OPTIMIZATION OF DEPLOYMENTS FOR SERVICE ORIENTED CLOUDS

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Chaitanya Dumpala

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

November 2015

Fargo, North Dakota

North Dakota State University
Graduate School

Title

OPTIMIZATION OF DEPLOYMENTS FOR SERVICE ORIENTED CLOUDS

By

Chaitanya Dumpala

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig
Chair

Dr. Jun Kong

Dr. Chao You

Approved:

11/10/2015
Date

Dr. Brain Slator
Department Chair

ABSTRACT

In cloud-computing environments, every service in an application is deployed as a different service instance. All service instances are deployed at a different level of end-to-end Quality of Service (QoS), which are described in the Service Level Agreement (SLA). To satisfy the given SLAs and end-to-end QoS requirements of an application, the application is required to optimize its deployment configuration of service instances. In this paper, a genetic algorithm is implemented, as proposed in literature, to solve this problem by searching for the optimal solution from a search space while satisfying the given SLA. The algorithm estimates the performance of an application by minimizing the latency allowing SLAs to be defined in a probabilistic manner. Simulation results demonstrate that the genetic algorithm implemented in this paper obtains the deployment configurations that satisfy the given SLA.

ACKNOWLEDGMENTS

I would like to express my sincere thanks to my adviser, Dr. Simone Ludwig, for her continued support throughout this paper. I am grateful for the ideas and suggestions given by my adviser, and the amount of guidance given by her is enormous. Also, special thanks to my advisory committee members, Dr. Jun Kong, and Dr. Chao You, for their valuable input, which helped me, complete this paper.

Finally, words alone cannot express the thanks I owe to my family members and friends for their enormous support.

I, once again, thank my adviser, Dr. Simone Ludwig, for her guidance throughout this paper.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. Cloud Computing

Cloud Computing (CC), or in simpler shorthand just "the cloud", focuses on maximizing the effectiveness of the shared resources. Cloud resources are usually not only shared by multiple users but are also dynamically reallocated per demand. CC is a general term for anything that involves delivering hosted services over the Internet. CC can be defined as a new style of computing in which dynamically scalable and often-virtualized resources are provided as services over the Internet [1].

The goal of CC is to provide end users with a considerable processing power and computing resources that allow them to run the applications and other user requirements. In general, CC depends on the power and resources of computer networks. With this architecture, clients have access to the resources provided by the cloud provider as described in their Service Level Agreement (SLA). Clouds using virtualization technology and data centers allocate distributed resources for clients, as they need. Often traditional scheduling techniques [2, 3] and allocation strategies [4] cannot be used in cloud computing, in which the number of end user requests increases and decreases over time in an unpredictable way. This leads to difficulties of analysis and discovery of information from incoming requests to distribute the available resources according to user requirements and constraints of cloud provider. Similarly, unpredictable requests due to the increased costs of server load, the total execution time of the task, and the difficulty of making an optimal decision in the whole group of tasks. Amazon EC2 [5], introduces cloud services that allow users to acquire and release resources on-demand. Amazon EC2 also allows workflow systems to increase and decrease the pool of available

resources when the demands are changing and unpredictable needs of users are involved in the allocation process.

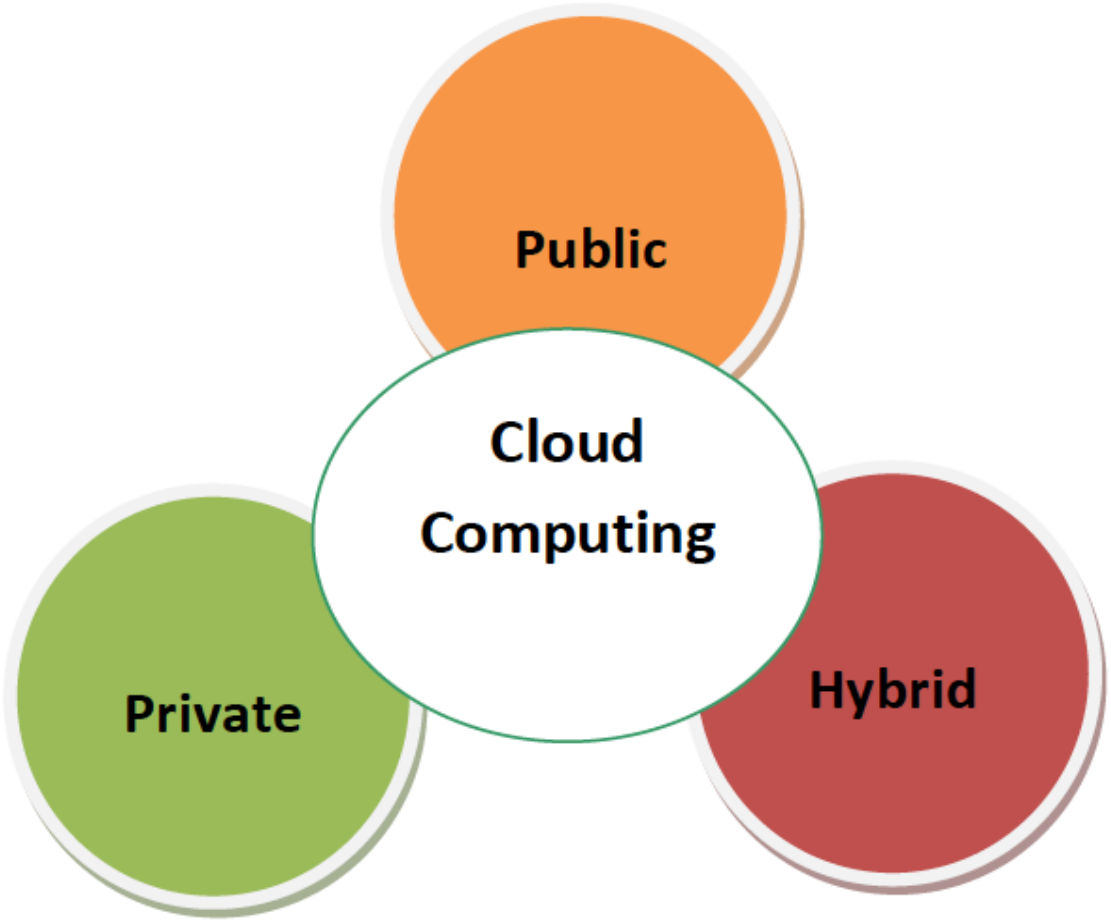

Figure 1. Types of cloud computing

Cloud computing allows users to run applications remotely, as shown in Figure 1. The first is the public cloud services, which can be sold to anyone on the Internet (e.g., Amazon Elastic Compute Cloud (EC2) [5] and Google App Engine [6]). The second type of cloud is a private cloud that supplies hosted services to a limited number of customers (end users). In the type of hybrid cloud, the infrastructure is a composition of two or more clouds (private, community or public) that remain unique entities but are bound together by standardized or proprietary technology. In general, clouds are deployed to clients by giving them three access

levels. These services are broadly divided into three categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (Saas).

1.1.1. Software as a Service

Software-as-a-Service (SaaS) is a software distribution model in which applications are accessible through a single interface, like a web browser over the Internet. Users do not have to consider the underlying cloud infrastructure including servers, storage, platforms, etc.

1.1.2. Platform as a Service

Platform-as-a-Service (PaaS) provides a high level of integrated applications that control of distributed applications and their hosting environment configurations. In general, developers accept all instructions on the type of software that can be written to change built-in scalability.

1.1.3. Infrastructure as a Service

Infrastructure-as-a-Service (IaaS) provides users with computation processing, storage, networks and computing resources. IaaS users can implement an arbitrary application, which is able to grow up and down dynamically. Also, IaaS sends programs and related data, while the cloud provider does the computation processing and returns the result.

This paper focuses on service-oriented applications in cloud computing environment Software-as-Service (SaaS).

1.2. Service Deployment Optimization

The main objective of this paper is optimizing service deployments in cloud computing environments. The deployment of a service can be done in one or more service instances. A service can be operated at different end-to-end Quality of Service (QOS) levels. A genetic algorithm is implemented in this paper by leveraging queuing theory, in order to optimize the deployment configuration of service instances by the application to satisfy given SLAs.

Simulation results demonstrate that the algorithm efficiently minimizes the latency to obtain deployment configurations that satisfy given SLA.

This paper envisions service-oriented applications in cloud computing environments. A service-oriented application consists of a set of services and a workflow. Each service encapsulates the function of an application component. Each workflow defines how services interact with each other. When an application is uploaded to a cloud, the cloud deploys each service in the application as one or more service instances. Each service instance runs on a process or a thread and operates based on a particular deployment plan. Different service instances operate at different quality of service levels. For example, Amazon Elastic [5] Compute Cloud offers five different deployment plans that allow service instances to yield different QoS levels by providing different amounts of resources at different prices. If an application is intended to serve different categories of users (e.g. users with for-fee and free memberships), it is instantiated with multiple workflow instances, each of which is responsible for offering a specific QoS level to a particular user category. A SLA is defined upon a workflow as its end-to-end QoS requirements such as throughput, latency and cost (e.g., resource utilization fees). In order to satisfy given SLAs, application developers (or cloud engineers) are required to optimize a deployment configuration of service instances for each user category by considering which deployment plans and how many service instances to use for each service. For example, a deployment configuration may be intended to improve the latency of a heavily accessed service by deploying its instance with an expensive deployment plan that allocates a large amount of resources.

1.3. Genetic Algorithms

Genetic algorithms are one of the best ways to solve a problem for which little is known, they use the principles of selection and evolution to produce several solutions to a given problem.

Genetic algorithms tend to thrive in an environment in which there is a very large set of candidate solutions and in which the search space is uneven. The basic process for a genetic algorithm is:

1. Initialization - Create an initial population. This population is usually randomly generated and can be of any desired size, from only a few individuals to thousands.
2. Evaluation - Each member of the population is then evaluated and we calculate a 'fitness' for that individual. The fitness value is calculated by how well it fits with our desired requirements. These requirements could be simple, 'faster algorithms are better', or more complex, 'stronger materials are better but they shouldn't be too heavy'.
3. Selection - We want to be constantly improving our population’s overall fitness. Selection helps us to do this by discarding the bad designs and only keeping the best individuals in the population. There are a few different selection methods but the basic idea is the same, make it more likely that fitter individuals will be selected for our next generation.
4. Crossover - During crossover we create new individuals by combining aspects of our selected individuals. The hope is that by combining certain traits from two or more individuals we will create an even 'fitter' offspring, which will inherit the best traits from each of its parents.

5. Mutation - We need to add a little bit of randomness into our population's genetics otherwise every combination of solutions we can create would be in our initial population. Mutation typically works by making very small changes at random to an individual genome.
6. And repeat! - Now we have our next generation we can start again from step two until we reach a termination condition.

Evolutionary algorithms are typically used to provide good approximate solutions to problems that cannot be solved easily using other techniques such as linear programming.

2. RELATED WORK

A number of research efforts have investigated the SSDO problem [7-10]. Based on previously proposed GA [11], this paper describes and implements SSDO Problem in Cloud computing environments.

Simple Additive Weighting (SAW) is the most widely used method by defining a fitness function [12] to solve the SSDO problem and a number of research efforts which are based on Search Based Software Engineering have investigated this to seek an optimal solution [13-15]; however, it is difficult to determine weight values in a fitness function as objectives often have different value ranges and priorities. And also, it is known that for non-trivial optimization problems, SAW does not work well as it is mainly designed to seek a single optimal solution.

Linear Programming is another major method used to solve SSDO problem [16, 17] but it is not suitable, as it cannot reveal the trade-offs among Quality of service objectives. In linear programming, the computational time grows exponentially with the search space size and it is not trivial to define the problem in linear form and it is not scalable [18-21].

This paper leverages a multi objective GA as a software base search engine technique to solve large-scale SSDO problems, as GA's scale better than linear programming. Several research efforts have investigated multi objective GAs for the SSDO problem [22-24]. However, most of them do not support the assumptions that the current cloud computing platforms make. For example, they do not consider binding multiple service instances to a service and do not consider differentiated SLAs and support service deployment configurations that model applications in cloud computing.

3. PROBLEM STATEMENT

3.1. Problem

In a Cloud environment, a SLA is defined upon a workflow as its end–to–end QoS requirements such as latency, throughput and cost. In order to satisfy given SLAs, cloud engineers are required to optimize a deployment configuration of service instances for each user category by considering which deployment plans and how many service instances to use for each service.

This decision-making problem, called the SLA-aware service deployment optimization (SSDO) problem [1], is a combinatorial optimization problem that searches the optimal combinations of service instances and deployment plans. There exist three research issues in this problem. First, it is known NP-hard, which can take a significant amount of time, labor and costs to find the optimal deployment configurations from a huge search space (i.e. a huge number of possible combinations of service instances and deployment plans). The second issue is that the SSDO problem often faces trade-offs among conflicting QoS objectives in SLAs. For example, in order to reduce its latency, a service instance may be deployed with an expensive deployment plan; however, this is against another objective to reduce cost. Moreover, if the service's latency is excessively reduced for a user category, the other user categories may not be able to satisfy their latency requirements. The third issue is that traditional SLAs often consider QoS requirements as their average (e.g., average latency). This fails to consider fluctuation/variance in runtime QoS measures.

This paper is closely aligned with the paper in [1] by implementing and evaluating the authors' proposed optimization algorithm, which addresses these research issues. Given multiple configurations, application developers can better understand the trades among QoS objectives

and make a well-informed decision to choose the best deployment configuration for them according to their requirements and preferences.

Simulation results demonstrate that quality deployment configurations that satisfy given SLAs by heuristically examining very limited regions in the entire search space.

3.2. Work Flow and Service Deployment

In a Cloud environment, a workflow consists of a set of services connected based on service dependencies. An example workflow in Figure 2 consists of four services. It has a branch after Service 1 and executes Service 2 and Service 3 in parallel and Service 4. In order to process requests, each service is instantiated as a service instance and deployed on a particular deployment plan. A set of service instances and deployment plans is collectively called a deployment configuration.

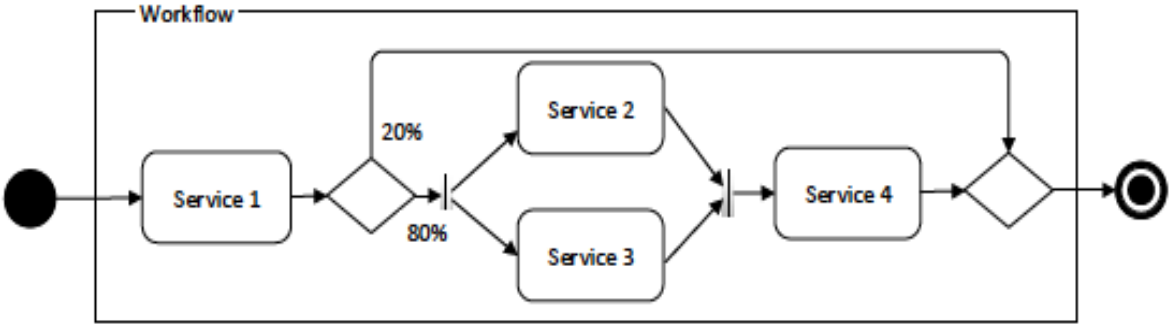

Figure 2. Example of a workflow

In Figure 3, Service 1 is instantiated as three service instances and deployed on two-deployment plan 1 and one deployment plan 3. We assume that a deployment plan cannot operate more than two service instances of the same service, but it can have instances of different services at a time. Deployment plan 2 in Figure 3 has two service instances. Each deployment configuration can have arbitrary number of deployment plans and service instances to improve

the service's latency. A deployment configuration is assumed to have one access point equipped with a load balancer (Figure 3). Load balancer's algorithm must be the same as that of a load balancer that an application under development uses. Currently a round robin and CPU load balancing algorithm is supported, which dispatches requests to balance deployment plans CPU usage. When a deployment configuration uses a round robin load balancer, requests for a certain service are dispatched to corresponding service instances with equal probability. For example, in Figure 2, when a deployment configuration receives 1,800 requests for Service 1, every second each instance of Service 1 receives 600 requests for every second since three instances are deployed in deployment configuration.

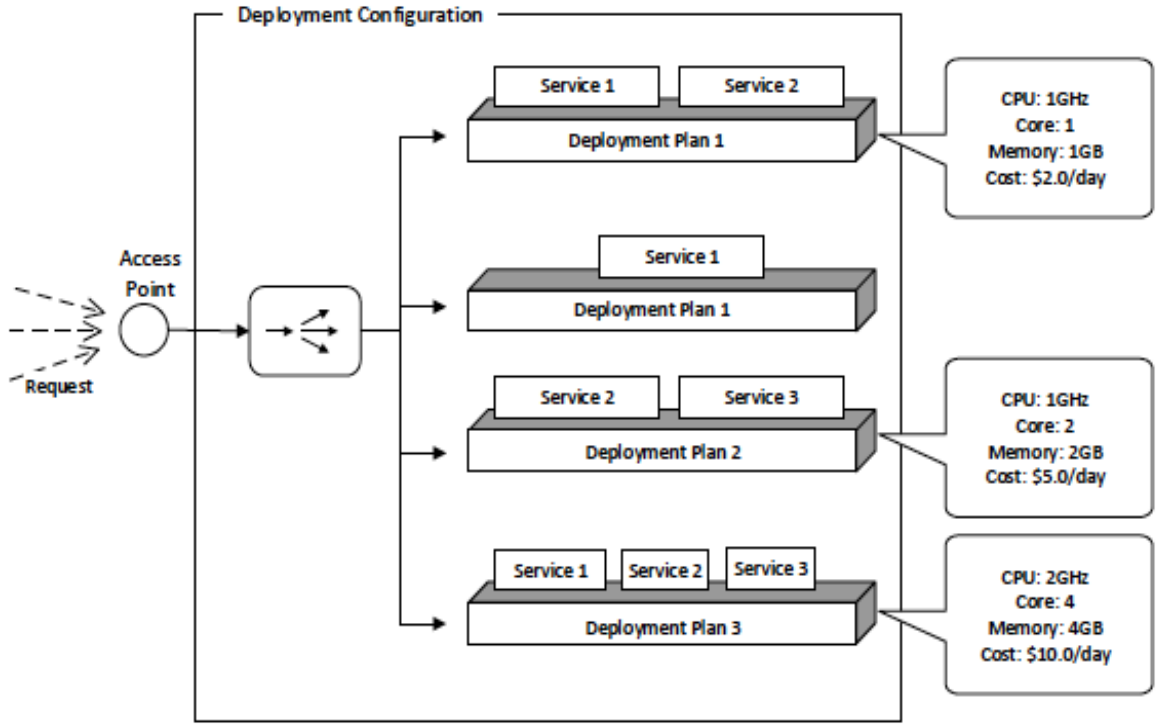

Figure 3. Example of a Deployment Configuration

Figure 3 illustrates how an application processes requests from users. When a user sends a request to an application, it calls a series of service instances in its deployment configuration

according to its workflow and sends the response back to the user. In this example, once receiving a request from a user, an application calls one of Service 1's instances, waits for a response from the service instance, calls Service 2 and Service 3 instances in parallel, waits for responses from them, then calls the Service 4 instance. A SLA is defined with end-to-end throughput, latency and cost of an application. In order to judge whether an application satisfies a given SLA, it is required to examine its end-to-end QoS by aggregating measures of individual services.

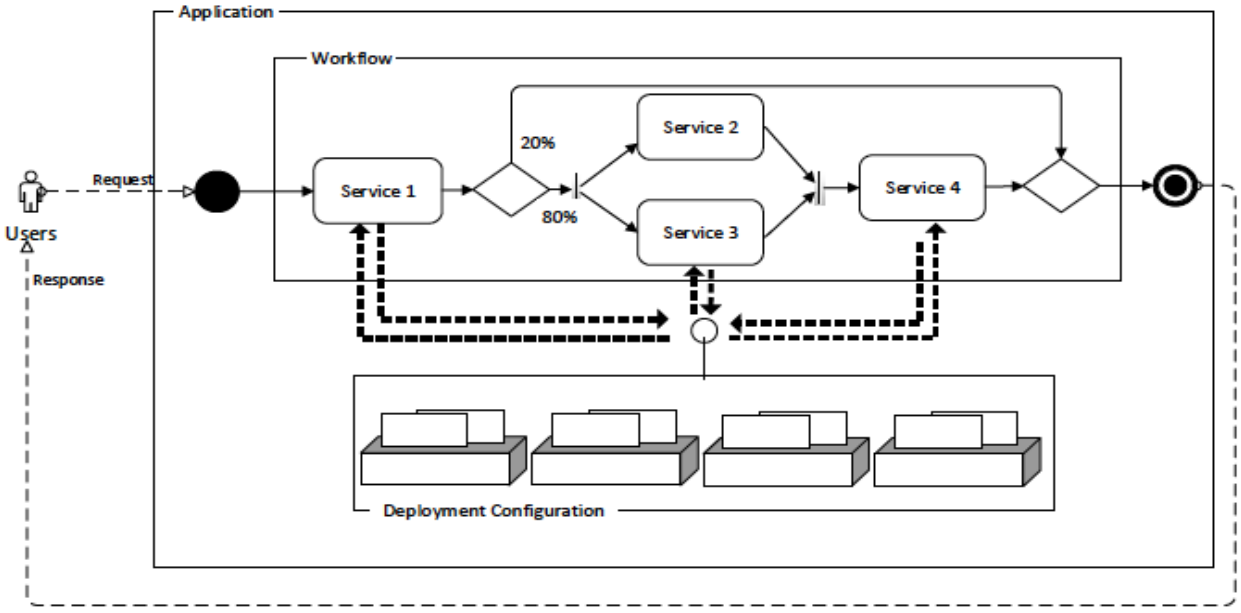

Figure 4. Example of an application

In this algorithm, the maximum cost of the application is fixed. Therefore, each workflow will not have deployment plans more than the summation of each deployment plans cost for each user category. The end-to-end throughput is same as the throughput defined in the SLA and latency is obtained by leveraging queuing theory to the deployment configurations.

3.3. Latency Distribution of Service Instances

In order to estimate the performance of distributed systems, queuing theory is a well-established method used in a large number of research studies. The algorithm estimates each service instance's latency in a workflow by leveraging queuing theory, and obtains the end-to–end latency by applying QoS aggregate functions.

Consider u_u be the mean unit service rate, which is the average number of requests processed per unit time by a service instance running on a unit CPU (e.g., 1GHz CPU). u_u is the inverse of the mean request processing time. According to queuing theory, when a service instance runs on a deployment plan with various CPU configurations, and under various request arrival rate, it estimates probability distribution of latency when a service instance runs on a deployment plan, if the mean unit service rate u_u of each service instance is known. From Equation 1, λ is the mean request arrival rate, which is the average number of requests to a service instance per unit time. By assuming that all services mean unit service rates in the work flow are know, we can calculate the probability that the latency (waiting time W in a queue) if the mean request arrival rate λ of every service instance runs on a deployment plan with n CPU cores each of then is p times faster than a unit CPU is given [10,21].

$$\Pr(W \geq \alpha) = \left[{}^{nP_n}/_{n-\rho} \right] \exp(-2(n-\rho)\alpha)$$

$$\rho = {}^{\lambda}/_{(p\mu_\mu)} \tag{1}$$

$$p_n = p_0(\rho^n/n!)$$

$$p_0 = \frac{1}{\left[\sum_{k=0}^{n-1} \frac{p^k}{k!} + \frac{\rho^n}{(n-1)!\,(n-\rho)}\right]}$$

For example consider a deployment plan in which multiple service instances are deployed as shown in the Figure 5, where Service a and b run on a deployment plan. Let $u_{ua}$and u_{ub} be the mean unit request rate of service a and b, respectively. λ_a and λ_b be the mean request arrival rates of the services. Here as multiple service instances run on one deployment plan, portion of CPU power is assigned to each service instance based on their CPU usage. From Figure 5, for each CPU core, service a and b occupy $\rho_a = {}^{\lambda_a}/_{(npu_{ua})}$ and $\rho_b = {}^{\lambda_b}/_{(npu_{ub})}$ of CPU power. Therefore, for each CPU core, (1 - ρ_b) and (1- ρ_a) are available portion of CPU for service a and service b, respectively. As each service cannot use complete CPU power, their service rate are reduced to ${}^{pu_{ua}}/_{(1-\rho_b)}$ and ${}^{pu_{ub}}/_{(1-\rho_a)}$. The reduced service rates are applied to Equation 1 when calculating the service instance's latency.

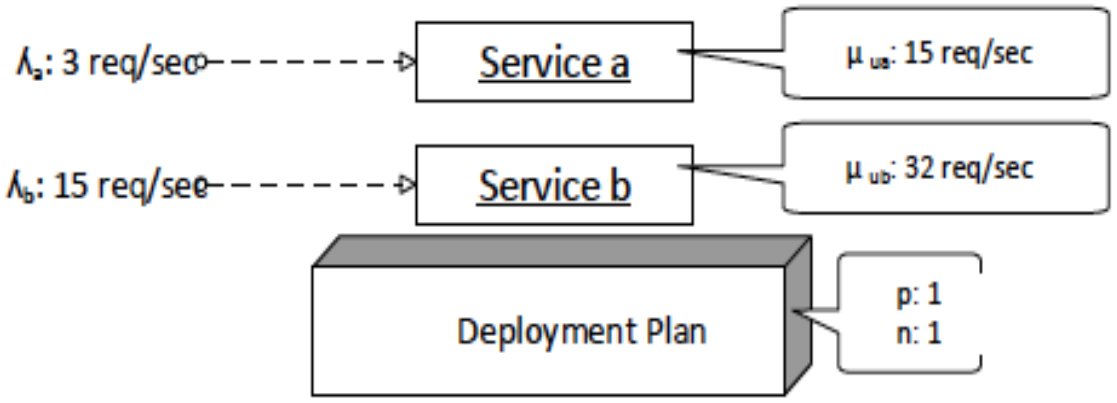

Figure 5. Example of services in deployment plan

When CPU usage exceeds its power, the service rates are reduced so that $\rho_a + \rho_b = 1$. In order to obtain the probability distribution of the end to-end latency, a Monte Carlo method has been adapted since a simple aggregation (e.g., summation of each service's average latency) cannot reveal the probability distribution and lead to a too pessimistic estimation.

The algorithm simulates the end-to-end latency of one request by:

1. Selecting services to execute (a path in a workflow) according to branching probabilities (all services are executed if a workflow has no branches),
2. For each service selecting one of instances according to their throughput (an instance with larger throughput has higher chance to be selected),
3. By determining each instance's latency (a certain value) according to the probability distribution, and
4. By aggregating the latency by applying aggregate functions in Table 1. By repeating this process many times, i.e., simulating many requests to an application, the algorithm approximates the probability distribution of the end-to-end latency.

Table 1. Latency Aggregation

	Sequence	Parallel
Latency (L)	$\sum_{s \rightarrow Services} L_s$	$\max_{s \rightarrow Services} L_s$

4. IMPLEMENTATION

4.1. Genetic Operations

4.1.1. Population

The algorithm maintains a population of individuals, each of which represents a service deployment configuration for each user category and encodes it as genes. As the generations proceed, the genes evolve and optimize the individuals in generations by repeatedly applying genetic operations to them.

Currently, it is assumed that three user categories: Category 1, 2 and 3. Figure 6 shows an example individual. An individual consists of three sets of genes, each of which represents a deployment configuration for each user category. A deployment configuration consists of deployment plans and service instances. An example in Figure 6 assumes that four types of deployment plans are available and three services are defined in a workflow. A deployment plan is encoded as a set of four genes; the first gene indicates the type of the deployment plan (i.e., 0 to 3 represents the index of the type), and the second to fourth genes indicate whether an instance of a certain service is deployed on it, i.e., 1 indicates that an instance is deployed. Therefore, the first four genes in the example, i.e., 2011, represent a deployment plan of the third type that instances of the second and third services are deployed on. Since a deployment configuration can have arbitrary number of deployment plans, the number of genes varies depending on the number of deployment plans.

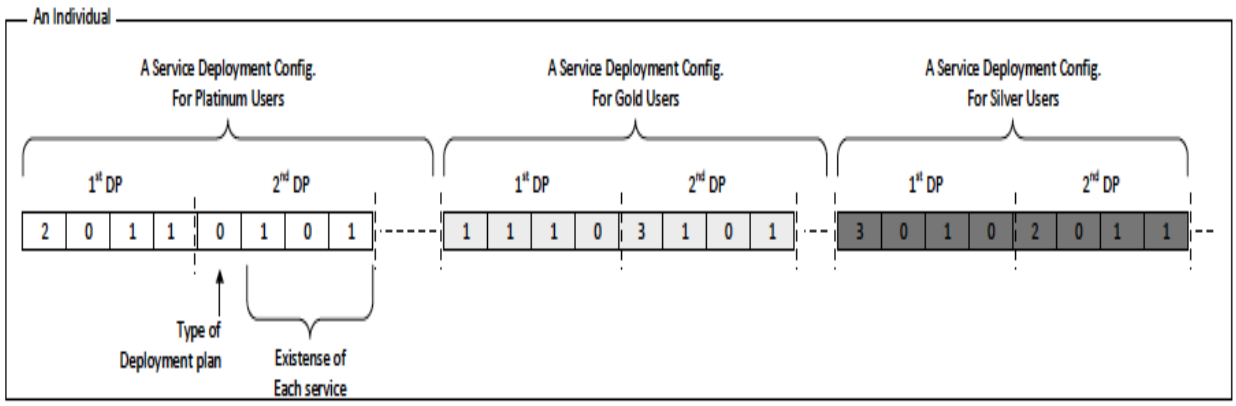

Figure 6. Example of an Individual

4.1.2. Fitness Function

The fitness function is designed to seek individuals that satisfy given SLAs and minimize the end-to-end latency as QoS objective. In order to fulfill these requirements, the fitness function estimates the distribution of service instances' latency as explained in Section 2. Table 1 describes the aggregation functions used when the services are connected in sequence or in parallel in a workflow.

4.1.3. Selection

Genetic Algorithms use a selection mechanism to select individuals from the population to insert into a mating pool. Individuals from the mating pool are used to generate new offspring, with the resulting offspring forming the basis of the next generation. As the individuals in the mating pool are the ones whose genes are inherited by the next generation, it is desirable that the mating pool be comprised of "good" individuals. A selection mechanism in GAs is simply a process that favors the selection of better individuals in the population for the mating pool. For this algorithm, a binary tournament as the selection mechanism is used, where two individuals are randomly selected by selecting the individual with the higher fitness.

Table 2. Parameters used in the simulations

G(max)	Maximum number of Generations
U	Population Size
Q(g)	A set of offspring generated at g-th generation
crossover(p1, p2)	A function returning two individuals created by one point crossover between p1 and p2
mutation(p)	A function randomly changing p's values with 1/n mutation rate
randomSelection(P)	A function returning two randomly selected individuals from P
fitSelection(p1, p2)	A function returning either p1 or p2 that has a higher fitness value
P(g)	A Set of individuals at the g-th generation

4.1.4. Crossover

A one-point crossover operation is performed on genes for each user category. When it performs a crossover on two individuals, the crossover operation first picks a set of genes for Category 1 users from both two individuals, selects a crossover point on each gene and performs a crossover. A crossover point is randomly selected from points dividing deployment plans. For example, in Figure 7, a crossover point must be between 4i -th and 4i+ 1-th genes, e.g., 4th and

5th or 8th and 9th genes, since a deployment plan is encoded as a set of four genes, crossover operation performs crossover on genes for gold users and silver users as well.

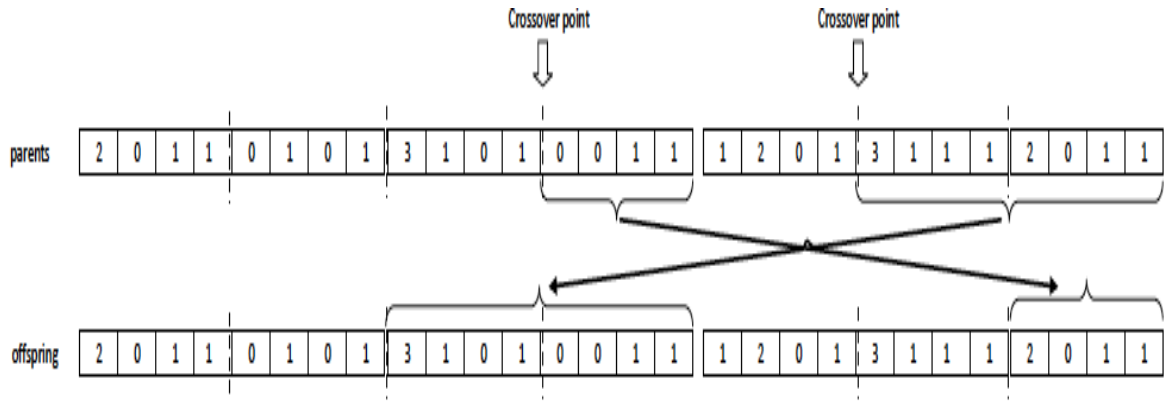

Figure 7. Crossover operation

4.1.5. Mutation

The mutation operation is designed to change the value of genes and the number of genes in an individual. First, in order to provide an opportunity to add a new deployment plan to a deployment configuration, the mutation operation adds an empty deployment plan, i.e., a deployment plan that no service instances run on, before mutating genes (the type of a deployment plan is randomly selected). Mutation occurs on genes with the mutation rate of 1=n where n is the number of genes in an individual. When a mutation occurs on a gene used for specifying the type of a deployment plan, its value is randomly altered to represent another type of deployment plan. When a mutation occurs on a gene used for indicating the existence of a service instance, its value is changed from zero to one or one to zero, i.e., non-existent to existent or existent to non-existent. Therefore, mutation may turn a newly added empty deployment plan into non-empty and may turn existing non-empty deployment plans into empty. After that, the mutation operation examines each deployment plan and removes empty deployment plans from a

deployment configuration. This way, the number of genes (the number of deployment plans) in an individual may change.

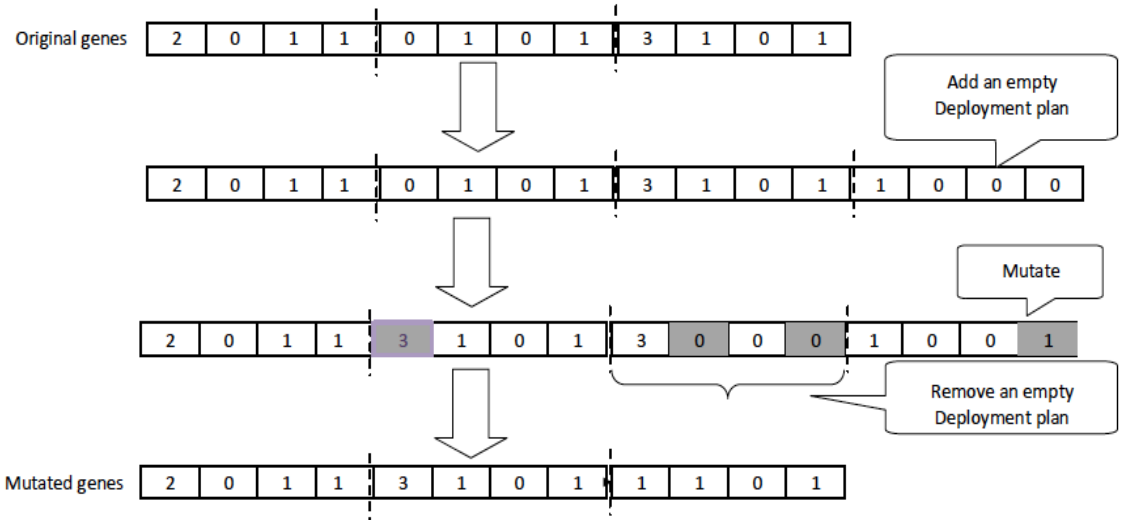

Figure 8. Mutation operation

4.1.6. Algorithm

The optimization process starts with randomly generated u individuals of population p^g at generation g. Two parents p_α and p_β are randomly selected via binary tournament and they reproduce two offspring by performing a one-point crossover operation. The offspring individuals are mutated. This process is repeated until the number of offspring population reaches u individuals. Then top u individuals are selected based on their fitness values for the next generation. This process continues until the maximum number of generations is reached. Table 2 show the parameters used in the algorithm.

Step 0: Start

Step 1: $g \leftarrow 0$

Step 2: $P^0 \leftarrow$ Population (A randomly generated u individuals)

Step 3: Repeat Until g = g_{max} (maximum number of generations)

Step 4: Assign Fitness Values to Individuals (P^g)

Step 5: $Q^g \leftarrow \phi$

Step 6: Repeat Until $|Q^g| == u$

Step 7: $p_a, p_b \leftarrow P^g$ (Randomly Selected Individuals via binary tournament)

Step 8: $p_\alpha \leftarrow p_a \; or \; p_b$ (Individual Selected with higher fitness value)

Step 9: $p_a, p_b \leftarrow P^g$ (Randomly Selected Individuals via binary tournament)

Step 10: $p_\beta \leftarrow p_a \; or \; p_b$ (Individual Selected with higher fitness value)

Step 11: $q_a, q_b \leftarrow crossover \; (p_\alpha, p_\beta)$ (A one point crossover operation is performed)

Step 12: $q_a \leftarrow mutation \; (q_a)$ (Mutation operation is performed)

Step 13: Add q_a to Q^g if Q^g does not contain q_a

Step 14: $q_b \leftarrow mutation \; (q_b)$ (Mutation operation is performed)

Step 15: Add q_b to Q^g if Q^g does not contain q_b

Step 16: Go to Step 6

Step 17: Assign Fitness Values to Individuals ($P^g \cup Q^g$)

Step 18: $P^{g+1} \leftarrow (P^g \cup Q^g)$ (Select top u Individuals in terms of Fitness value)

Step 19: $g \leftarrow g + 1$

Step 20: Go to Step 3

Step 21: End

5. SIMULATION AND RESULTS

5.1. Simulations

The algorithm is evaluated through a simulation study. This simulation study simulates different workflows. When running on a deployment plan with one 1.0 GHz CPU, each service instance possess different requests per second. For example for Figure 1, Service 1, 2, 3 and 4 process 25, 23, 21, and 19 requests/second, respectively. Table 3 & 4 shows the SLAs used in this simulation study. The simulation study assumes 25 category 1 users, 90 of category 2 users, and 750 of category 3 users access to the application on the average and allows users in each category to send 2, 1 and 0.2 requests per second. Therefore, the required throughput of category 1, 2 and 3 is 50, 90 and 150 requests per second, respectively. The workflow used in the simulation 1 is Service 1 and Service 2, 3 in parallel, and Service 4 in sequence and the workflow for Simulation 2 is Service 1, Service 2, 3, 4 in parallel, Service 5, 6 in parallel, and Service 7 in sequence. The users have the average latency of 0.40, 0.45, and 0.45 for category 1, 2 and 3 users, respectively for Simulation 1, and the users have the frequent latency of 0.2, 0.35 for category 1 and category 2 users, and the latency is not available for category 3 users for Simulation 2. There is a limit on the total costs incurred by all of three user categories, which is $1000. The Genetic algorithm uses the population size of 100 and the maximum generation of 150.

Table 3 shows the SLAs used in the first simulation study, and Table 4 shows the SLAs used in the second simulation study, and Table 5 shows the deployment plans used in both simulations.

Table 3. SLA used in the Simulation1

SLAs				
User Category	Throughput (Req/Sec)	Latency (Sec) (Average)	Cost ($)	Total Cost ($)
User Category 1	50	0.55	N/A	1,000
User Category 2	90	0.45	N/A	
User Category 3	150	0.58	N/A	

Table 4. SLA used in the Simulation 2

SLAs				
User Category	Throughput (Req/Sec)	Latency (Sec) (Frequent)	Cost ($)	Total Cost ($)
User Category 1	50	0.20	N/A	1,000
User Category 2	90	0.35	N/A	
User Category 3	150	N/A	N/A	

5.2. Results

Figure 9, 10 and 11 shows what the mean and minimum latency of Simulation1 yield for Category 1,2 and 3 users where the individuals successfully evolve and satisfy SLA for all the user categories. Figure 12, 13 and 14 show the mean and minimum latency of Simulation 2. Results are obtained through multiple runs and satisfy the given SLAs.

Table 5. Deployment plans

Name	CPU Core Speed (GHz)	Number of Cores	Cost ($)
High	1.0	4	50
Mid	1.2	2	30
Low	1.5	1	10

Simulation 1 starts with the workflow of {s1, {s2, s3}, s4} which posses mean unit service rates of {25, 23, 21, 19}. The throughput is 50, 90 and 150 req/sec user category 1, 2 and 3, respectively, and the total cost limit is $1,000.

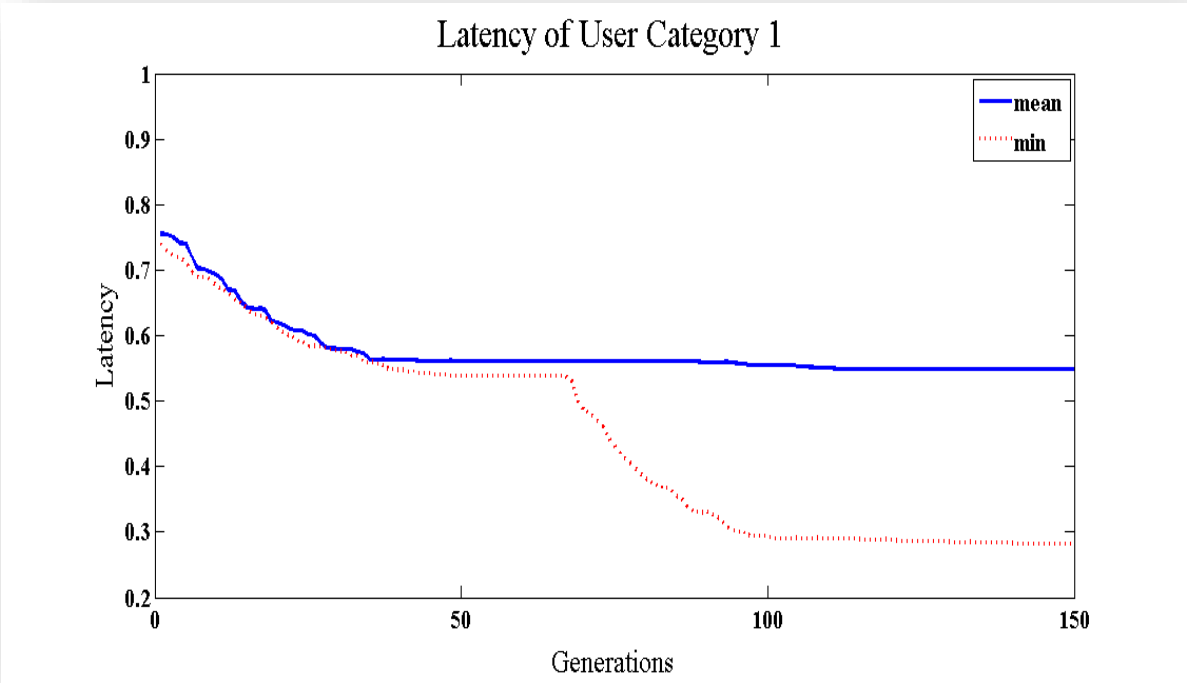

Figure 9. Mean and Minimum Latency of Simulation 1 for User Category 1

Figure 9 shows the mean and minimum latency for user category 1, which satisfies the average latency given in the SLA.

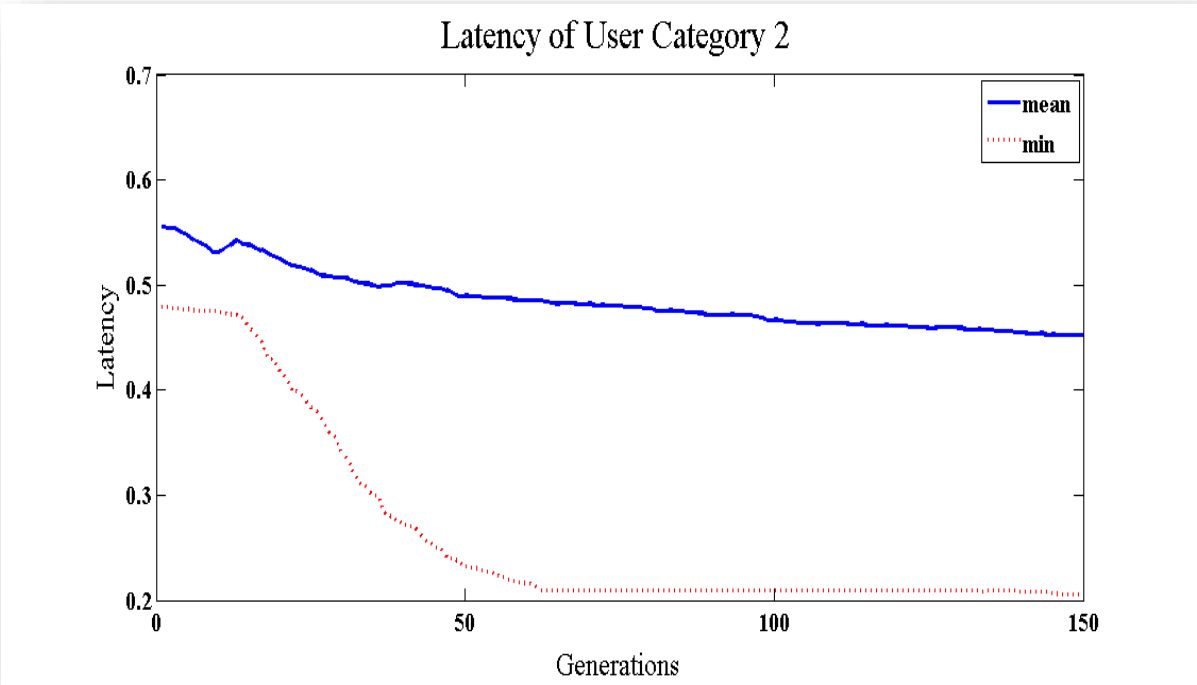

Figure 10. Mean and Minimum Latency of Simulation 1 for User Category 2

Figure 10 shows the mean and minimum latency for user category 2, which satisfies the average latency given in the SLA.

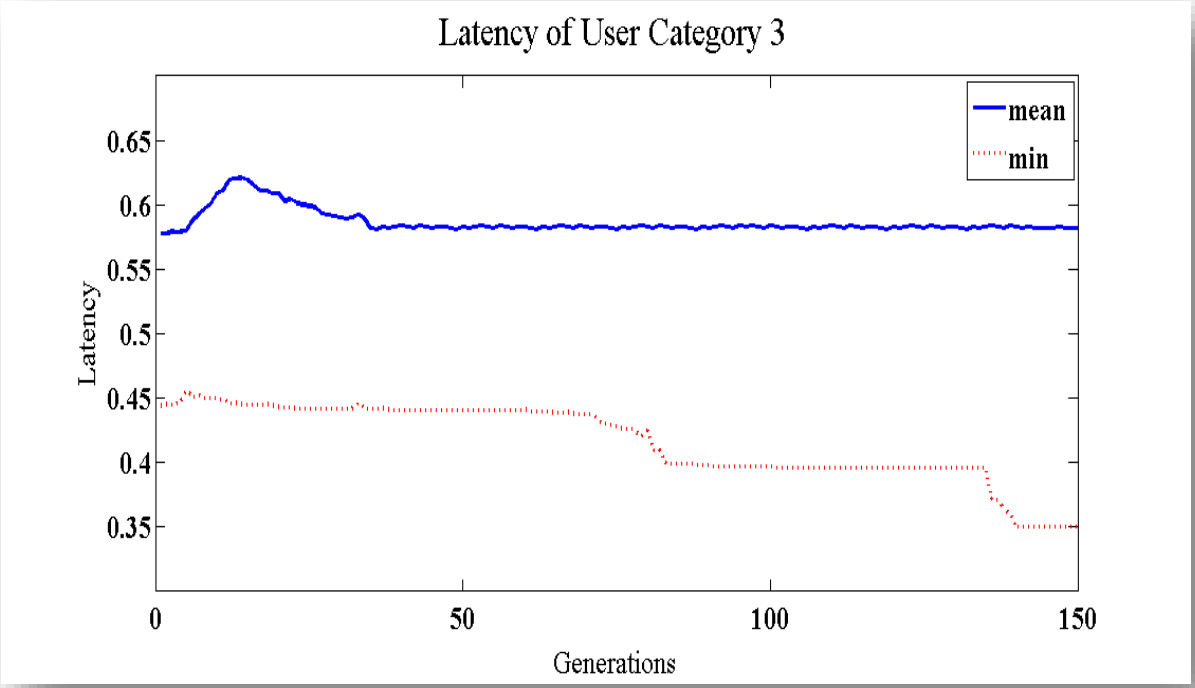

Figure 11. Mean and Minimum Latency of Simulation 1 for User Category 3

Figure 11 shows the mean and minimum latency for user category 3, which satisfies the average latency given in the SLA.

Simulation 2 starts with the workflow of {s1, {s2, s3, s4}, {s5, s6}, s7} which possess mean unit service rates of {29, 24, 22, 20, 21, 22, 27}. The throughput is 50 req/sec and the total cost limit is $1,000.

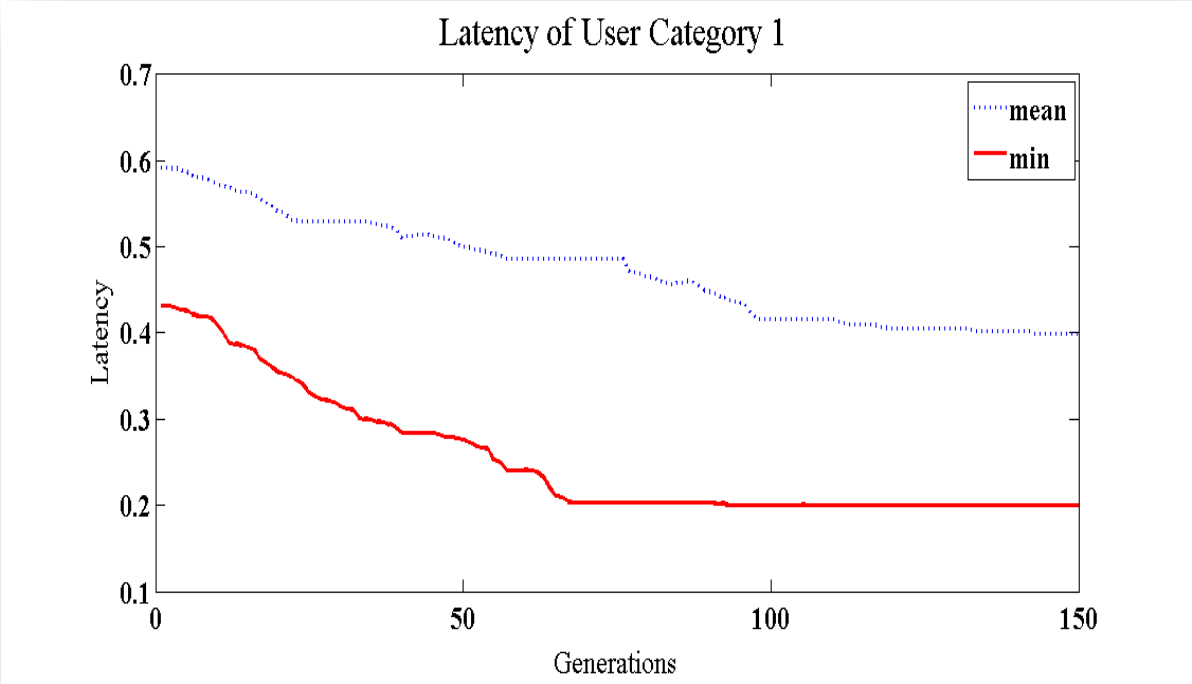

Figure 12. Mean and Minimum Latency of Simulation 2 for User Category 1

Figure 12 shows the mean and minimum latency for User Category 1 for simulation 2. The frequent latency is minimized to 0.2 as its optimal solution as the generations evolve and satisfying the given SLA.

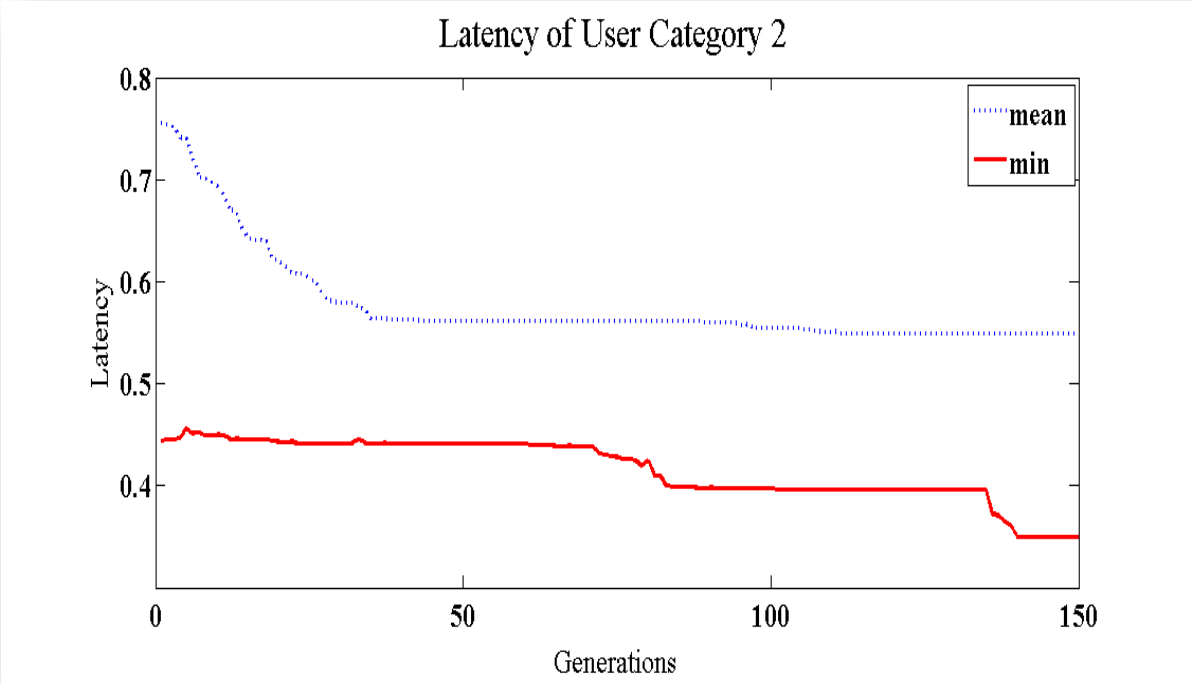

Figure 13. Mean and Minimum Latency of Simulation 2 for User Category 2

Figure 13 shows the mean and minimum latency for user category 2 for simulation 2, which satisfies the frequent latency given in the SLA, in which latency is minimized from 0.65 to 0.35 as the generations evolve finding the optimal solution in the search space.

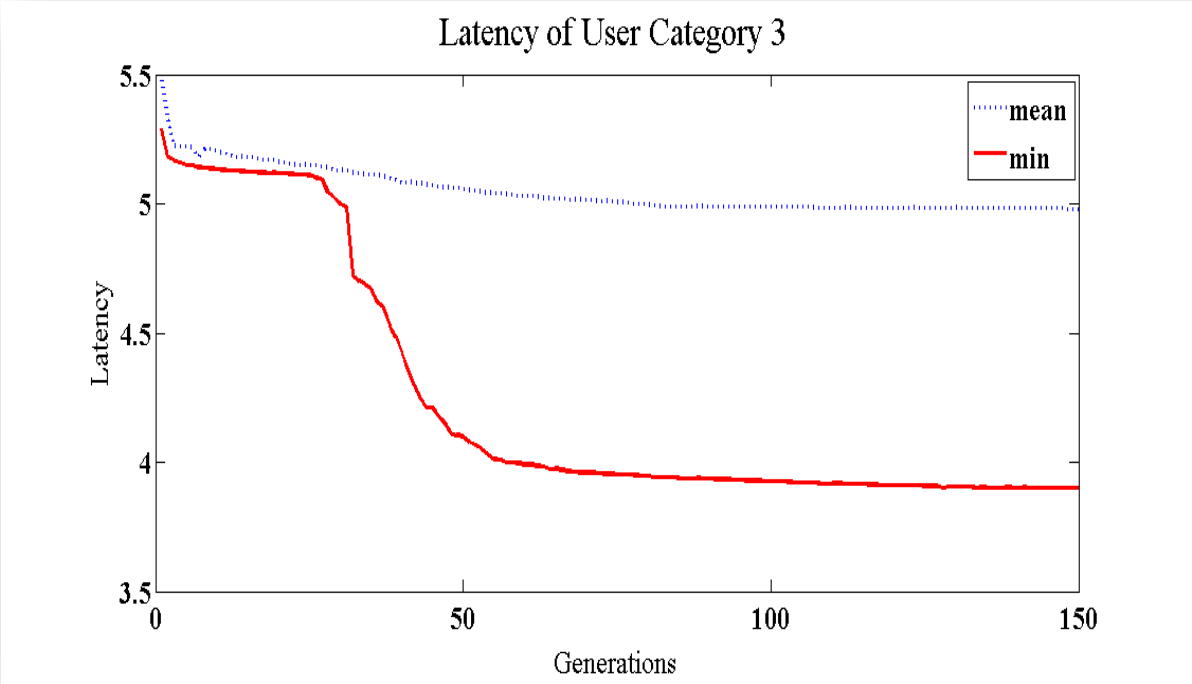

Figure 14. Mean and Minimum Latency of Simulation 2 for User Category 3

Although, user category 3 does not have any latency requirement in the SLA, the algorithm lowers the latency. Figure 14 shows the minimized latency for the user category 3 as the latency is optimized to its minimum.

6. CONCLUSION

6.1. Summary

In this paper, a genetic algorithm, that was previously proposed [11], is implemented to solve the SLA aware Service Deployment Optimizer (SSDO) problem in Cloud computing environments. To solve the large-scale SSDO problem, it is required to use the heuristic methods like Genetic Algorithms [23-25], as they scale better than linear programming methods. By leveraging queuing theory applied to the SSDO problem [31-35], the paper has implemented a genetic algorithm to optimize the latency of the service instances deployment in different workflows. Simulation results estimate the performance of the application by optimizing the deployment configurations and allowing the user to describe the latency in a probabilistic manner.

6.2. Future Improvements

The paper takes into consideration only the total cost as the objective but does not consider the individual cost for each user category, and the throughput is assumed to be the throughput as described in the SLA. Future work may include the cost and throughput as end-to-end QoS objectives to minimize the cost for each user category and maximize the throughput of the deployment configurations.

7. REFERENCES

[1] B. Furht, A. Escalante (eds.), "Handbook of Cloud Computing", DOI 10.1007/978-1-4419-6524-0_1, © Springer Science + Business Media, LLC 2010.

[2] Martin Randles, David Lamb, A. Taleb-Bendiab, "A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing", 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops, pp. 551-556.

[3] T. Gopalakrishnan Nair, M. Vaidehi, K. Rashmi, V. Suma, "An Enhanced Scheduling Strategy to Accelerate the Business performance of the Cloud System", Proc. InConINDIA 2012, AISC 132, pp. 461-468, © SpringerVerlag Berlin Heidelberg 2012.

[4] B. Rimal, E. Choi, I. Lumb, "A taxonomy and survey of cloud computing systems," in Proc. IEEE Fifth International Joint Conference on INC, IMS and IDC, 2009, pp. 44–51.

[5] Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/

[6] Google App Engine. http://code.google.com/appengine/

[7] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms. In ACM International Conference on Genetic and Evolutionary Computation Conference. ACM Press, June 2005.

[8] M. C. Jaeger and G. Mühl. QoS-based Selection of Services: The Implementation of a Genetic Algorithm. In Workshop on Service-Oriented Architectures and Service- Oriented Computing, March 2007.

[9] Penta MD, Esposito R, Villani ML, Codato R, Colombo M, Nitto ED. WS Binder: A Framework to Enable Dynamic Binding of Composite Web Services. International Workshop on Service-Oriented Software Engineering, Shanghai, China, 2006; 74–80.

[10] Berbner R, Spahn M, Repp N, Heckmann O, Steinmetz R. Heuristics for QoS-aware Web Service Composition. International Conference on Web Services, Chicago, IL, 2006; 72–82

[11] Queuing Theoretic and Evolutionary Deployment Optimization with Probabilistic SLAs for Service Oriented Clouds, 2009 IEEE Congress on Services. SERVICES 2009, pp. 661-669, doi:10.1109/SERVICES-I.2009.59.

[12] Harman M, Clark J. Metrics Are Fitness Functions Too. IEEE International Symposium on Software Metrics, Chicago, IL, 2004; 58–69.

[13] Coello CAC. Recent Trends in Evolutionary Multiobjective Optimization. Evolutionary Multiobjective Optimization, Jain L, Wu X, Abraham A, Jain L, Goldberg R (eds.). Springer, 2005; 7–32.

[14] Harman M. The Current State and Future of Search Based Software Engineering. ACM/IEEE International Conference on Software Engineering, Minneapolis, MN, 2007; 342 357.

[15] Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London April 2009.

[16] T. Yu, Y. Zhang, and K. J. Lin. E_cient Algorithms for Web Services Selection with end-to-end QoS Constraints. ACM Transactions on the Web, 1(1), Dec 2007.

[17] D. Ardagna and B. Pernici. Adaptive Service Composition in Flexible Processes. IEEE Transactions on Software Engineering, 33(6):369–384, June 2007.

[18] Anselmi J, Ardagna D, Cremonesi P. A QoS-based Selection Approach of Autonomic Grid Services. ACM Workshop on Service-Oriented Computing Performance, Monterey, CA, 2007

[19] Cardellini V, Casalicchio E, Grassi V, Presti FL. Flow-based Service Selection for Web Service Composition Supporting Multiple QoS Classes. IEEE International Conference on Web Services, Salt Lake City, UT, 2007; 743–750.

[20] Qu Y, Lin C, Wang YZ, Shan Z. QoS-Aware Composite Service Selection in Grids. International Conference on Grid and Cooperative Computing, Hunan, China, 2006; 458–465.

[21] Zeng L, Benatallah B, Ngu A, DumasM, Kalagnanam J, Chang H. QoS-Aware Middleware for Web Services Composition. IEEE Transactions on Software Engineering May 2004

[22] Srinivas N, Deb K. Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. Evolutionary Computation 1994; **2**(3):221–248.

[23] Deb K, Pratap A, Agarwal S, Meyarivan T. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE Transactions Evolutionary Computation 2002; **6**(2):182–197.

[24] Fieldsend J, Everson RM, Singh S. Using Unconstrained Elite Archives for Multiobjective Optimization. IEEE Transactions on Evolutionalry Computing 2003; **7**(3):305–323

[25] C. Stewart, T. Kelly, and A. Zhang. Exploiting Nonstationarity for Performance Prediction. In ACM European Conference on Computer Systems, Mar 2007.

[26] Y. Liu, I. Gorton, and L. Zhu. Performance Prediction of Service-Oriented Applications based on an Enterprise Service Bus. In IEEE Int'l Computer Software and Applications Conference, July 2007.

[27] W. C. Chan and Y. B. Lin. Waiting time distribution for the M/M/m queue. IEEE Proceedings Communications, 2003.

[28] P. QoS and S. C. f Transaction-Based Web Services Orchestrations. Model Driven Development for Business Performance Management. IEEE Transactions on Services Computing, 2006.

[29] N. Srinivas and K. Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. Evolutionary Computation, 2(3):221–248, 1994.

[30] H. Ishibuchi, N. Tsukamoto, Y. Hitotsuyanagi, and Y. Nojima. Effectiveness of scalability improvement attempts on the performance of NSGA-II for many-objective problems. In ACM International Conference on Genetic and Evolutionary Computation Conference, July 2008.

[31] H. Wada, P. Champrasert, J. Suzuki, and K. Oba. Multiobjective Optimization of SLA-aware Service Composition. In IEEE Workshop on Methodologies for Non-functional Properties in Services Computing, July 2008.

[32] D. B. Claro, P. Albers, and J. Hao. Selecting Web Services for Optimal Composition. In IEEE International Workshop on Semantic and Dynamic Web Processes, July 2005.

[33] H. A. Taboada, J. F. Espiritu, and D.W. Coit. MOMS-GA: A Multi-Objective Multi-State Genetic Algorithm for System Reliability Optimization Design Problems. IEEE Transactions on Reliability, March 2008.

[34] T. Unger, F. Leymann, S. Mauchart, and T. Scheibler. Aggregation of Service Level Agreements in the Context of Business Processes. In IEEE Int'l Enterprise Distributed Object Computing Conference, Sept 2008.

[35] Evolutionary deployment optimization for service-oriented clouds by Hiroshi Wada, Junichi Suzuki, Yuji Yamano and Katsuya, Article published online: DOI: 10.1002/spe.1032

.