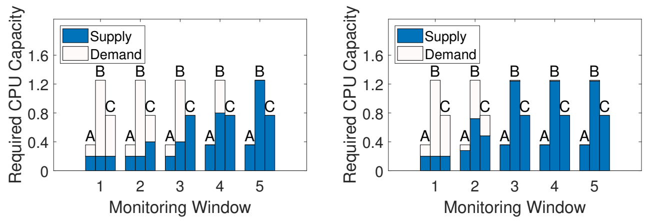
(a) T_u - lower is better (b) A_u - lower is better (c) TPS - higher is better

Fig. 9. Comparison of the autoscalers for different elasticity and performance metrics with the increase of concurrent users



(a) T_u - lower is better (b) A_u - lower is better (c) TPS - higher is better

Fig. 10. Comparison of the autoscalers for different elasticity and performance metrics with change in request mix



(a) UV (b) ATOM

Fig. 11. The demand vs. supply in CPU capacity for 3 microservices, involved in a layered bottleneck, after each scaling action by UV and ATOM. The microservices are: A - Router, B - Front-end and C - Cart Svc.

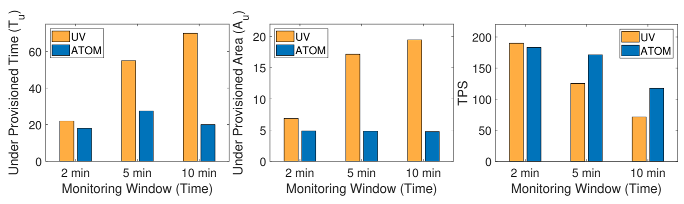
according to the considered metrics, with increase in concurrent number of users (N). The results are considering 3 microservices since UH does not scale the router and 2 database services. The results show that ATOM outperforms UH and UV considering all the metrics, particularly for higher values of N. It is seen that with the increase of N, T_u and A_u increases for all the scalers though lower is better in this case. However, this increase is expected as with the increase in N, the initial gap between required and allocated CPU capacity also increases. Among these values, the values for ATOM is the lowest because it can reduce the CPU capacity gap significantly after the first monitoring window. This helps ATOM in improving the TPS than the other scalers. Considering $N = 3000$, the improvement in TPS is 30% higher for ATOM than the next best approach UV.

We also present the results from the perspective of change in request mix in Figure 10. In the figure, we can see that, in terms of TPS, ATOM by far outperforms both UH and UV in ordering mix, while performing similar in browsing and shopping mix. For the ordering mix, the TPS is 37% higher than UV, which is the second best approach. From Figure 10a, we see that ATOM has a very low T_u comparing to UH and UV in all the mixes. However, as seen in Figure 10b, since the difference in A_u is not as high as it is in T_u for browsing and shopping mix, the TPS is nearly similar for all the scalers in browsing and shopping mix.

Besides being able to reason on horizontal and vertical scaling choices, another reason for ATOM's better performance is that it addresses layered bottlenecks [38], [39] better than UH and UV. A layered bottleneck scenario can cause a delay in reducing the difference between required and allocated CPU capacity for a microservice. We present one such scenario in Figure 11 for the ordering mix with $N = 2000$, where a layered bottleneck has evolved among 3 microservices - router, front-end and cart service. For UV, from Figure 11a, it is seen that the router remains in starvation up to window 3 as the front-end does not receive an additional capacity until window 3. The front-end remains in starvation up to window 4 as the cart service's requirement is fulfilled in window 3 and from there the front-end's large capacity gap cannot be reduced solely in window 4. On the other hand, ATOM starts to remove all the bottlenecks at once in window 2 since it has obtained the whole picture of the system's capacity requirement after window 1. The bottlenecks are partially resolved in window 2 due to ATOM's delay in optimization, affecting the average CPU allocation in that window. Continuing from window 3, the bottlenecks are resolved by ATOM.

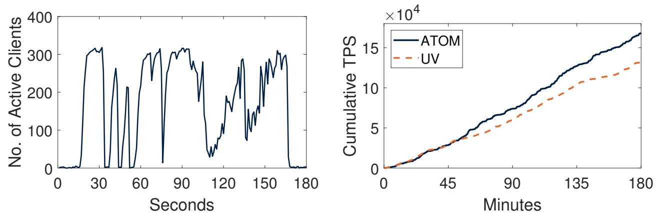
To observe the effect of a monitoring window over the scalers' performance, we have considered 3 different windows and evaluated them using an ordering mix with $N = 2000$. Since UH is outperformed by UV, in these experiments, we have only considered UV. From the results, as presented in Figure 12, it is seen that T_u and A_u remains almost same for ATOM in all cases. This is because those values for ATOM are dominated by its optimization time, which has been bounded by a fixed value of 2 minute in these experiments. The results show that ATOM outperforms UV for the 5 and 10 minute window. For the 2 minute window, they perform similarly. However, such performance for a short monitoring window is not guaranteed for two reasons. Firstly, for a large number of layered bottlenecks, which is likely for microservices, even with a short window that allows quick changes in system configuration, the time for resolving those bottlenecks can be significantly high. Secondly, such quick shifts in system configuration can affect the system steady state if new replicas are created or a microservice needs to be restarted. This case is not applicable to UV as it only scales vertically. However, vertical scaling is not an effective option in all scaling scenarios [6], which is also evident from our experiments.

It is important for an autoscaler to work with workload having burstiness. We have created such workloads by injecting burstiness in the ordering mix with $N = 500$ using the index of dispersion I [40]. We have used two values, $I = 400$ and $I = 4000$, representing moderate and high burstiness. We have evaluated the performance of UV and ATOM with these two workloads. From the results, we have not observed



(a) T_u - lower is better (b) A_u - lower is better (c) TPS - higher is better

Fig. 12. Performance comparison of UV and ATOM with change in monitoring window size



(a) Arriving Clients ($I = 4000$) (b) Cumulative TPS

Fig. 13. TPS yielded by the scalers with high burstiness in workload

any significant difference for moderate burstiness. However, for high burstiness, ATOM has outperformed UV. We have presented the results in Figure 13. It is seen that when using ATOM as an autoscaler, the traffic surges are also reflected in the system TPS, which is not exactly the case of UV. This also results in a better cumulative TPS of the system yielded by ATOM, which is 28% higher than UV. The reason for UV being outperformed is that since the CPU capacity is bounded, the utilization values do not reflect the traffic surges. In addition, the low load periods affect the average utilization. Thus, UV does not get an appropriate picture of the workload intensity and cannot scale accordingly.

VI. Related Work

Researchers have long been working with different autoscaling issues in the cloud [5]. They have suggested multiple approaches which are usually based on meta-heuristic algorithms [8], [41], application profiling [9], [42] or analytical modeling [10], [43]–[45]. The meta-heuristic approaches rely on methods like genetic algorithm [41] or ant colony optimization [8] to search a sample space of scaling configurations and suggest the optimal configuration. Application profiling approaches aim to establish a relation between QoS, workload intensity and the amount of required software and hardware resources. This relation is then used for dynamic resource provisioning.

The analytical methods focus on using a performance model to gain performance insights of the system. These performance insights are commonly leveraged to minimize a cost function for autoscaling [43], [44]. For performance model, abstractions like LQN [10], [43] or traditional queueing network is used [44], [45]. Such analytical methods mostly address the issue regarding the amount of required resource share and allocating that share either horizontally or vertically. However, recent researches [6], [7] have highlighted that it is important to make an assessment combining both horizontal and vertical scaling before deciding a scaling strategy.

The issue with these approaches are that their capacity estimation technique is not orchestrated for microservices. It has been already suggested that for autoscaling microservices, it is important to consider container level metrics [11] and message queue metrics [12]. This is particularly applicable to profiling and meta-heuristic based techniques as they provide poor estimates if appropriate metrics are not used. The analytical methods also need to incorporate the properties of microservices, like its fractional CPU share, and assess whether the common performance assumptions regarding vertical and horizontal scaling holds for them.

Recently, researchers have proposed different rule based autoscalers for microservices [12]–[15]. Most of these scalers [13]–[15] consider simple container level metrics, like CPU and memory usage, rather than exploring new dimensions with metrics like container start-up time [11]. The researchers in [12] focused on that aspect and used different conditions of the message queues as a measure of system state. Considering such metrics makes these works applicable to microservices. However, these scalers only consider horizontal scaling though vertical scaling can outperform horizontal scaling depending on the nature of the workload [6].

VII. Conclusion and Future Work

In this paper we have presented ATOM, a model-driven autoscaler tailored to microservices application. Leveraging the ease of scaling and replicating microservices in Docker-based deployments, we have first demonstrated that if a services can be both vertically and horizontally scaled the workload characteristics need to be taken into account to decide which one of the two scaling actions is better. We have then proposed an autoscaler that leverages layered queueing network models to scale application capacity in the presence of time-varying workloads, including those affected by burstiness. Results indicate that ATOM can significantly improve over scalers based on horizontal or vertical scaling actions only.

Possible lines of future research include online profiling of service demands, which are in the present work assuming to be statically profiled via testing, and an extension of the method to include microservice migration and support for cloud-based serverless functions.

Acknowledgment

A. Gias is a commonwealth scholar, funded by the UK government. The work of G. Casale is partially supported by RADON, funded by the European Union's Horizon 2020 research and innovation program under grant agreement No. 825040. The research of M. Woodside was supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada. We would like to thank Lulai Zhu for his feedback on the optimization program. The data referenced in this paper is available at https://doi.org/10.5281/zenodo.2640909, released under the CC BY 4.0 license.

References

[1] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[2] L. Zhu, L. Bass, and G. Champlin-Scharff, "DevOps and Its Practices," *IEEE Software*, vol. 33, no. 3, pp. 32–34, 2016.

[3] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in Practice, Part 1: Reality Check and Service Design," *IEEE Software*, vol. 34, no. 1, pp. 91–98, 2017.

[4] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers," in *Proc. of Int'l. Symposium on Network Computing and Applications*. IEEE, 2015, pp. 27–34.

[5] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey," *ACM Computing Surveys*, vol. 51, no. 4, p. 73, 2018.

[6] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Model-driven optimal resource scaling in cloud," *Software & Systems Modeling*, vol. 17, no. 2, pp. 509–526, 2018.

[7] E. Incerto, M. Tribastone, and C. Trubiani, "Combined Vertical and Horizontal Autoscaling Through Model Predictive Control," in *Proc. of European Conference on Parallel Processing*. Springer, 2018, pp. 147–159.

[8] T. Chen and R. Bahsoon, "Self-Adaptive Trade-off Decision Making for Autoscaling Cloud-Based Services," *IEEE Trans. on Services Computing*, vol. 10, no. 4, pp. 618–632, 2017.

[9] C. Qu, R. N. Calheiros, and R. Buyya, "A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances," *Journal of Network and Computer Applications*, vol. 65, pp. 167–180, 2016.

[10] C. Barna, M. Litoiu, M. Fokaefs, M. Shtern, and J. Wigglesworth, "Runtime Performance Management for Cloud Applications with Adaptive Controllers," in *Proc. of Int'l. Conference on Performance Engineering*. ACM, 2018, pp. 176–183.

[11] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance Engineering for Microservices: Research Challenges and Directions," in *Proc. of Int'l. Conference on Performance Engineering Companion*. ACM, 2017, pp. 223–226.

[12] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in cloudiot-environments," in *Proc. of Int'l. Conference on Performance Engineering*. ACM, 2018, pp. 157–167.

[13] L. Florio and E. Di Nitto, "Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures," in *Proc. of Int'l. Conference on Autonomic Computing*. IEEE, 2016, pp. 357–362.

[14] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert, "Self-managing cloud-native applications: Design, implementation, and experience," *Future Generation Computer Systems*, vol. 72, pp. 165–179, 2017.

[15] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and Monitoring as a Service," in *Proc. of Annual Int'l. Conference on Computer Science and Software Engineering*. IBM Corp., 2017, pp. 234–240.

[16] M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software," *IEEE Trans. on Computers*, vol. 44, no. 1, pp. 20–34, 1995.

[17] J. A. Rolia and K. C. Sevcik, "The Method of Layers," *IEEE Trans. on Software Engineering*, vol. 21, no. 8, pp. 689–700, 1995.

[18] R. T. Marler and J. S. Arora, "The weighted sum method for multi-objective optimization: new insights," *Structural and multidisciplinary optimization*, vol. 41, no. 6, pp. 853–862, 2010.

[19] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge Univ. Press, 2013.

[20] A. Ufimtsev and L. Murphy, "Performance Modeling of a JavaEE Component Application using Layered Queuing Networks: Revised Approach and a Case Study," in *Proc. of Conference on Specification and Verification of Component-based Systems*. ACM, 2006, pp. 11–18.

[22] M. Tribastone, P. Mayer, and M. Wirsing, "Performance Prediction of Service-Oriented Systems with Layered Queueing Networks," in *Proc. of Int'l. Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2010, pp. 51–65.

[21] Y. Shoaib and O. Das, "Web Application Performance Modeling Using Layered Queueing Networks," *Electronic Notes in Theoretical Computer Science*, vol. 275, pp. 123–142, 2011.

[23] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks," *IEEE Trans. on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2009.

[24] J. F. Pérez and G. Casale, "Line: Evaluating Software Applications in Unreliable Environments," *IEEE Trans. on Reliability*, vol. 66, no. 3, pp. 837–853, 2017.

[25] S. Spinner, G. Casale, F. Brosig, and S. Kounev, "Evaluating approaches to resource demand estimation," *Performance Evaluation*, vol. 92, pp. 51–71, 2015.

[26] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson, "Estimating Service Resource Consumption From Response Time Measurements," in *Proc. of Int'l. Conference on Performance Evaluation Methodologies and Tools*. Pisa, Italy: ICST, 2009, pp. 48:1–48:10.

[27] D. A. Menascé, "TPC-W: A benchmark for e-commerce," *IEEE Internet Computing*, no. 3, pp. 83–87, 2002.

[28] J. Rolia, G. Casale, D. Krishnamurthy, S. Dawson, and S. Kraft, "Predictive modelling of SAP ERP Applications: Challenges and Solutions," in *Proc. of Int'l. Conference on Performance Evaluation Methodologies and Tools*. ICST, 2009, pp. 9:1–9:9.

[29] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall Upper Saddle River, 1984, vol. 22.

[30] G. P. Gu and D. C. Petriu, "From UML to LQN by XML algebra-based model transformations," in *Proc. of Int'l. Workshop on Software and Performance*. ACM, 2005, pp. 99–110.

[31] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, no. 1, pp. 41–50, 2003.

[32] S. Meng and L. Liu, "Enhanced Monitoring-as-a-Service for Effective Cloud Management," *IEEE Trans. on Computers*, vol. 62, no. 9, pp. 1705–1720, 2013.

[33] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, 2006.

[34] T. M. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[35] T. Yokota, M. Gen, and Y.-X. Li, "Genetic algorithm for non-linear mixed integer programming problems and its applications," *Computers & Industrial Engineering*, vol. 30, no. 4, pp. 905–917, 1996.

[36] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *Proc. of Int'l. Conference on Autonomic Computing*. USENIX, 2013, pp. 23–27.

[37] S. Dipietro, R. Buyya, and G. Casale, "PAX: Partition-aware autoscaling for the Cassandra NoSQL database," in *Proc. of Network Operations and Management Symposium*. IEEE/IFIP, 2018, pp. 1–9.

[38] J. E. Neilson, C. M. Woodside, D. C. Petriu, and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendezvous Networks," *IEEE Trans. on Software Engineering*, vol. 21, no. 9, pp. 776–782, 1995.

[39] G. Franks, D. Petriu, M. Woodside, J. Xu, and P. Tregunno, "Layered Bottlenecks and Their Mitigation," in *Proc. of Int'l. Conference on the Quantitative Evaluation of Systems*. IEEE Computer Society, 2006, pp. 103–114.

[40] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting Realistic Burstiness to a Traditional Client-Server Benchmark," in *Proc. of Int'l. Conference on Autonomic Computing*. ACM, 2009, pp. 149–158.

[41] S. Frey, F. Fittkau, and W. Hasselbring, "Search-Based Genetic Optimization for Deployment and Reconfiguration of Software in the Cloud," in *Proc. of Int'l. Conference on Software Engineering*. IEEE, 2013, pp. 512–521.

[42] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers," *ACM Trans. on Computer Systems*, vol. 30, no. 4, p. 14, 2012.

[43] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai, "Performance Model Driven QoS Guarantees and Optimization in Clouds," in *Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing*. IEEE Computer Society, 2009, pp. 15–22.

[44] Z. Zhu, J. Bi, H. Yuan, and Y. Chen, "SLA Based Dynamic Virtualized Resources Provisioning for Shared Cloud Data Centers," in *Proc. of Int'l. Conference on Cloud Computing*. IEEE, 2011, pp. 630–637.

[45] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, Model-driven Autoscaling for Cloud Applications," in *Proc. of Int'l. Conference on Autonomic Computing*, 2014, pp. 57–64.