Received August 20, 2020, accepted September 26, 2020, date of publication October 2, 2020, date of current version October 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3028310

A Quantitative Approach for Estimating the Scaling Thresholds and Step Policies in a Distributed Microservice Architecture

CHAITANYA KRISHNA RUDRABHATLA, (Member, IEEE)
Sony Pictures Entertainment, Los Angeles, CA 90232, USA
e-mail: chaitanya2409@gmail.com

ABSTRACT Microservice architecture (MSA) has become a de facto standard for developing complex web applications lately. Horizontal scalability, domain isolation, agility and the provision to use heterogenous technologies are some of the key factors for the growing popularity of this architecture. To automatically cater to varying load patterns, quite a lot of advancements have been made in the field of cloud computing, containerization and orchestrating mechanisms which aid to perform the auto scaling of the microservices. However, setting up the scaling policies, optimal upper and lower thresholds is a daunting task for large applications. It generally involves some initial guess work followed by multiple rounds of tuning based on the real time load variations. This process causes situations where either the service becomes unavailable to the load when the thresholds are on the lower side, (or) underutilization of the compute resources when they are on higher side. This paper aims to find a quantitative way of determining the thresholds and step-up policies by deducing the mathematical formulas. To solve this formidable problem, we propose a model in which the total resource consumption of a container running in the peak load scenario can be calculated by – (1) first identifying the critical transactions and their maximum concurrency rates,(2) then calculating the resource consumption of such transactions in a controlled environment and (3) finally applying those values to the mathematical formulas based on Gaussian functions to calculate the total resource consumption for the peak load scenario. Using the total resource consumption value and considering the network and startup latencies, an optimal upper threshold value for step-up functions can be calculated. In this paper, we calculated the upper threshold values using the above-mentioned approach and verified using a research project that the calculated value is indeed the minimum number of containers to handle load.

INDEX TERMS Microservices, cloud computing, orchestration, containerization, auto scaling, thresholds, step policies, Gaussian distribution, horizontal scaling.

I. INTRODUCTION

In the last few years microservice architecture has garnered a huge popularity in the field of web development [3], [4] and software design. This can primarily be attributed to the plethora of benefits associated to the framework like horizontal scalability, high availability, ease of deployments, multi cloud load distribution capability, domain isolation, agility in build and deployment cycles, etc., to name a few. The rise of public cloud platforms, major advancements in the containerization [2] and their orchestrating mechanisms have greatly contributed to the auto scaling techniques needed for making the services resilient and highly available. Scaling is the ability to increase or decrease the compute capacity and resources in response to an event or an action which cause a fluctuation in the load. Performing such scaling manually all day long for a high traffic application is humanly impossible [1]. Auto scaling is a feature which is provided by the orchestrating software like Kubernetes, docker swarm etc., in conjunction with the public cloud platforms like AWS, Azure, GCP etc., This feature would automatically spin up more containers or roll back the same based on the load fluctuations all day long and also would optimally size them based on the demand, without any human intervention. For the compute resources to scale up or down quickly, it needed a slicker and light weight solution which can be started and stopped at a faster pace compared to the traditional virtual machines. This thought process has given rise to

The associate editor coordinating the review of this manuscript and approving it for publication was Sabah Mohammed.

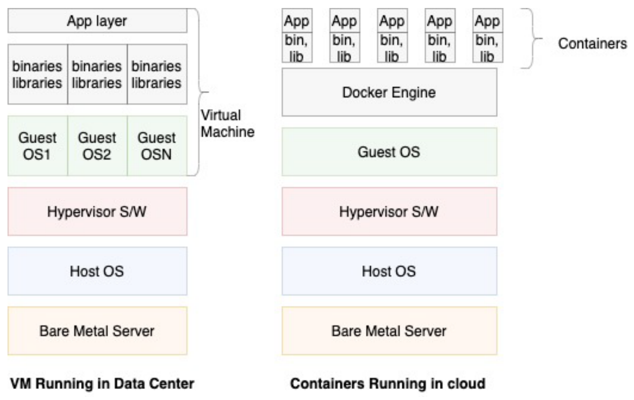

FIGURE 1. Composition of a Hypervisor based Virtual machine versus a lightweight container running on cloud.

the containers. Though the concept of containers was not new to the Unix world, it gained real significance only with the advent of public cloud and microservice architecture. As shown in Figure 1, a typical server running in the data center is made up of a host operating system which runs on the bare metal server. This in turn is segregated into multiple virtual machines (VM) using a software manager called Hypervisor. Each of these virtual machines have their own guest OS with their own heavy binaries and libraries. This heavy setup makes a VM slow to boot up. A container on the other hand runs on a virtual OS with the lightweight binaries and libraries. Due to this setup these containers can quickly be created, restarted or destroyed. This light weight and fast nature of containers gave rise to the modern horizontal scaling mechanisms which are offered in the public cloud infrastructures. Containerization can be considered as the first big step which paved the way towards auto scaling.

Auto scaling needs the management of compute resources which allows them to quickly scale up by spinning up more nodes or scale down by destroying the extra nodes based on the varying load and demand [5]. Containers proved to be the obvious choice for this. It is the need for management of containers which has given rise to orchestrating software like Kubernetes, Docker swarm, AWS ECS etc., to name a few. Once the initial configurations are set, these orchestrating technologies can manage the auto scaling of containers without the human intervention.

The rest of the paper is organized as follows. In Section II, we explain the related work. In Section III, we cover the background of autoscaling of services and explain how the load distribution thresholds can be calculated using mathematical equations. In Section IV, we go through the research project, discuss the work conducted in the project and outline the results. In Section V, the conclusions are presented.

II. RELATED WORK

Though the initial configurations for any popular orchestration mechanism are not too many, we observed that guessing the right values needed for an application is tough. This becomes even more challenging when the number of micro services grow in a large enterprise application. To set up the auto scaling thresholds, it normally involves initial guess work and couple of rounds of fine tuning based on the real time experiences in production environments. During this guessing process, it might lead to situations where either the auto-scaling thresholds are under configured there by leading to scenarios where the service becomes unavailable to the callers during a peak load situation (or) extra compute resources are provisioned which might go unused, due to the over configured auto-scaling threshold values, causing monetary losses [6], [7]. Some of the researchers like Muhammad Abdullah and others [24] have solved this problem for the micro services which are already functional in the production environments by using a resource prediction model which is trained based on the historical autoscaling performance traces. Few other researchers have used techniques like neural networks and machine learning for predicting workloads on existing services [25], [29]. Few researchers also investigated the impact on performance using the reactive autoscaling policies and configurations [26], [27]. Some researchers also used fog computing techniques based on to develop a stochastic performance model for capacity planning using Markovian chain model [28]. Few other researchers developed Configuration Recommender tools, based on Support Vector Regression (SVR) and Genetic Algorithms (GA) [30]. But there are not many algorithms out there to determine the new microservice scaling thresholds. The major motivation for this research is to to provide a mechanism to predict the optimal upper threshold value for the compute resources in an auto-scaling group for the micro services which are about to be provisioned.

We tried to meet the following objectives in this research work – (1) Elucidate how the auto scaling options, policies, and configurations work and present an overview of challenges involved in setting them up. (2) Present a model for calculating the peak load resource utilization using the Gaussian functions, by identifying the key transactions and their projected concurrency rates. (3) Provide a method to calculate the upper thresholds for a step-up function in the auto scaling group by considering the container start up time and network latencies involved. (4) Demonstrate the above steps by conducting an experiment using a custom micro service project setup in a public cloud platform. (5) And finally, cross verify that the step-up function with the calculated upper threshold handles the load when a peak load scenario is simulated.

III. MATERIALS AND METHOD

As discussed in earlier sections, auto scaling is used to adjust the compute resources automatically based on certain conditions or events or actions. In any of the public cloud environments or orchestrating mechanisms, there is an auto scaling group which launches or destroys the containers based on the events. Sections below describes some of the scaling options and their scaling policy types that can be configured.

A. SCALING OPTIONS

There are multiple options with which an auto scaling group can be scaled – (1) Manual scaling is the most basic way of scaling the compute resources. In this setup one needs to specify the number of containers needed for optimal performance and the auto scaling mechanism would scale to the number specified. But the disadvantage with this is that it constantly needs a human to monitor the system alerts and tune the scaling numbers. (2) One can also understand the load and demand patterns of the application services and predict the time and date when the containers need to be scaled up or down. But the disadvantage with this mechanism is that the load may have deviations from the standard patterns based on the amount of data being processed. This might cause service interruptions or system outages if not addressed in a timely fashion. (3) The auto scaling group can also be configured to maintain a fixed level of compute resources at all times. In this setup, auto scale groups can subscribe to the health events of the containers and respond by spinning more containers to meet the configured level, if some container becomes unhealthy and go down. This approach works for the applications with predictive and steady loads, but the disadvantage with this model is that it cannot meet the needs of a dynamic application loads. (4) The newer and advanced ways of auto scaling let the scaling process be controlled by parameters [8][9]. This dynamic scaling technique helps to auto scale in response to fluctuating conditions, particularly when one cannot predict when the load fluctuations can occur. This auto scaling is based on demand and is more advanced than the others.

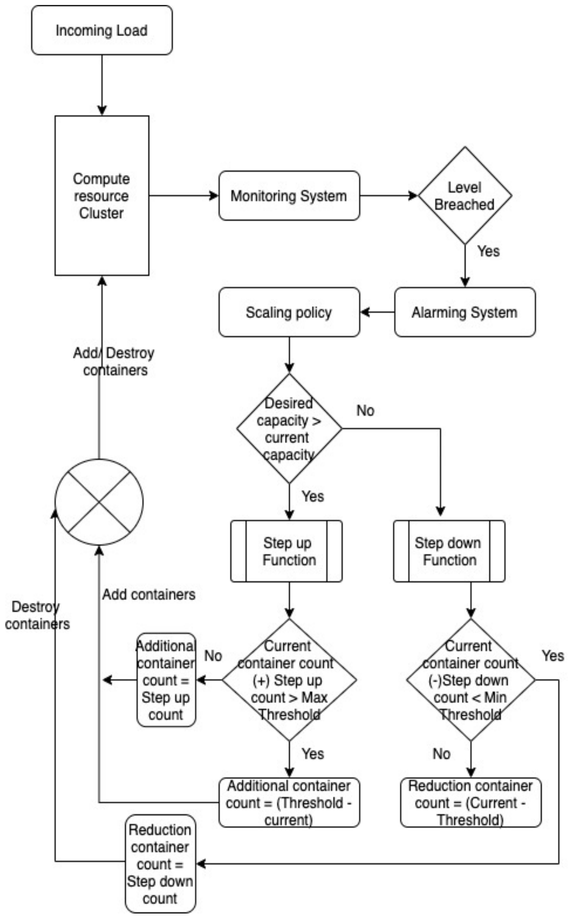
Incoming Load
Compute resource Cluster
Monitoring System
Level Breached
Yes
Scaling policy
Alarming System
Add/ Destroy containers
Desired capacity > current capacity
No
Yes
Step up Function
Step down Function
Add containers
Destroy containers
Additional container count = Step up count
No
Current container count (+) Step up count > Max Threshold
Current container count (-)Step down count < Min Threshold
Yes
Yes
No
Additional container count = (Threshold - current)
Reduction container count = (Current - Threshold)
Reduction container count = Step down count

FIGURE 2. **Flowchart for dynamic auto scaling policy action with step functions.**

B. SCALING POLICIES

Dynamic scaling option as mentioned above is one of the best suited techniques [10] for managing large application with high loads. For this kind of scaling there are a few kinds of scaling policies which can be configured as follows – (1) Simple scaling policy where in the capacity of the auto scaling group based on a single scaling adjustment. (2) Step scaling policy is the mechanism where the containers are adjusted in the form of steps based on the size of deviation of the load pattern. For an advanced configuration, it is possible to set up multiple step scaling policies to scale in or scale out. Due to the reasons mentioned in the above sections, the best combination which works for major large-scale applications is the dynamic auto scaling option with the step function policy. As shown in Figure 2 below, the dynamic auto scaling mechanism when configured with step functions goes through a series of step up and step-down functions before it gets auto tuned to the demand [11], [12]. This can be better explained with an example. Consider an auto scaling group with multiple nodes for a microservice named MSa. Let the nodes be called MSa1, MSa2 and so on. Suppose the maximum threshold of containers is set up as 4, minimum as 2, step scaling policy is set such that the step-up function adds 3 new nodes and the step-down function removes 2 existing nodes. Let the alarm system be configured in a way that it triggers the step-up function when the average CPU utilization of all the nodes is greater than 60% and triggers a step-down function when the CPU goes below 40%. Now consider the case when the 2 nodes MSa1 and MSa2 are at the CPU levels 80% and 60% respectively. This takes the average CPU to 70% which is greater than the configured policy of 60% for step up. In this case it triggers the step-up policy which tries to add 3 additional nodes MSa3, MSa4 and MSa5.

But the scaling policy checks that the maximum threshold is 4 and the total result of step up is going to be 5. In this case, the scaling policy restricts the total new nodes to only 2 rather than 3. Similarly,when the load eases out, the policy triggers the step-down function which reduces the number of nodes by 2 which brings the desired capacity to the minimum threshold level. This is the workflow for all the major orchestrating tools present in the current market.

C. CHALLENGES WITH THE CONFIGURATIONS

As discussed above, the orchestrating software mechanisms can keep spinning up or destroying container nodes dynamically all through the day based on the varying demand patterns and pre-configured policies. However, there is no standard way of determining the maximum threshold [13] for the number of containers or the pattern for configuring the step functions. Due to this, one has to configure the values based on approximation. The problem with this is that if the configurations for maximum threshold are on the lower side, then the system may not scale to the load. If the step-up function is not configured correctly, the users might get service unavailable when the load is high. On the other hand, if the values are configured on a higher side, it might lead to underutilization of compute resources and end up in monetary loss [14].

D. QUANTITATIVE APPROACH FOR SCALING CONFIGURATIONS

In order to determine the configurations for an auto scaling system, calculating the maximum threshold for the number of containers per service is the most key factor [15]. Once the maximum is determined, the other configurations like step up policies and step-down policies can be determined based on the estimated rate of change of load and the approximate time taken for starting the container. For calculating the maximum number of containers which are needed for the resilient operation, we need to determine the worst-case scenario for the system load and calculate the maximum resource consumption like CPU or Heap space or RAM under those conditions. For this calculation, we need to assume that the system satisfies certain pre-conditions – (I) System sees a sharp increase in load at a given short time interval (II) The load is evenly distributed across all the containers via a load balancer (III) There are a large number of user transactions in the given time interval. These conditions are put in place to calculate the total load on the system resources when there is a sudden increase in demand within a short span of time. In the real time scenario, it is more likely to have some pre-existing user transactions on the system when the sudden spike occurs. This would give the minimum number of containers needed to handle the worst-case scenario. For this calculation let the variables be defined as follows – Here is a quick walk through of the variables and symbols involved – (i) R(x) – This is any point of time (x) on the bell curve which represents the resource consumption of the system during the small interval of peak load. This value is calculated using the mathematical formulas below. (ii) RT(x) – This is the representation of total resource consumption like CPU utilization during the peak load scenario. This is the total area of the Bell curve. This value is also calculated using the mathematical formulas using Reiman' sum as explained below. (iii) CPU – this variable represents the average CPU utilization of an individual transaction. For calculating the total resource utilization RT(x) during the peak load scenario, all the possible critical and resource consuming transactions need to be considered. In a real-world scenario, these transactions are identified based on the analysis of the business use cases. The average CPU utilization of each transaction can be calculated by running the individual transaction under varying concurrent load scenarios and taking their average value. This process is demonstrated in the related work sections below. Since we are interested in the peak load scenario, it is better to consider at least the top 2 transactions. The total load can be more accurately be calculated if a greater number of transactions are considered. (iv) m, n – These variables represent the concurrency with which the transaction is performed during the peak load scenario. It is not possible to accurately predict the concurrency of each transaction, but with the business knowledge of the system and the number of users predicted during the peak load period, these values can be predicted. Since we are planning to prepare the system to handle the worst-case load scenario, it is suggested to take the higher predictions of concurrency into consideration. (v) T- this is the effective time range for calculating the total resource consumption in the peak load scenario. To calculate the number of containers needed to withstand the peak load scenario, we need to calculate the total resource utilization for the system during the effective time range. This effective interval of time is the short time interval when the user load spikes up suddenly and system running with standard number of nodes is able to tolerate this load till it calls for help by triggering the step-up function in auto scaling group to add more nodes to the cluster. Since we are trying to design a system which doesn't become unresponsive to the peak load situation, this interval can be calculated as the sum of time taken for the event to trigger the step up function, time taken to start a new container node in the cluster, time taken to start the micro service, time taken to perform a load balancer health check to start distributing the load to these new nodes. (vi) μ - The resource utilization during the effective time range is going to vary as the number of users using the system and the transactions involved keep fluctuating over time. This symbol indicates the mean value of the resource utilization during the peak load scenario. (vii) σ – This symbol indicates the standard deviation of resource utilization from the mean

TABLE 1. Variables and symbols.

Variable/symbol	Description
R(x)	Resource consumption at any point of time (x)
R_T (x)	Total resource consumption at any point of time (x)
CPU	CPU utilization per transaction
m,n	Random number of user transactions
T	Total time
μ	Mean
σ	Standard deviation

value due to the varying users and transactions. It is only an intermediate value used to derive the equations. Values need not be supplied for this.

Figure 3 given below shows the distribution of resource utilization for a system represented as a Gaussian normal distribution [16]. It represents the probability density function of the system resource usage or the Load function R(x) over the time range $-\infty$ to $+\infty$.

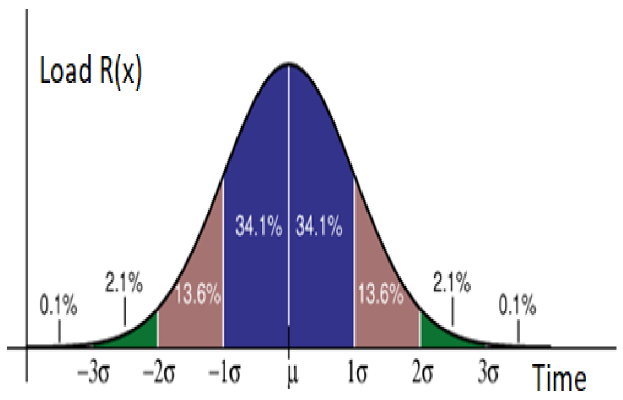

FIGURE 3. Gaussian distribution of system resource utilization.

Since the calculation is aimed at designing a system with minimum number of containers to handle the large and sudden increase of load situation, an effective time range need to be considered [17]. As shown in the Figure 3 above, the resource utilization for the time range of $[-2\sigma, 2\sigma]$ is 95.4%. Similarly, the load distribution for the time range $[-3\sigma, 3\sigma]$ is 99.6%. Since the remaining time doesn't have much effect on the total calculation, we can assume the total time for calculation purposes as 6σ.

$$T = 6\sigma \tag{1}$$

The standard deviation as shown in the figure 3 is σ and the mean is μ. Since the total time is considered within the range of $[-3\sigma, 3\sigma]$, the mean for this range becomes 3σ.

$$\mu = 3\sigma \tag{2}$$

From (1) and (2), it can be deduced as

$$\mu = T/2 \tag{3}$$

In mathematics, a Gaussian function is represented as below equation, where a is the height of the curve's peak, b is the position of the center of the peak and c is the standard deviation

$$f(x) = ae^{-(x\text{-}b)2/2c2} \tag{4}$$

When it is used to represent the probability distribution function of a normally distributed random variable with $\mu = b$ and $\sigma 2 = c2$, it takes the form

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{5}$$

When there are m user transactions which consume CPU_1 each and n transactions which consume CPU_2 each, the load function R(x) can be represented using the above equation as -

$$R(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{(x-\mu)^2}{\sigma^2}\right)} [CPU_1 \times m + CPU_2 \times n] \tag{6}$$

When the values for μ and σ are replaced in the above equation from (1) and (3), then it becomes -

$$R(x) = \frac{6}{T\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{(x-\frac{T}{2})^2}{(\frac{T}{6})^2}\right)} [CPU_1 \times m + CPU_2 \times n] \tag{7}$$

This above equation gives the resource consumption at a particular point of the gaussian distribution. To calculate the entire resource consumption, we need to take the integral of the above function to get the area of the curve. Reiman Sum can be used for the certain kinds of approximation of an integral by a finite sum. This can be represented in the form of an equation as -

$$S = \sum_{i=1}^{n} f(x_i)\Delta x_i \tag{8}$$

Based on Reiman Sum formula [18], the area of the curve is calculated by dividing the area into smaller rectangles and summing up the area of each rectangle to approximate the total area. Using this, we can calculate the area of the Bell curve shown in Figure 3, by diving the X axis into smaller parts and calculate the area of each partition. Since the calculation is based on the condition of sudden peak load at a short interval of time, we need to consider the effective time range where the new nodes can be added to the auto scaling group. Hypothetically, if a new node takes 60 seconds to be active, we can assume the time period to be 60 seconds and divide it into 6 equal partitions to find the area as given below. Using this the total resource consumption can be calculated as -

$$R_T(x) = R_{10}10 + R_{20}10 + R_{30}10 + R_{40}10 + R_{50}10 + R_{60}10 \tag{9}$$

This would give the total resource consumption when there is a sudden peak load on the system. When it is divided by the average proposed resource utilization of each container, the maximum number of resources needed for the system can be calculated. The system can be designed with the minimum number of containers needed per availability zone. It is better to start with 2 nodes per zone. An educated decision can be made for the minimum number based on the understanding of the system and the microservice which is provisioned. Once the minimum is determined and maximum is calculated, the a simple scaling policy can be designed by adding a step up (S_{up}) function which can handle the estimated rate of change of load (ΔR) and the time interval (T_s) which it takes to spin up a new container. Which means if the rate of change of load is estimated to be high in the amount of time it takes for the container to start, the step function can be made broader.

$$S_{up} \propto (\Delta R)(T_s) \tag{10}$$

Similarly, a step-down function can be calculated in such a way that it doesn't cause a load spike and trigger a step-up function again. This would cause a circular effect. In order to avoid this, a step down can be configured for a resource utilization which is considerably far from the optimal utilization.

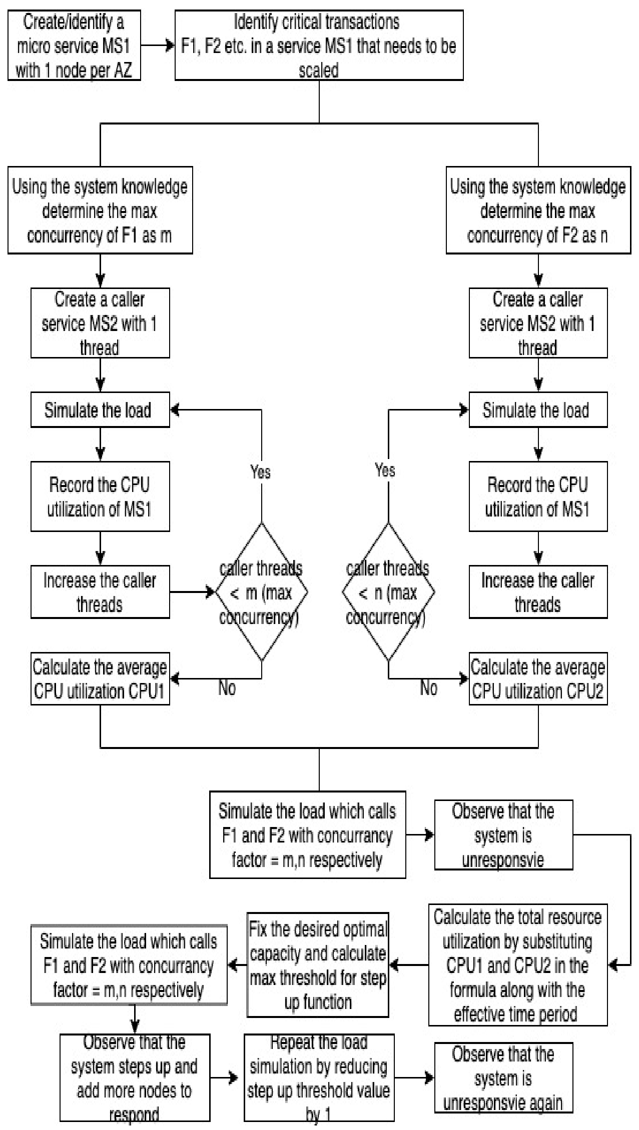
Create/identify a micro service MS1 with 1 node per AZ
Identify critical transactions F1, F2 etc. in a service MS1 that needs to be scaled
Using the system knowledge determine the max concurrency of F1 as m
Using the system knowledge determine the max concurrency of F2 as n
Create a caller service MS2 with 1 thread
Create a caller service MS2 with 1 thread
Simulate the load
Simulate the load
Record the CPU utilization of MS1
Record the CPU utilization of MS1
Increase the caller threads
Increase the caller threads
Yes
Yes
caller threads < m (max concurrency)
caller threads < n (max concurrency)
No
No
Calculate the average CPU utilization CPU1
Calculate the average CPU utilization CPU2
Simulate the load which calls F1 and F2 with concurrency factor = m,n respectively
Observe that the system is unresponsvie
Calculate the total resource utilization by substituting CPU1 and CPU2 in the formula along with the effective time period
Fix the desired optimal capacity and calculate max threshold for step up function
Simulate the load which calls F1 and F2 with concurrency factor = m,n respectively
Observe that the system steps up and add more nodes to respond
Repeat the load simulation by reducing step up threshold value by 1
Observe that the system is unresponsvie again

FIGURE 4. Block diagram showing the work process.

IV. RESULTS AND DISCUSSION

We have implemented a research project and simulated various scenarios to check the validity of the above mathematical equations. The work which we did can be understood from the block diagram shown in Figure 4. For this work, we developed microservices which are written in java programming language using spring boot technology. Using this we developed a host micro service MS1 which has 2 functions – a large resource consuming function F1, a relatively small resource consuming method F2 and a caller micro service MS2 which can spoof the multiple user transactions by invoking the functions F1 and F2 in MS1 at a small interval of time, by spinning up multiple parallel threads based on the input parameter supplied. This is done to simulate a sharp increase in user load which spans multiple transactions where each transaction consumes its own amount of CPU and RAM [20], [21]. These micro services were packaged as docker containers. Monitoring and dashboarding configurations like Graphana and Prometheus were configured in conjunction with the microservices to record the CPU utilization. These dockerized micro services were deployed on ECS (elastic container service) cluster in AWS. This container orchestration is comparable with other prominent software in the market like Kubernetes, Docker swarm. An application load balancer is used to distribute the load between the containers [19]. This whole setup is coded as infrastructure as code (IAC) using Hashicorp's Terraform software libraries. The high-level architecture for this setup can be visualized from the Figure 5 below.

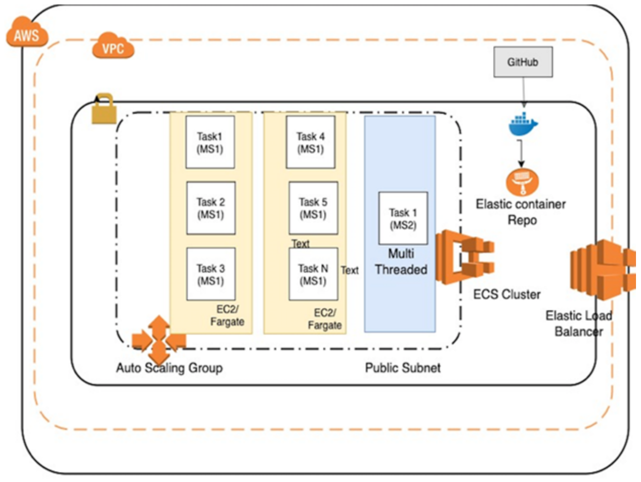

FIGURE 5. Container deployment architecture for microservices in public cloud.

Run1: In order to calculate the resource utilization (CPU_1) of a user transaction, we simulated the scenario by invoking the calling service MS2 by passing a input parameter of 1. This parameter of 1 spun up 1 thread of the calling service. This calling service was made to invoke a large function F1 running in the microservice MS1. MS1 was made to run as a single instance in the ECS cluster with a run configuration of 80 MB RAM (Xmx = 80m). We used a lower capacity server T2 micro for test purposes with a lower Xmx for the microservice, because we were running the test with a dummy large function which was designed for test purposes. Once the requests were made by the calling service, we recorded the CPU utilization using the Graphana setup as shown in Figure 6 below. As it can be seen from the graph above, the large function consumed a little over 18% of the resources. When we repeated the same exercise one more time with the same configurations and same parameters. This time the CPU utilization was little over 18%. We performed

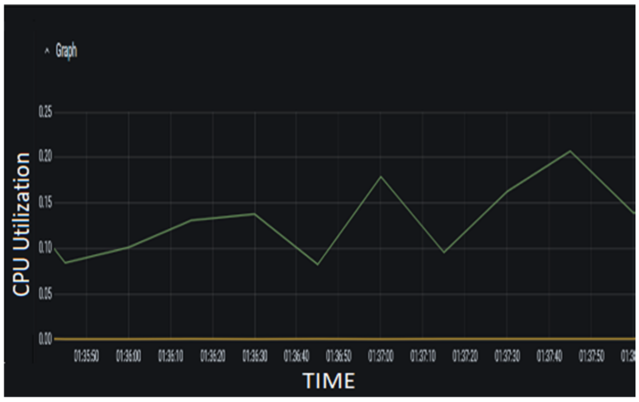

FIGURE 6. CPU utilization trend for large function.

few more rounds of this by increasing the number of threads to 2, 3 and performed few iterations. We determined the average to be 19%. With this we determined the resource utilization (CPU_1) for the large function as 19%.

Run2: Then we performed the next run by passing a parameter 1 for the calling service MS2, there by invoking a single thread. But this time we invoked a smaller function F2 running in microservice MS1. The resource utilization was measured using Graphana dashboard as shown in Figure 7 below. We repeated the same test couple of times by varying the number of threads in each case and performing few iterations for each input. We repeated this exercise multiple times and found the average utilization was 10%. It can be noted that the CPU utilization was measured roughly to be around 10With this we determined the resource utilization (CPU_2) for the small function as 19%.

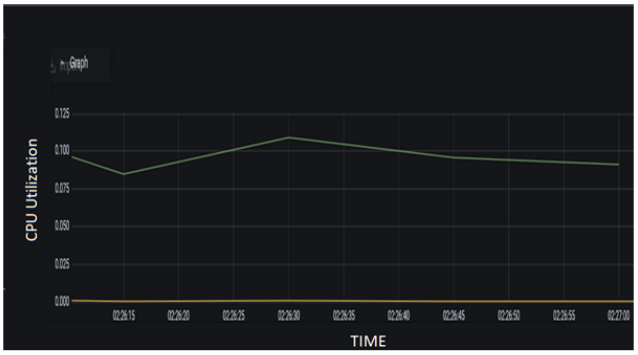

FIGURE 7. CPU utilization trend for the small function.

Run3: Now to simulate a real time scenario where there is a large set of user transactions performing different functions on a microservice, we invoked the large function F1 and small function F2 simultaneously by spinning up 10 parallel threads of the calling service. We noticed that the CPU utilization was showing a rapid upward trend for a short time and then it did not show any further movement. At this point the microservice MS1 became unresponsive as it was not able to handle the load. We also observed that there was no response code returned after some time. Once the web socket time out exception occurred, it gave 503 as the response code. We repeated the same test multiple times by adjusting the number of input threads. We noticed that the pattern was repeatable. One thing to note from the observations was that as the number of threads were becoming less the amount of time the service tried to cater the requests increased. This can be compared to a real time situation when a service which is running in a cluster of nodes gets a sudden and rapid increase in user demand. When this happens the load balancer tries to balance the load on all the containers by evenly distributing it. This might sustain the load for some time till the servers become unhealthy and unresponsive due to the increasing load.

To handle such real time load fluctuations an auto scaling system configuration is necessary. Also, the step-up function needs to be in place which can spin up right number of containers to handle the increase of load. While setting up this policy the rate of change of load and the startup time of each container needs to be considered to provide the right step increase. However, the most important factor is to determine the maximum number of containers needed to handle the total load. Once the maximum is determined, calculating the other parameters might not be too difficult.

To handle the Run3 successfully, we need to setup the autoscaling function with a new maximum number of containers needed for MS2. For this calculation, the values of CPU_1 and CPU_2 are determined to be 18% and 10% approximately in the above run outcomes. The maximum number of user transactions which we want to scale the system for is 10 as per Run3 inputs. That makes the value of m, n in equation (7) as 10. Using these values, the resource utilization at any point of time T can be calculated as per equation (7). To calculate the total resource utilization, we used equation (9) based on Reiman sum, by splitting the time intervals as 10,20,30,40,50 and 60 seconds. This gave us a total CPU utilization of 392% approximately. Now to calculate the maximum number of instances needed in the cluster, we divided the total utilization by the optimal resource load which each node in the cluster is capable of handling for a smooth run. We assumed this optimal load as 60% per node. That gave the maximum threshold of $392/60 = 7$ nodes in the cluster.

Run4: Now that we calculated the maximum nodes, we repeated the Run 3 again by setting up an auto scale policy. We setup the initial cluster size as 3 and the step function size as 4. When we replicated the scenario mentioned for Run3 and monitored the usage, the CPU utilization graph went above 60% (desired optimal value) when the load increased. As it can be seen from the graph in Figure 8 below, it continued to peak for around 50 seconds while the step-up function triggered 4 new nodes. Once the new nodes were active and load started to get distributed and the utilization curve flattened for a bit and then showed the downward trend. When the similar experiment is repeated with a step value less than the calculated value, the system again becomes unresponsive. This behavior can be prorated and compared

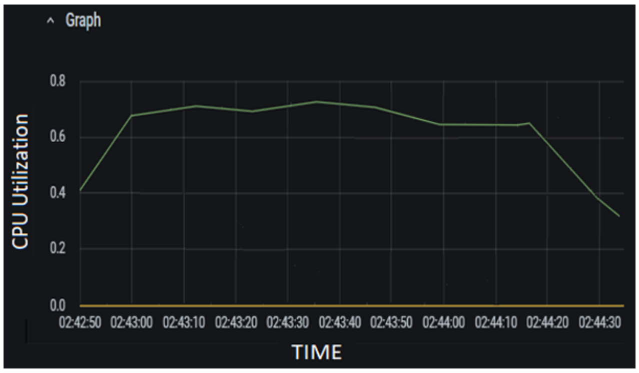

FIGURE 8. CPU utilization trend for an autoscaling cluster with step up policy.

to large number of nodes when serving a real time load in production environment.

V. CONCLUSION

With the rapid growth of public cloud technologies and ever-increasing demand of web applications [22], [23], autoscaling is an absolute necessary feature to achieve the horizontal scaling needed to handle sudden load fluctuations in any live environment. With the hugely distributed hybrid cloud infrastructures involved in all the large enterprises, there is a very likely chance that the resource utilization may fall out of optimal levels. Determining the maximum threshold number for the containers in a cluster is very important for designing the autoscaling policies and functions.

As shown in this paper the maximum threshold can be attempted by using a combination of - (1) mathematical equations, (2) a clear understanding of all the critical functions and transactions involved in each of the micro service, (3) practical simulations to calculate the load under controlled environment and observation of results, (4) and finally a clear understanding of the business function of the service and approximate rate of change of load. This paper attempts to calculate the maximum threshold using the 4 considerations mentioned above. Further scope of research work includes adding more considerations while performing the calculations such as formulas to account for prior existing load when the peak load occurs, uneven distribution of load to the containers in the cluster by the load balancer either due to the health of the container or proximity of the container to the incoming load etc. Further research scope also includes determining the mathematical ways to calculate the step up and step-down functions based on rate of change of load.

REFERENCES

[1] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: State-of-the-art and research challenges," *J. Internet Services Appl.*, vol. 1, no. 1, pp. 7–18, Apr. 2010, doi: 10.1007/s13174-010-0007-6.

[2] Z. Mahmood, "Cloud computing: Characteristics and deployment approaches," in *Proc. 11th IEEE Int. Conf. Comput. Inf. Technol.* Washington, DC, USA: IEEE Computer Society, Aug./Sep. 2011, pp. 121–126, doi: 10.1109/CIT.2011.75.

[3] N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017, doi: 10.1007/978-3-319-67425-4_12.

[4] C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in microservice architecture," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 8, pp. 18–22, 2018, doi: 10.14569/IJACSA.2018.090804.

[5] A. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1994–2004, doi: 10.1109/ICDCS.2019.00197.

[6] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Trans. Cloud Comput.*, early access, Apr. 6, 2020, doi: 10.1109/TCC.2020.2985352.

[7] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jul. 2019, pp. 68–75.

[8] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a service," Dept. Elect. Comput. Eng., Univ. Toronto, Toronto, ON, Canada, Tech. Rep., 2017.

[9] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang, "Auto-scaling microservices on IaaS under SLA with cost-effective framework," in *Proc. 10th Int. Conf. Adv. Comput. Intell. (ICACI)*, Mar. 2018, pp. 583–588, doi: 10.1109/ICACI.2018.8377525.

[10] Y. Hirashima, K. Yamasaki, and M. Nagura, "Proactive-reactive auto-scaling mechanism for unpredictable load change," in *Proc. 5th IIAI Int. Congr. Adv. Appl. Inform. (IIAI-AAI)*, Jul. 2016, pp. 861–866, doi: 10.1109/IIAI-AAI.2016.180.

[11] T. Fankhauser, Q. Wang, A. Gerlicher, C. Grecos, and X. Wang, "Web scaling frameworks for Web services in the cloud," *IEEE Trans. Services Comput.*, vol. 9, no. 5, pp. 728–741, Sep. 2016, doi: 10.1109/TSC.2015.2454272.

[12] J. Jiang, J. Lu, G. Zhang, and G. Long, "Optimal cloud resource auto-scaling for Web applications," in *Proc. 13th IEEE/ACM Int. Symp. Cluster, Cloud, Grid Comput.*, May 2013, pp. 58–65, doi: 10.1109/CCGrid.2013.73.

[13] M. Masdari and A. Khoshnevis, "A survey and classification of the workload forecasting methods in cloud computing," *Cluster Comput.*, early access, Dec. 2019, doi: 10.1007/s10586-019-03010-3.

[14] M. S. Das, A. Govardhan, and D. V. Lakshmi, "Cost minimization through load balancing and effective resource utilization in cloud-based Web services," *Int. J. Natural Comput. Res.*, vol. 8, no. 2, pp. 51–74, Apr. 2019, doi: 10.4018/IJNCR.2019040103.

[15] J. Novak, S. Kasera, and R. Stutsman, "Cloud functions for fast and robust resource auto-scaling," in *Proc. 11th Int. Conf. Commun. Syst. Netw. (COMSNETS)*, Jan. 209, pp. 133–140, doi: 10.1109/COMSNETS.2019.8711058.

[16] G. L. Squires, *Practical Physics*, 4th ed. Cambridge, U.K.: Cambridge Univ. Press, doi: 10.1017/cbo9781139164498.

[17] A. Hamon. *Do the Math: Scaling Microservices Applications With Orchestrators*. [Online]. Available: https://www.toptal.com/devops/scaling-microservices-applications

[18] D. Hughes-Hallett, *Calculus*, 4th ed. Hoboken, NJ, USA: Wiley, 2005, p. 252.

[19] T. Chaabouni and M. Khemakhem, "Cloud computing: A new vision of the distributed system," MIRACL Lab, FSEG, Univ. Sfax, Sfax, Tunisia, Tech. Rep., 2012.

[20] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert, "Self-managing cloud-native applications: Design, implementation, and experience," *Future Gener. Comput. Syst.*, vol. 72, pp. 165–179, Jul. 2017, doi: 10.1016/j.future.2016.09.002.

[21] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, vol. 49, 2011, pp. 1–12, doi: 10.1145/2063384.2063449.

[22] C. Richardson, "Pattern: Monolithic architecture," *Microservices*, Oct. 2018.

[23] D. Goel and A. Nayak, "Reactive microservices in commodity resources," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2019, pp. 3658–3665, doi: 10.1109/BigData47090.2019.9006584.

[24] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices," *IEEE Trans. Services Comput.*, early access, May 20, 2020, doi: 10.1109/TSC.2020.2995937.

[25] J. Kumar and A. K. Singh, ''Workload prediction in cloud using artificial neural network and adaptive differential evolution,'' *Future Gener. Comput. Syst.*, vol. 81, pp. 41–52, Apr. 2018.

[26] M. C. Calzarossa, L. Massari, and D. Tessera, ''Evaluation of cloud autoscaling strategies under different incoming workload patterns,'' *Concurrency Comput., Pract. Exper.*, vol. 32, no. 17, p. e5667, Sep. 2020, doi: 10.1002/cpe.5667.

[27] P. Pereira, J. Araujo, and P. Maciel, ''A hybrid mechanism of horizontal auto-scaling based on thresholds and time series,'' in *Proc. IEEE Int. Conf. Syst., Man Cybern. (SMC)*, Oct. 2019, pp. 2065–2070, doi: 10.1109/SMC.2019.8914522.

[28] P. Pereira, J. Araujo, M. Torquato, J. Dantas, C. Melo, and P. Maciel, ''Stochastic performance model for Web server capacity planning in fog computing,'' *J. Supercomputing*, early access, Feb. 2020, doi: 10.1007/s11227-020-03218-w.

[29] J. Kumar and A. K. Singh, ''Decomposition based cloud resource demand prediction using extreme learning machines,'' *J. Netw. Syst. Manage.*, vol. 28, no. 4, pp. 1775–1793, Oct. 2020, doi: 10.1007/s10922-020-09557-6.

[30] T. Reis, M. Teixeira, J. Almeida, and A. Paiva, ''A recommender for resource allocation in compute clouds using genetic algorithms and SVR,'' *IEEE Latin Amer. Trans.*, vol. 18, no. 6, pp. 1049–1056, Jun. 2020, doi: 10.1109/TLA.2020.9099682.

CHAITANYA KRISHNA RUDRABHATLA (Member, IEEE) received the B.S. degree in electrical and electronics engineering from Kakatiya University, India, in 2004. He is currently a Technology Leader and a Senior Solutions Architect with over 15 years of experience in Media, Entertainment, Telecom and consulting firms, with a proven ability to research and develop efficient, salable, and fault-tolerant solutions for complex problems and enterprise applications. He is also working as an Executive Director–Solutions Architect with Sony Pictures Entertainment, Los Angeles, CA, USA.