**A NETWORK SCIENCE BASED APPROACH FOR OPTIMAL MICROSERVICE GOVERNANCE**

L. S. Jayasinghe

IT17012966

B.Sc. (Hons) Degree in Information Technology

Specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

September 2020

**A NETWORK SCIENCE BASED APPROACH FOR OPTIMAL MICROSERVICE GOVERNANCE**

L.S. Jayasinghe

IT17012966

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of

Science specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

September 2020

**Declaration**

I declare that this is our own work and this proposal does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of our knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

**Signature:**

|  |  |  |
| --- | --- | --- |
| **IT17012966** | **L.S Jayasinghe** |  |

The above candidates are carrying out research for the undergraduate Dissertation under my supervision.

**Signature of the Supervisor: ……………………………………**

**Date: …………………………….**

**Abstract**

Even though deployment of microservice applications could be performed through the utilization of orchestration tools such as Kubernetes, these tools are not intelligent enough to deploy services optimally and only follow what the developer defines in the Kubernetes YAML deployment file. Moreover, even though Kubernetes provides autoscaling tools such as Horizontal Pod Autoscaler, these autoscaling tools make use of a limited number of infrastructure-level metrics such as CPU, which are not enough to achieve optimal performance of a microservice application.

This research pertains to the development of an optimal deployment algorithm through the inclusion of a multitude of metrics based on predicted load-based dependency measures, node latency measures, metrics such as CPU, memory metrics as well as even centrality measures performed on microservice co-dependency networks to optimize Kubernetes deployments. Through the utilization of these metrics, the research conducted guarantees higher performance and higher availability of the microservice deployments.

Lastly, a key fact to note in this regard is the fact that the developed optimal deployment algorithm consumes metrics which are predicted values that are provided by the Autoscaler component, so developers could effectively see future deployment plans as well as scale up or down their cluster according to the optimized deployment plan. Therefore, through the application of the developed optimization algorithm, developers could effectively utilize cluster resources, while minimizing costs. Hence, the developed optimization model is more productive and efficient for any industry-based microservice application.

**Keywords:** Auto-scaling, Kubernetes, Machine Learning, Microservices, Time Series

**Acknowledgement**

I would like to thank my supervisor, Dr. Dharshana Kasthurirathna for the guidance and motivation provided in order to make this research a success. I would also like to thank the Department of Software Engineering of the Sri Lanka Institute of Information Technology as well as the CDAP lecturers and staff for the guidance and support provided.

**Table of Contents**

[**Declaration** i](#_Toc51622419)

[**Abstract** ii](#_Toc51622420)

[**Acknowledgement** iii](#_Toc51622421)

[**Table of Contents** iv](#_Toc51622422)

[**List of Figures** vi](#_Toc51622423)

[**List of Tables** vii](#_Toc51622424)

[**List of Abbreviations** viii](#_Toc51622425)

[**1.0** **INTRODUCTION** 1](#_Toc51622426)

[**1.1 Background and Literature** 4](#_Toc51622427)

[**1.2 Research Gap** 6](#_Toc51622428)

[**2.0** **Research Problem** 7](#_Toc51622429)

[**3.0** **OBJECTIVES** 8](#_Toc51622430)

[**3.1 Main Objective** 8](#_Toc51622431)

[**3.2 Specific Objectives** 8](#_Toc51622432)

[**4.0** **METHODOLOGY** 13](#_Toc51622433)

[**4.1 System Analysis** 13](#_Toc51622434)

[**4.2 System Development and Implementation** 17](#_Toc51622435)

[**4.2.1 HPA Algorithm Component** 17](#_Toc51622436)

[**4.2.2 Objective Function** 18](#_Toc51622437)

[**4.2.3 Latency Objective** 19](#_Toc51622438)

[**4.2.4 Availability Objective** 21](#_Toc51622439)

[**4.2.5 Ratio Objective** 22](#_Toc51622440)

[**4.2.6 Scale and Cost Objective** 23](#_Toc51622441)

[**4.2.7 Resource constraint and minimal maximum instances constraint** 24](#_Toc51622442)

[**4.2.8 NSGA II Implementation and Execution Order** 24](#_Toc51622443)

[**4.3 Commercialization** 29](#_Toc51622444)

[**5.0** **RESULTS AND DISCUSSION** 30](#_Toc51622445)

[**5.1 Algorithm selection** 30](#_Toc51622446)

[**5.2 Results** 32](#_Toc51622447)

[**5.2.1 Best Performance Deployment Strategy** 34](#_Toc51622448)

[**5.2.2 Best Availability Deployment Strategy** 36](#_Toc51622449)

[**5.2.3 Cost-effective Deployment Strategy** 38](#_Toc51622450)

[**5.2.4 Optimal Deployment Strategy** 40](#_Toc51622451)

[**5.3 Research Findings and Discussion** 41](#_Toc51622452)

[**6.0 CONCLUSION** 42](#_Toc51622453)

[**References** 43](#_Toc51622454)

[**Appendix** 46](#_Toc51622455)

[**Appendix A: Implementation of NSGA II Algorithm** 46](#_Toc51622456)

[**Appendix B: Implementation of Fitness function** 52](#_Toc51622457)

[**Appendix C: Implementation of Fast Non-Dominate Sorting Algorithm** 55](#_Toc51622458)

[**Appendix D: Implementation of Constraints functions** 57](#_Toc51622459)

[**Appendix E: Implementation of Pandas Query Class** 62](#_Toc51622460)

[**Appendix F: Implementation of HPA Component** 63](#_Toc51622461)

**List of Figures**

|  |  |  |
| --- | --- | --- |
|  |  | Page |
|  |  |  |
| Figure 1.1 | MAPE control loop | 4 |
| Figure 1.1 | Network Latency Between Multiple Regions | 3 |
| Figure 4.1 | System Inputs and Outputs | 13 |
| Figure 4.2 | Optimization Algorithm Methodology | 17 |
| Figure 4.3 | Pod Level Dependency Map with Dependency Links | 20 |
| Figure 4.4 | Generalized Logistic Function | 22 |
| Figure 4.5 | Generalized Logistic Function Graph | 22 |
| Figure 4.6 | Chromosome Design | 25 |
| Figure 5.1 | Latency Fitness Graph | 30 |
| Figure 5.2 | Mean Latency Fitness Graph | 31 |
| Figure 5.3 | Mean Availability Fitness Graph | 31 |
| Figure 5.4 | Structure of sample JSON input | 32 |
| Figure 5.5 | Existing Performance Deployment View | 33 |
| Figure 5.6 | Existing Performance instances Table | 33 |
| Figure 5.7 | Best Performance Deployment View | 34 |
| Figure 5.8 | Best Performance Instances Table | 35 |
| Figure 5.9 | Best Performance Metrics Improvement Table | 35 |
| Figure 5.10 | Best Availability Deployment View | 36 |
| Figure 5.11 | Best Availability Instances Table | 37 |
| Figure 5.12 | Best Availability Metric Table | 37 |
| Figure 5.13 | Cost-Effective Deployment View | 38 |
| Figure 5.14 | Cost-Effective Instances Table | 39 |
| Figure 5.15 | Cost-Effective Metric Improvement Table | 39 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**List of Tables**

|  |  |  |
| --- | --- | --- |
|  |  | Page |
| Table 4.1 | Tools and technology | 18 |
| Table 4.2 | Latency Objectives Calculation | 20 |
| Table 4.3 | Availability Objectives Calculation | 21 |
| Table 4.4 | Ratio Objectives Calculations | 23 |
| Table 4.5 | Final Solution Option Characteristics | 26 |
| Table 5.1 | Metric Measurement Units | 32 |
| Table 5.2 | NSGA II Algorithm Initialization Parameters | 33 |

**List of Abbreviations**

|  |  |
| --- | --- |
| Abbreviation | Description |
| API | Application Programming Interface |
| AWS | Amazon Web Service |
| HPA | Horizontal Pod Autoscaler |
| JSON | JavaScript Object Notation |
| VM | Virtual Machine |
|  |  |

1. **INTRODUCTION**

The introduction of the microservice architecture in the early 2010s brought about a new era in the development of software applications, with many leading organizations such as Facebook, Netflix, and Amazon, etc. utilizing microservice architecture towards the development of software applications. This was mainly due to the fact that through adopting microservice architecture, these organizations could ensure high availability, maintainability, and performance from the deployed microservices.

However, even though microservices brought about a multitude of advantages as discussed above, they are not devoid of inherent challenges. In this regard, a key challenge faced is the presence of a vast amount of configuration issues related to microservices during deployment. Therefore, in response to the presence of such issues, containerization tools such as Docker was introduced in 2013 [1].

Docker provided and effective solution to deploy microservice applications through efficiently packaging microservices by encompassing all the required libraries and dependencies needed during runtime. Hence, through the utilization of Docker, organizations could effectively deploy software applications anywhere without any hesitation. Nevertheless, even though Docker was able to solve some issues pertaining to microservice deployment, all issues prevalent in microservice deployments were not resolved.

Generally, microservice related applications use more than 100 microservices. Even organizations such as Netflix use about 700 microservices [2]. Consequently, a typical microservice application might require a vast array of microservices. However, as a microservice application becomes increasingly large it becomes difficult to manage these microservices. To solve this problem, container-orchestration tools such as Kubernetes and Docker Swarm [3] were introduced. These container orchestration tools are basically used to increasing availability by increasing the number of instances as well as management services.

In this regard, availability is one of the most prominent non - functional requirements in the microservices model since failure to ensure sufficient availability could cause most services to fail. To increase availability, we can use multiple instances per microservices. It is called replication in the microservice model and through replication, when one instance fails, there is another instance made available in its place. However, replication could also prove ineffective if all the instances are deployed on a single node. This is due to the fact that when such a node goes down, no instances are made available to execute remaining tasks [4]. Hence, in order to ensure optimum availability, it is imperative to deploy multiple instances ideally on multiple nodes co-located in multiple data centers.

Nevertheless, although adopting the microservice model is a great solution for gaining availability in an application, it dramatically reduces application performance. In fact, as per the statement,

*“We observed a significant overhead due to the microservice architecture; the performance of the microservice model can be 79.1% lower than the monolithic model on the same hardware configuration. The microservice model spent much more time in runtime libraries to process one client request than the monolithic model by 4.22 times on a Node.js application server and by 2.69 times on a Java EE application server.”* [5]

stated by IBM researchers, we can identify the performance as a main issue in the microservice architecture.

Moreover, if we consider the performance of the web application, we can measure performance by evaluating the response time. In this regard, if the average response time is lower across the entire application, it can be identified as a high application that ensures high performance. Although factors such as computational time, bandwidth, and network latency directly affect response time, a bigger portion of pertaining to increased response times in an application is due to network latency. This metric is calculated through the evaluation of the Round-Trip Time (RTT) [6], typically measured in milliseconds.

Even if the network has a high bandwidth rate, high network latency time takes a long time to get the response back. This effect can be experienced in the multiplayer gaming networks because a lower ping (RTT) may prove to be highly advantageous to players.

In the microservice model, microservices intercommunicate via API calls. Moreover, these services are deployed over multiple nodes, hence even though one request is sent it goes through multiple nodes. In a cluster network, nodes are interconnected with network links, when the request goes through these links, network latencies are added for each link. Also, if the nodes are located very far, it increases the network latency thereby reducing the overall performance of the application [7].

A screenshot of text

Description automatically generated

Figure 1.1 Network Latency Between Multiple Regions.

Currently, there is a multitude of tools available for automated deployment. Even tools such as Kubernetes is capable of performing automated deployments. However, no tools are available for generating optimal deployment strategy plans for microservice applications, as evident through this research. This strategic plan describes how microservices are deployed over the node network. It is guaranteed that the microservice application has optimal performance and availability.

**1.1 Background and Literature**

In science, engineering, IT, or economics, optimization is the process of finding the best solution among all feasible solutions. Optimization problems typically consist of maximization or minimization objective as well as input values. Moreover, it can also generate a globalized or localized solution according to the given criteria[9].

There have been multiple approaches to solve optimization problems. The most popular and commonly used approach is through the utilization of optimization algorithms such as the Linear Programming (LP) algorithm which was primarily developed by a Soviet mathematician named Leonoid Kantorovich during the World War Ⅱ to reduce the cost of the army with a plan for optimized expenditure while increasing losses imposed on the enemy. Later, in 1975 Leonoid Kantorovich shared the Noble prize in economics for the development of the LP algorithm [10].

The microservice optimal deployment is an NP-hard [11] problem, and it requires that typical approaches to optimization need to evaluate every possible solution. Therefore, if an approach to the optimization uses exact algorithms, it takes a long time to reach an optimal solution. The NSGA Ⅱ algorithm is a heuristic multi-objective optimization algorithm based on the regular genetic algorithm. The NSGA Ⅱ algorithm is deemed a multi-objective optimization algorithm since it could address complex optimization problems, with multiple objectives in contrast to regular optimization algorithms. These objectives are measured by using different units, so when the solutions are being generated, some of the generated solutions get the same rank, because some solutions are not dominated by other solutions. Hence to effectively rank such solutions, the NSGA Ⅱ algorithm uses non-dominating sorting techniques and crowding distance measurements to select the best possible solution [12].

Moreover, these kinds of heuristic algorithms are not going to evaluate every possible solution. It only generates random solutions at the beginning of the algorithm. After that, it uses a converging mechanism to reach the optimal solution. Also, these kinds of algorithms cannot guarantee to generate one best solution. Hence the NSGA Ⅱ algorithm can be stated as one of the most appropriate optimization algorithms to solve the above discussed microservice deployment problem [12].

Although the deployment process of containerized microservices in Kubernetes allows functionalities such as scheduling deployments and autoscaling, the capability to determine the optimal placement of microservices in the deployment of containers does not exist [13]. In fact, Kubernetes is unable to determine the optimal placement for the deployment of containers unless explicitly configured. Furthermore, deploying an application without consideration of dependencies may result in low application performance. Besides, tools such as HPA, which perform real-time autoscaling, also do not consider the effect of dependent services in the determination of the optimum number of instances [14].

**1.2 Research Gap**

There have been few pieces of research related to the optimal deployment of microservices, which gives prominence to optimal microservice placement. The research conducted by the authors of [15] proposes a solution for determining the optimal microservice placement in microservices deployment through analysis of historical values and microservice dependencies, similar to that proposed in this research.

However, the research conducted does not take into consideration key factors such as the inter-node latency between nodes in the cluster as wells as the inclusion of measures such as resilience and centrality evaluation Moreover, although the approach suggested in the above-stated research makes use of historical data in the determination of microservice placement it does not make use of the historical data to make effective prediction mechanisms using the gathered historical data to adapt to changes that may affect future placement decisions.

1. **Research Problem**

All typical application systems have pre-defined non- functional requirements such as availability, adaptability, durability, interoperability, reliability, etc. In this research, we are mainly focused on performance and availability.

When microservices are deployed in the cloud cluster, we typically utilize container orchestration tools such as Kubernetes to manage and handle microservice deployments. Moreover, through utilizing tools such as Kubernetes, we can also perform automated deployments. Regardless, a key point to note concerning orchestration tools such as Kubernetes is that they are not intelligent enough to optimally deploy microservices. In this regard, tools such as Kubernetes only follow pre-configured deployment policies by users such as system administrators or DevOps Engineers irrespective of the impact on the cluster performance. Hence, when a new microservice is introduced to the cluster, it is difficult to optimize and maintain the required performance level of the cluster.

Nodes are located in multiple places in the cloud. In some scenarios, it is located in another country, called a region. In the same region, there are multiple data centers. These distances create network latency within a cluster. Even in the same data center, there is some network latency. Regardless, if the entirety of a microservice application is deployed in one virtual machine, we can annul network latency and increase application performance [6][7][8]. However, the downside of this approach is that it reduces availability.

Before microservices are deployed, users have no idea about current microservices resource consumption and cluster resource power (including all the nodes). After the application is deployed, if the resource power is not enough to execute all the requested tasks, the whole application will crash. Moreover, if resource power is larger than the required resource power, a large sum of money could be wasted.

The main target of this research is to increase overall application performance without losing availability while ensuring, each microservice in the cluster is allocated sufficient node resource power. Moreover, this model provides a future deployment view of the cluster such that DevOps engineers can get a good idea about how to manage future requirements.

1. **OBJECTIVES**

**3.1 Main Objective**

To identify key factors that lead to performance reduction in microservice deployments and come up with an optimal deployment strategy.

**3.2 Specific Objectives**

The following are the sub-objectives of conducting this research.

* To increase the efficiency of microservices deployments by applying the metrics used in network analysis, such as centrality and resilience measures, and link predictions on identified dependency measurements.
* To develop a business intelligence dashboard to evaluate performance and monitor microservice deployments.
* To identify key factors that lead to performance reduction in microservice deployments and come up with an optimal deployment strategy.
* Generate solutions based on the best performance and best availability.
* Generate deployment view of future solutions.

**3.1 Requirement Gathering**

Requirement gathering was mainly performed through performing an extensive analysis of past research conducted throughout recent years, identification and analysis of the existing systems, as well as reading through a variety of online resources.

**3.1.1 Past Research Analysis**

The past research analysis process was mainly performed through reading and analyzing a wide array of research publications published through recent years. Key topics of interest included microservice deployment optimization, microservice performance engineering, microservice governance, centrality evaluation, load prediction, and forecasting, resource prediction and optimization, resiliency analysis, and microservice monitoring. During the research analysis process, the primary focus was given in the identification of the methodology used, tools used, experiments conducted, as well as the overall findings of the research with respect to performance optimization in microservices.

**3.1.2 Identifying Existing Systems**

A thorough analysis was conducted on a variety of existing APM tools as well as other similar systems, that were available to use with the Kubernetes platform. This process was mainly done by visiting the various online sources and analyzing the available documentation and videos published. During this process, the primary focus was given in identifying the key features and drawbacks that were present in the tools analyzed. Moreover, even if there is no existing system directly related to this research. There may exist automated deployment tools and related researches that could prove useful for testing the final product.

**3.2 Feasibility Study**

**3.2.1 Technical Feasibility**

This microservice optimization model is implemented to be applied for Kubernetes related applications. Hence a thorough knowledge of the fundamentals of Kubernetes, deployment methodologies pertaining to microservice applications, as well as knowledge on containerization tools such as Docker is required.

Optimization algorithms are used in a wide variety of fields. It is an applied mathematics field that is part of data science. These optimization algorithms reach the optimal solution by maximizing or minimizing variables. Different types of optimization algorithms have different accuracy levels and performance levels. Hence, it is very important to select the optimum algorithm for producing the optimum result. Moreover, execution time also is one of the important things which is highly important pertaining to optimization algorithms, especially in NP-hard problems.

Cloud networks consist of WAN, MAN, LAN, and virtual networks. Similarly, the Kubernetes network includes node network and pod network. Research regarding the changing of pod placement while minimizing latency is another important feature of this research.

**3.2.2 Schedule Feasibility**

The proposed project should be able to be implemented within the scheduled time period of about five months, with about two months allocated for research, requirement gathering, and analysis. Finally, the proposed project should be completed within the end of 7 months, including sufficient testing.

**3.2.3 Economic Feasibility**

The cost of the proposed project should be as minimal as possible in order for it to be included and accepted in the existing APM tool market in Kubernetes. This is mainly due to the fact that most APM tools and solutions offered currently with respect to Kubernetes are often opensource.

**3.3 Requirement Analysis**

During the requirement analysis phase, key information obtained during the requirement gathering phase is analyzed. Analyzing the gathered information will prove to be of most importance to the research process, since key information regarding the potential challenges that may be faced, the potential complexity of tasks involved, as well as other key information regarding the tools used by other research teams will be easily identifiable.

Also, since the research carried out a software-based approach, by performing requirement analysis, key information regarding the schedule, technical and economic feasibility was realized and helped in aligning research goals such that the research carried out does not exceed the technical skills of the research members while maintaining the expected deadlines.

In the research paper analysis, the primary focus will be given to the analysis of the methodology and tools used, as well as the outcomes of the research conducted. This helps in improving the decision-making process in the current research by providing credible evidence that will help in deciding upon the direction in which the current research should progress by highlighting the research gaps.

Furthermore, analyzing the online resources regarding the available tools will help in the identification of the existing tools that possess similar features to what the current research aims at implementing and help in identifying the research gap by comparing the existing feature with those that are proposed. Also, by analyzing tools that could prove to be of use in the implementation of the current research, a clear idea regarding the features they possess, and how they could be integrated into the current research could be identified.

1. **METHODOLOGY**

**4.1 System Analysis**

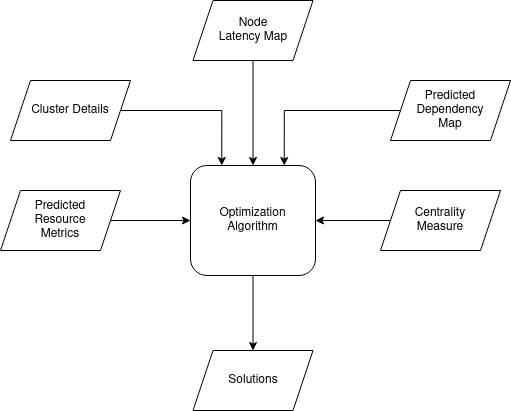


Figure 4.1 System Inputs and Outputs

As per Fig 4.1 provided above four input measures are passed into the optimization algorithm to facilitate the optimization process. These input measures are then utilized to the generation of optimized solutions pertaining to the best performance and highest availability of a given microservice deployment. In this regard, the sub-sections provided below provide and detailed description regarding each key input measure provided to the optimization algorithm

**4.1.1 Cluster Details**

Cluster details metric primarily concern metrics pertaining to the resource power of each node and resource consumption of each pod. These include metrics such as CPU, and memory metrics utilized to derive the maximum number of instances for a given cluster. For example, if any nodes in the cluster contain pods that require more resource power than the maximum resource power available, it will be considered as an invalid solution.

**4.1.2 Predicted Microservice Co-dependency Map**

The predicted microservice co-dependency map defines the predicted load-based inter-microservice link dependency measures for a given forecasted period and is retrieved via the load prediction and centrality analysis component of the developed governance model. In this regard, the predicted dependency measures are derived through performing a time series prediction process through the utilization of a developed machine learning model, deployed in the load prediction and centrality analysis component.

The dependency measures are derived from service level metrics such as the number of requests per second passing through a given microservice link, through the utilization of monitoring tools deployed in the Istio service mesh. Moreover, the process is of retrieving the predicted microservice co-dependency map can be performed as per a scheduled process to continually retrieve predicted link dependency measures for future time periods.

Moreover, the dependency measures derived from the predicted microservice co-dependency map is an essential input measure required by the optimization algorithm since it is utilized in the determination of optimal placement for each microservice in a given deployment. In this regard, through the utilization of the predicted dependency measures potential co-dependent microservices could be identified in advance and be placed on the nearest or same nodes to improve the performance of a given deployment.

**4.1.3 Predicted Resource Utilization Metrics and Centrality Measures**

These predicted resource utilization metrics and centrality measuresare primarily used for achieving availability in the cluster. Predicted resource utilization metrics primarily concerns metrics such as pod CPU utilization metrics and memory metrics and are obtained from the load prediction and centrality analysis component of the developed governance model. These predicted resource utilization metrics are essentially obtained through a time series prediction process in which resource utilization of pods in the cluster are forecasted for a given time period, similar to the process utilized in the prediction of microservice link dependency measures. This process also could be performed as per a scheduled process and could be utilized in conjunction with the HPA of Kubernetes to facilitate the determination of the optimal number of required instances of pods of a particular service along with proactive autoscaling.

However, centrality measures are used to achieve a different scenario. In this case, during initial deployment as well as throughout the first few days of deployment, when there is essentially a limited array of resource utilization metrics available to make use of, evaluated centrality measures performed on the microservice co-dependency map will be primarily utilized to deduce optimal instances of microservices in during the initial deployment of the cluster. In this regard, through evaluation of the ratio of the distribution of centrality measures ideal instance levels of pods could be deduced.

**4.1.4 Node latency map**

The node latency map component defines the intra-node latencies as well as inter-node latencies. It identifies a pair of nodes among the combinations of nodes and it measures the Round-Trip Time (RTT) values between nodes. This measurement is used to identify nearby nodes and nodes placed further away in the cluster. This measure is used in combination with the predicted link dependency measures to effectively facilitate the identification of highly co-dependent microservices in the deployment and perform the optimization process.

**4.2 System Development and Implementation**

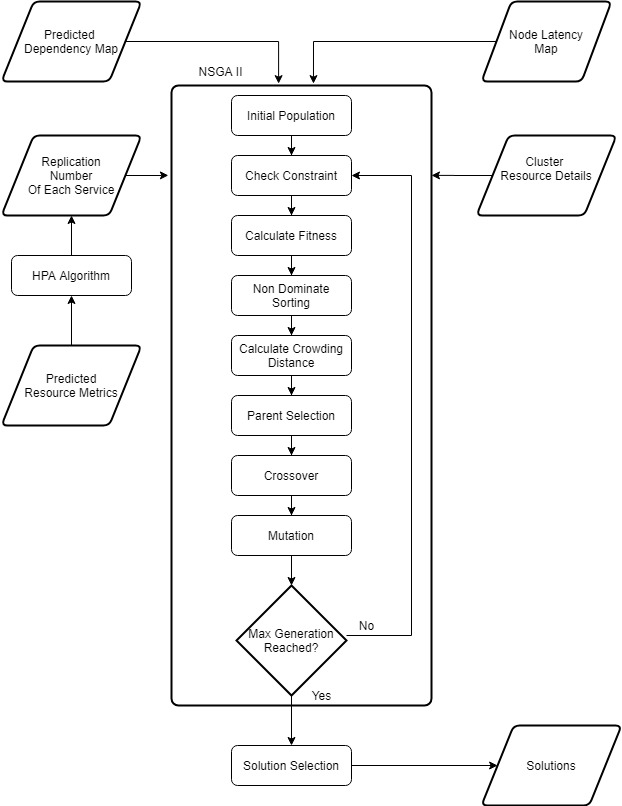


Figure 4.2 Optimization Algorithm Methodology

**4.2.1 HPA Algorithm Component**

The Horizontal Pod Autoscaler (HPA) is a key tool in the Kubernetes platform. It is quite a simple tool that requires a minimal configuration process and has the responsibility for minimizing or maximizing the number of pod instances according to the resource metrics such as CPU and memory. In the HPA, the number of instances is generated by using the simple algorithm. This is described as per formula (1), on the Kubernetes official website [14].

Table 4.1 HPA Algorithm Variable Definition

|  |  |
| --- | --- |
| CurrentReplicas | The number of instances that are currently created. |
| CurrentMetricValue | The current average metric value of the instances. |
| DesiredMetricValue | The metric values that DevOps engineers want to operate. |

(1)

In contrast to the above described HPA algorithm which works with real-time resource metrics and utilizes a reactive approach to autoscaling, the developed optimization algorithm primarily utilizes predicted data to generate optimized solutions. This proactive approach to the determination of the optimal number of replicas is a key advantage of the developed optimization algorithm since auto-scaling decisions could be performed ahead of time, hence improving upon factors such as QoS. In this regard, the developed optimization algorithm utilizes a slightly altered version of the above-depicted formula (1) to predict replicas required in the future by replacing the “*currentMetricValue*” term with the "*predictedMetricValue*" term as in formula (2). Finally, the algorithm produces the desired number of replicas for future deployments.

(2)

**4.2.2 Objective Function**

The determination of microservice optimal deployments is a total optimization problem. It means that the optimization process consists of multiple minimizations and maximization functions with constraints. In simple terms, the optimization algorithm needs two objective functions such as performance and availability, but it is broken into 5 sub-functions to reduce function complexity.

**4.2.3 Latency Objective**

There are several ways to calculate the latency of a particular application. Nevertheless, the most well-known method of evaluating latency is to calculate the RTT values. In this regard, when the client sends the HTTP request to the API server, after a small-time, the response comes back to the client machine. The total time taken for this process is identified as the RTT time and is calculated by obtaining the sum of CPU execution time and network latency time. Unfortunately, Kubernetes cannot go inside to code level to minimize CPU execution time. However, it can do pod level modifications such as changing pod placement or changing pod resource boundaries. In this regard, the governance model minimizes overall network latency. That increases application performance, and in other words, it minimizes response time.

TABLE 4.2 LATENCY OBJECTIVE CALCULATION

|  |  |
| --- | --- |
| n | Number of dependencies in pod-level |
| m | Number of dependency links in app-level |
| W | Dependency request weight in app-level |
| L | The latency of dependency in pod-level |
| D | Dependency average latency in app-level |
| TL | Total latency |

According to the above algorithms (3) and (4), first, the model identifies the dependencies between services through evaluation of the request weight (requests per second) between two microservices in the cluster. Next, the model identifies where the relevant services should be deployed over the cluster. After that, it identifies the pod level dependency links.

For example, as per Fig. 4.3 provided below, assume a cluster has three microservices A, B, and C. It has two dependency links A-B and B-C. A has three instances, B has two instances and C has only one instance. These instances have virtual links for communicating with each other. In this example, consider that between A and B microservice instances, there exists 6 links in total.

Note that microservice A-C have not any links, since it has a request weight 0, indicating there is no dependency level between the two microservices.

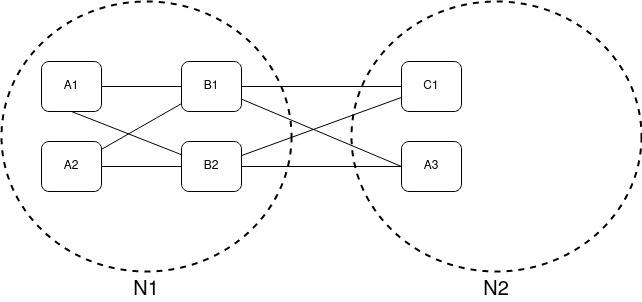


Figure 4.3 Pod Level Dependency Map with Dependency Links

In the next step, the governance model obtains these links latencies by using a node latency map. Next, the model gets each dependency average latency according to formula (3). After that, each dependency average latency is multiplied by each dependency weight to determine the sum of latency as depicted by formula (4).

**4.2.4 Availability Objective**

The availability objective is broken into three sub-objectives for ease of implementation. The sub-objectives consist of determination the required microservice instances, evaluation of the ratio microservice distribution as well as evaluation of the scale of the distribution of microservices among nodes in the cluster. However, the major concern of the availability objective is the determination of the required microservice instances.

HPA algorithm produces the desired replicas of each microservice. That value is sent to the availability function to calculate availability metrics as follows.

TABLE 4.3 AVAILABILITY OBJECTIVE CALCULATION

|  |  |
| --- | --- |
| R | Required instances for each service |
| S | The current number of instances in each service |
| TA | Availability fitness |
| n | Number of microservices |

In this algorithm, in each service, the current number of instances divided by desired replicas. If it is greater than 1, that means the current solution has a greater number of instances than required. However, these values are not directly used in the algorithm. The reason for this is due to the fact that is there is a possibility of obtaining a higher score by generating more instances from one service. To resolve this issue, the resulting value is put into a generalized logistic function [17]. In this regard, the generalized logistic function is configured to generate values between –1 and 1, such that one service can get the maximum score of 1.

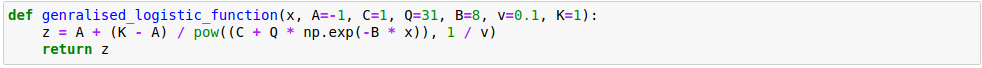


Figure 4.4 Generalized Logistic Function

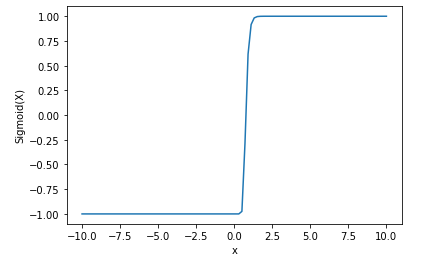


Figure 4.5 Generalized Logistic Function Graph

As depicted by fig 4.4 and fig 4.5, ratio 0.9 to ratio 1 generated value difference is configured as very high. If the ratio is below the 1, it means it does not have the desired replicas set. After that, each service generalized logistic value is multiplied by the desired replicas. Because most required values need more importance level in the scoring method. Finally, it calculates the total availability.

**4.2.5 Ratio Objective**

Even if the availability function is there, it does not ensure the algorithm keeps the proper instances ratio between services. Hence a different function is needed to keep the proper ratio between services.

TABLE 4.4 RATIO OBJECTIVE CALCULATION

|  |  |
| --- | --- |
| R | Required instances for each service |
| S | The current number of instances in each service |
| M | Multiplication value |
| TRD | The total value of instances ratio difference |
| n | Number of microservices |

The first function calculates multiplication (M) value through calculating the ratio of current services present over the required instances as defined in the algorithm (6) above. The calculated value M is then utilized to ensure whether each microservice instance is properly spread over the cluster according to the calculated multiplication values as depicted by formula (7). If the TRD value is 0, the ratio of existing and required instance are the same, however, if the value is higher than 0 it implies there is a significant difference between the current and desired instance levels, hence a minimization process is required.

**4.2.6 Scale and Cost Objective**

The scale function calculates another type of availability. But this value is more related to fault tolerance. It takes high values when the instances of each service are spread over the whole cluster. This is a maximization objective.

The cost function calculates the total cost of the allocated nodes. If a node has not any deployed services, its cost is not added to the total cost. This is a minimization objective.

**4.2.7 Resource constraint and minimal maximum instances constraint**

Before a pod is deployed in a Kubernetes cluster, usually DevOps engineers configure pod threshold values pertaining to CPU and memory, since it is the best practice when performing a typical Kubernetes deployment [14][16]. The threshold values are used to check whether maximum resource consumption values are exceeded or not. If any node resource power is exceeded by deployed pod resource consumption, it is considered as an invalid solution. In this regard, the model does not need to monitor real-time resource consumption, it uses pod threshold value for resource consumption values.

Also, the governance model checks whether services have a minimal number of instances. The default minimum value is one. This constraint ensures whether the services are present in the solution. Also, some services like DB master services need only one service [18], so the model checks whether the service exceeds the maximum instances and hence also plays a key role in regulating other objective functions.

**4.2.8 NSGA II Implementation and Execution Order**

To solve the optimization problem, the NSGA II algorithm is used. In this genetic algorithm, one solution is equal to one chromosome [19]. By generating many random solutions through the utilization of the NumPy library, the governance model initializes the NSGA algorithm. Moreover, the initialization function needs a preassigned number list containing values ranging from 0 to the maximum number of microservices instances that can be deployed in a given node. also, the initialization function adds a sequence of numbers between 0 and the maximum range. The algorithm is designed to generate numbers with a high value in lower probability because higher numbers have more chances to generate invalid solutions.

This developed genetic algorithm needs a good chromosome design to reach the optimal solution. A regular genetic algorithm uses a 1 - dimensional chromosome. However, this problem is too complex to solve using a 1 - dimensional chromosome. In this case, the algorithm uses a 2-dimension chromosome for ease of development.

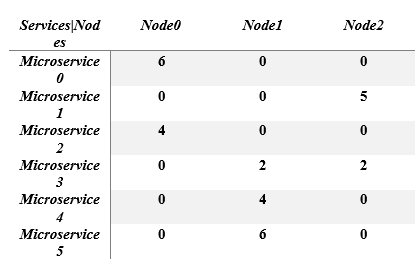


Figure 4.6 Chromosome Design

As shown in Fig. 4. 6 above, each service is identified by rows of the 2-dimensional chromosome. Each node is identified by columns. The intersection of rows and represents the optimal number of instances of a given microservice that should be present in order to achieve the required optimization goal.

For example, as depicted in Fig. 4. 6 above, the value present in the first row of the first column is 6. Whereas the remaining two values of the remaining two columns of the first row are 0. This representation indicates that 6 instances of microservice 0 should be deployed in node 0, and no instances from the microservice 0 should be deployed on the remaining two nodes.

In the NSGA II algorithm, the first step is generating the initial chromosome population. To do this DevOps engineers can set the initial population size or time duration. However, configuring population time is not the optimal way to perform this task. The primary reason for this is due to the fact that the algorithm execution time is unpredictable. However, the best way to perform this task is to configure the time duration, since the model could predict the future solution before the given deadline time.

Each chromosome is made by using NumPy random array modules. After creating one chromosome, the model immediately checks the constraint. If a chromosome is invalid, another chromosome is regenerated. It takes considerable time to completely generate the initial population.

Next, the initial population is delivered for the fitness calculation. At this stage, the model calculates all the solutions pertaining to the best performance, best availability, overall optimal solution, and overall cost-effective cluster deployment solution. In this regard, Table 4.5 depicts the key characteristics pertaining to each of the above-stated solution types.

Table 4.5 Final Solution Option Characteristics

|  |  |
| --- | --- |
| Best Performance | * Minimize latency. * Keep good availability. * Keep instances ratio. |
| Best Availability | * Maximize availability. * Maximize scale value * Keep instances ratio. |
| Overall Optimal | * Minimize latency. * Keep good availability. * Keep scale value * Keep instances ratio. |
| Cost-Effective | * Keep good availability * Reduce cost |

For ease of implementation, the algorithm is implemented to support only maximization objective functions. Hence, for minimization objectives, the inverse ratio derived through the optimization process is utilized.

(8)

For example, in the latency objective algorithm, the solution with the best performance is obtained through replacing by as depicted by formula (8). Also, the ratio objective is multiplied by –1 to convert to maximization objective function. So Now all the objectives are maximization objective. Also, the availability function is modified as given below.

(9)

In the above formula, the values for the *desiredAvailability* and *solutionAvailability* are calculated by using the availability objective function. The desired availability instance ratios are always one. However, the solution availability value changes according to the microservice instance ratio Most of the time this new availability function returns values greater than 0 if the desired replicas are fulfilled. Otherwise, the function always returns minus values.

Next, all the chromosomes and their fitness values are delivered to the Non-dominating Sorting Genetic Algorithm (NSGA). This algorithm ranks all the solutions according to fitness values. However, the algorithm outputs some solutions which receive the same ranks and hence do not dominate each other.

When this type of solution set is obtained, it is known as one Pareto front. After that, each Pareto front solution is ranked by using crowding distance measurement. However, still, there could be a possibility of some solutions having the same ranks in the solution list.

Next, the ranked solution list is delivered to the crossover function. Before performing crossover, the NSGA algorithm assigns the meeting probability according to their rank, because parent selection happens on the Rank Selection method [20]. Finally, the crossover is performed along with a single-point-crossover [21].

Also, in this process, if the produced child has an invalid chromosome, the model does parent selection and reproduces another child. This loop continuously recurs until the required child population is fulfilled.

After children are produced, these new solution chromosomes belong to the next generation. Once again, the fitness function is calculated and the above-described processes are repeated until the maximum generation count is reached or the time duration is over. Finally, the resulting solution is saved in a temporary file through the utilization of the Pandas data frame library available in Python, such that the solutions could be retrieved in the future. Also, users can change algorithm parameters such as population size, number of generations by changing values in the config.ini file.

This algorithm is continuously run in constant time intervals facilitated by using the python scheduler module. Using by panadas data frame, the model retrieves the above-stated solutions by querying. To access these solutions and information, the governance model uses an API server that is implemented by using the flask framework.

**4.3 Commercialization**

The commercialization of this research project is mainly considered through the development of a tool through the use of the developed governance model. In this regard, the developed tool will be implemented as a Business Intelligence Dashboard which makes use of the developed governance model to provide developers and system administrators an easy and efficient way in which to optimize their Kubernetes deployment by aiming to provide the following benefits.

* Visualize the level of inter-dependency among deployed microservices.
* Receive suggestions in potential ways to optimize the performance and configure current deployments and automatically perform deployments based on the suggestions.
* Provide an overview of the resiliency of the deployed microservices.
* Automatically configure and auto-scale Kubernetes autoscaling tools based on predicted load and centrality measures.

The developed Business Intelligence Dashboard will allow users to access all the above-mentioned features and provide a holistic view of their deployments. Hence, this tool will be mainly targeted to be marketed as an APM tool for Kubernetes deployments for system administrators and developers. Due to the wide variety of APM tools currently available in the market which are mostly free and opensource, the initial plan is to develop this dashboard into an opensource tool in order to enter the current market space effectively. However, throughout the years, a freemium based marketing strategy will be adopted with the inclusion of additional features.

1. **RESULTS AND DISCUSSION**

**5.1 Algorithm selection**

Microservice optimal deployment is a multi-objective NP-hard optimization problem[11]. To solve this kind of problem, a brute force approach can be used in order to guarantee to find a globally optimal solution. However, this approach consumes a lot of time ,because brute force method evaluates all the possible scenarios rather than advance techniques to improve efficiently[22]. On the other hand, NSGA Ⅱ is a metaheuristic optimization algorithm. In this case, the algorithm finds the best generate solution through self-learning, in a relatively smaller time. However, it does not guarantee to a globalized optimal solution [12].

In our example, the sample test-bed for evaluating the developed optimization algorithm consisted of 6 microservices deployed on 3 nodes in a Kubernetes cluster. The above-stated representation of microservices is then arranged into a 1-dimensional array as in Fig 5.1. This is consist of 18 integer numbers.

A picture containing clock

Description automatically generated

Figure 5.1 Latency Fitness Graph

For example, assume if the brute force model only generates numbers between 0 - 10, there would be 1018 possible solutions. When these solutions are evaluated on a Core i7 5500u CPU with 12 GB DDR3 ram, it takes 108421.41 ns to calculate all the objective values per a given solution. Hence, if we had to use the brute force method, the worst-case time would be as below.

Worst case execution time = 108421.41 ns \* 1018

= 1.0842141×10²³ ns

This above value is equal to 82512488.584474886 years. Hence, it would unreachable, even if the model had been run on high-performance computers.

In this example, the NSGA Ⅱ algorithm uses 300 solutions, 1200 generations, and 1000 initial solutions. Its execution time is 20 minutes. However, the best solutions can occur from lower generations as well. In this scenario, generation 730 produces the solution with the best performance.

Figures 5.2, and 5.3 depict graphs that depict the improvements in the overall mean fitness pertaining to each increasing generation number pertaining to latency and availability.

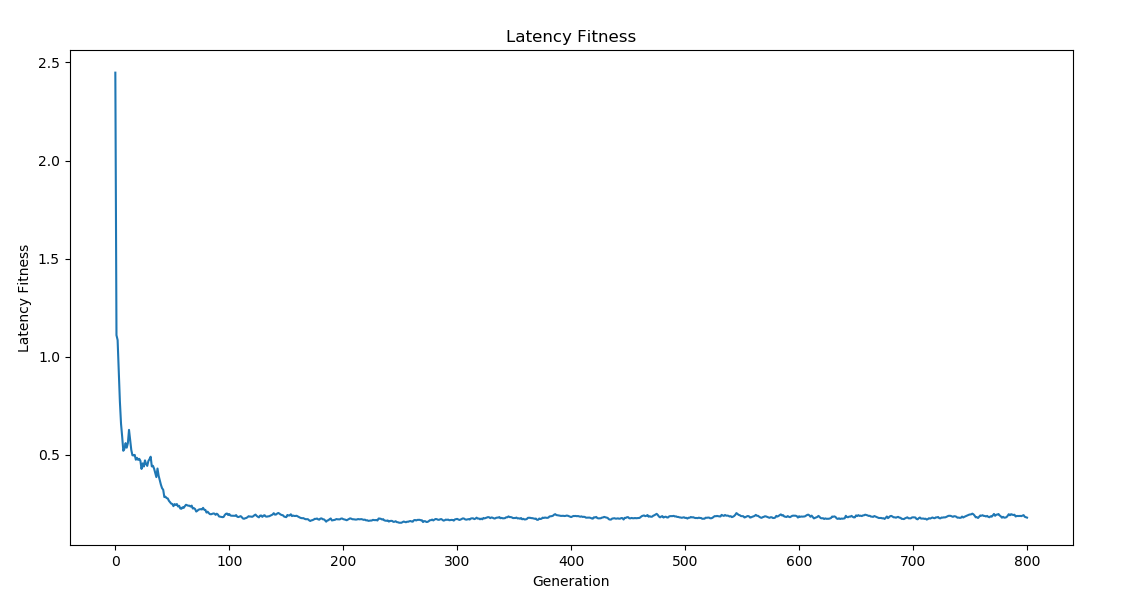


Figure 5.2 Mean Latency Fitness Graph

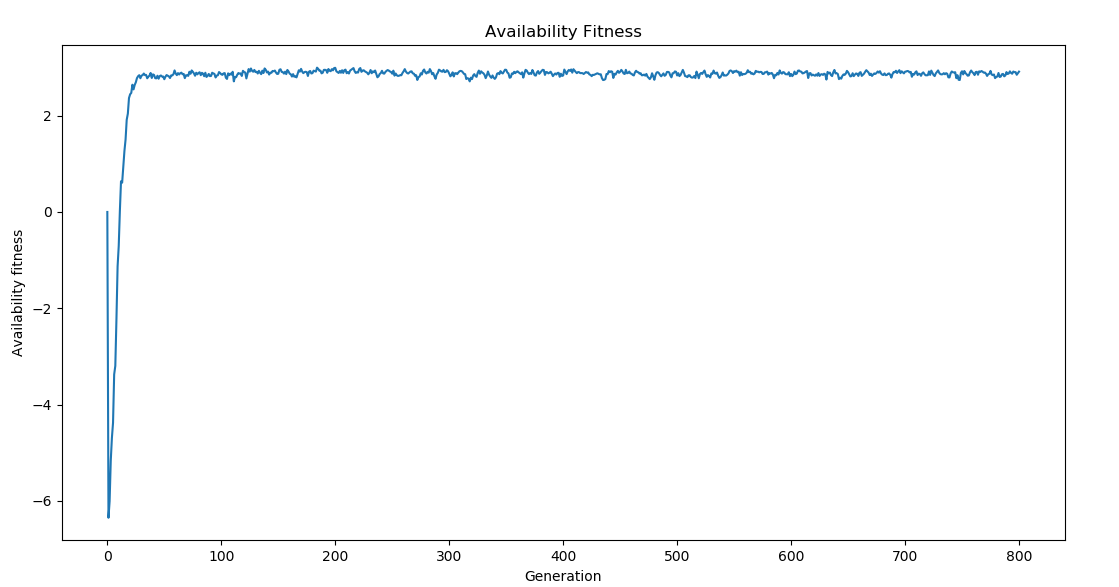


Figure 5.3 Mean Availability Fitness Graph

**5.2 Results**

As previously stated, in the testing stage, the 6 sample microservice applications are deployed on 3 nodes cluster. For evaluation purposes, the JSON (JavaScript Object Notation) representation of this cluster dataset, along with the additional information required by the optimization algorithm which includes the node latency map, predicted inter-microservice dependency measures as well as the required number of microservice instances, is provided to the developed optimization algorithm in order to compute the optimized solutions. Fig. 5.4 below depicts the structure of the sample input JSON provided to the optimization algorithm.

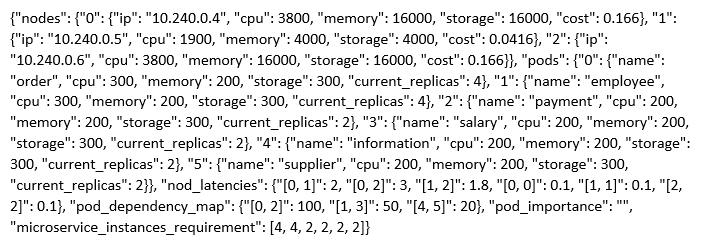


Figure 5.4 Structure of sample JSON input

Table 5.1: Metric Measurement Units

|  |  |
| --- | --- |
| Latencies | Milliseconds |
| Dependency | Number of requests per minute |
| CPU | Milli cores |
| Memory, storage | Megabytes |

The existing cluster deployment view is as depicted given below.



Figure 5.5 Existing Performance Deployment View

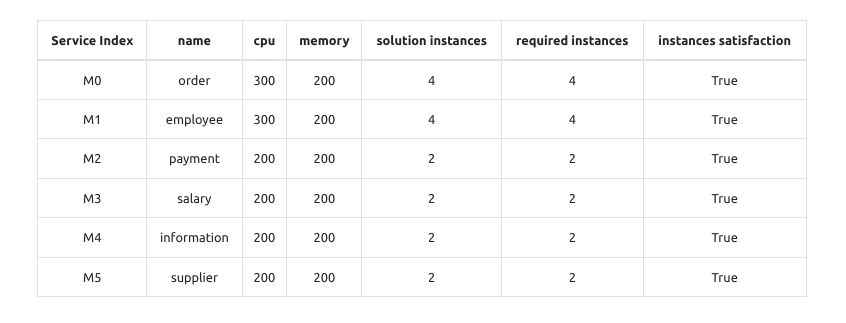


Figure 5.6 Existing Performance instances Table

After running the optimization algorithm, 4 optimal solutions which represent the deployment solutions pertaining to the best performance, best availability, overall optimal cluster, and highest cost-effectiveness, are generated.

The parameters for the initialization of the NSGA II algorithm is as depicted in Table 5.2 below.

Table 5.2 NSGA II Algorithm Initialization Parameters

|  |  |
| --- | --- |
| Initial Population size | 1000 |
| Number of generations | 1200 |
| Number of solutions for the generation | 300 |

**5.2.1 Best Performance Deployment Strategy**

The resulting output which represents the solution pertaining to the best performance of the cluster is as follows.



Figure 5.7 Best Performance Deployment View

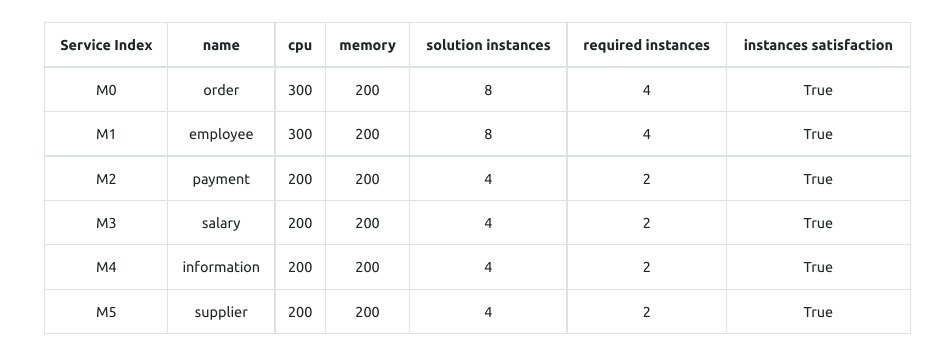


Figure 5.8 Best Performance Instances Table

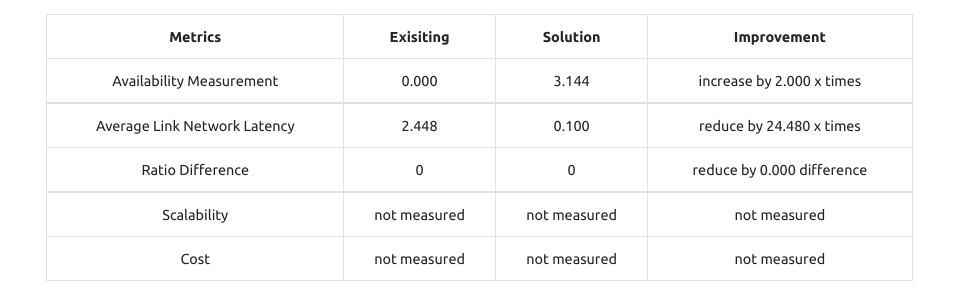


Figure 5.9 Best Performance Metrics Improvement Table

As depicted by Fig 5.4, according to the sample dependency map, between the microservice at index 0 and microservice at index 2 microservice, 1000 requests per minute have been sent. The requests per minute represent the microservice dependency level between the payment and order microservices. Likewise, between employee and salary microservices, 500 requests per minute have been recorded and between the information and supplier microservices have 200 requests per minute have been recorded. More as evident in the diagram, there are only 3 dependencies between each of the microservices.

As depicted in Fig. 5.7, we could see that dependent services are deployed on the same node. Hence, as a result of this optimization process, the latency has been minimized by 24 times. Moreover, in the cluster, there is enough space to deploy all dependent services on the same nodes, otherwise, the algorithm selects nearby nodes to deploy the highly dependent services while also maintaining the ratio between the desired replicas.

Furthermore, the memory and CPU margins are set as 10% of the resources available. Hence, each generated solution always keeps more than 10% of resources free. As evident in Fig. 5.9 this solution increases availability by maximizing the number of instances 2 times more than the required instances.

**5.2.2 Best Availability Deployment Strategy**

This solution has 2.125 times more instances than the existing deployment. Also, it keeps a good ratio between instances and keeps the 10% resource margins.



Figure 5.10 Best Availability Deployment View

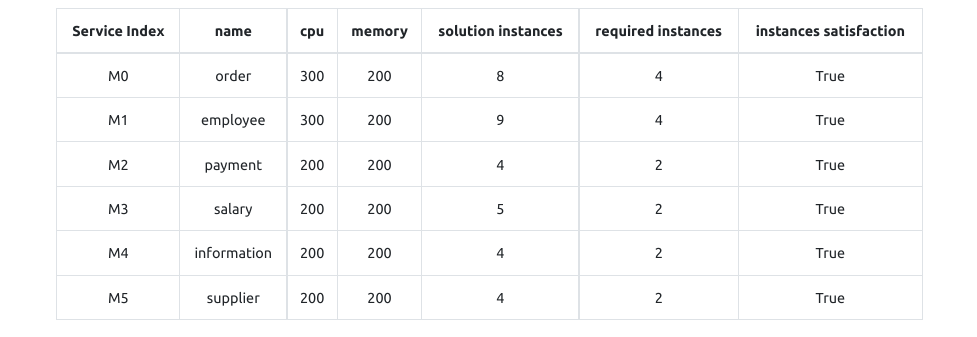


Figure 5.11 Best Availability Instances Table

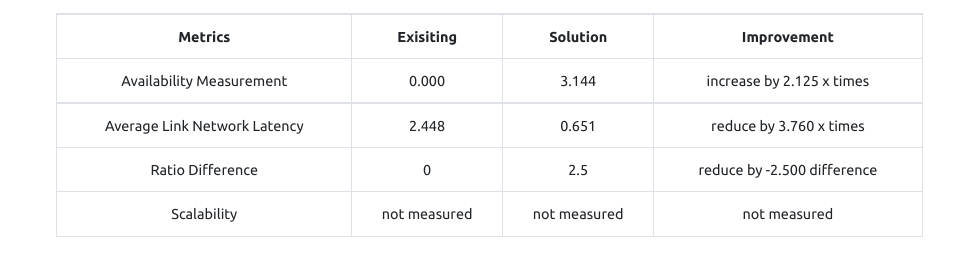


Figure 5.12 Best Availability Metric Table

**5.2.3 Cost-effective Deployment Strategy**

This option keeps good availability and reduces the total cost. In this example, it only deploys over the 2 nodes and it selects a cluster with a lower budget.



Figure 5.13 Cost-Effective Deployment View.

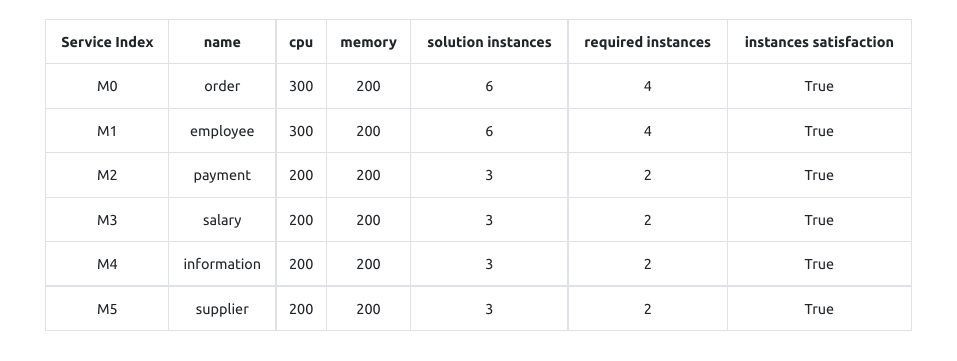


Figure 5.14 Cost-Effective Instances Table

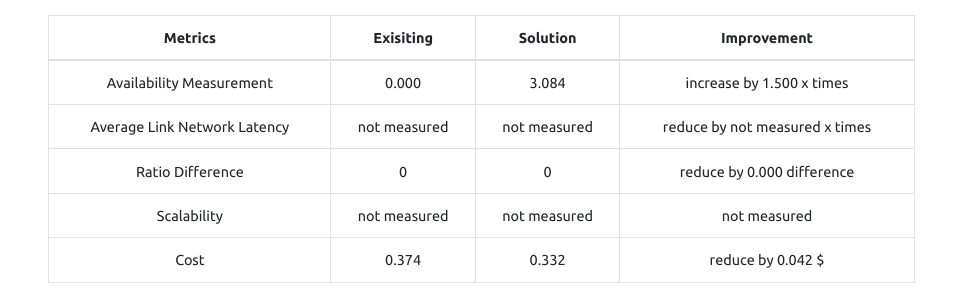


Figure 5.15 Cost-Effective Metric Improvement Table.

**5.2.4 Optimal Deployment Strategy**

This option generates the same solution as the performance-optimized solution delivered. The only difference is the scale value. The scale values determine the distribution of microservices throughout the cluster

Moreover, the primary reason for that creation of the optimal deployment strategy solution is because, in deployment strategies pertaining to the best performance, multiple services may be converged into a single node, which in turn may reduce fault tolerance within the cluster. Moreover, the developed optimization algorithm allows users to configure the minimal scale values (number of instances of a particular microservice deployed in a given node) such that the optimal deployment strategy could be created as per user requirements.

**5.3 Research Findings and Discussion**

The above result reveals how effective the NSGA II algorithm is for solving optimization problems. Results also reveal the algorithm reaches a more feasible, optimal solution in a relatively small-time duration. Moreover, the NSGA algorithm also supports non dominate sorting as well as multi-objective functionalities, hence a good a chromosome design is more important to converge toward the optimal solution.

The average latency of the entire application can be minimized by using the optimal placement method. However, it also depends on cluster network latency. Users can get more beneficial if the application is deployed over the large cluster because nodes are located some distance apart. Even if the cluster is small, the algorithm guarantees to minimize latency. However, at times, the difference can be too small.

The application availability is highly dependent on the number of instances that it deploys over the cluster. Whether the cluster is small or large, it produces the maximum number of instances. Also, it maintains good instances ratio between services. Indirectly, the developed governance model minimizes the application response time when the application is in high traffic.

**6.0 CONCLUSION**

The current implementation of MAPE based autoscaling techniques used in autoscaling tools in microservice orchestration tools such as Kubernetes are primarily rule-based and hence it becomes difficult to adapt to the dynamic workloads microservices experience resulting in ineffective scaling and a drop in QoS.

Furthermore, in order to effectively develop an autoscaling policy that optimally makes use of available resources and, in turn, minimize the effect of the above-mentioned problems, it becomes necessary to take into consideration a global view of each microservice and how it is being utilized.

This research, as described above, thereby aims to provide a solution to this problem through a statistical and machine learning-based approach to effectively predict load through metric analysis while incorporating centrality-based evaluation techniques to determine the centrality of a microservice so as to obtain a more holistic view of the utilization of each microservice and ultimately develop a more effective autoscaling policy.

**References**

[1]"docker history", Docker Documentation, 2020. [Online]. Available: https://docs.docker.com/engine/reference/commandline/history. [Accessed: 24- Feb- 2020].

[2]"How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play", Medium, 2020. [Online]. Available: https://medium.com/refraction-tech-everything/how-netflix-works-the-hugelysimplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b. [Accessed: 24- Feb- 2020].

[3]"8 Container Orchestration Tools to Know - Linux.com", Linux.com, 2020. [Online]. Available: https://www.linux.com/news/8-open-source-containerorchestration-tools-know/. [Accessed: 25- Feb- 2020].

[4]Vaadin.com, 2020. [Online]. Available: https://vaadin.com/blog/microserviceshigh-availability. [Accessed: 25- Feb- 2020].

[5]"Evaluating Critical Performance Needs for Microservices and Cloud-Native Applications", Medium, 2020. [Online]. Available: https://medium.com/netifi/evaluating-critical-performance-needs-formicroservices-and-cloud-native-applications-7675c50a8460. [Accessed: 25- Feb- 2020].

[6]"Test Azure virtual machine network latency in an Azure virtual network", Docs.microsoft.com, 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-testlatency. [Accessed: 25- Feb- 2020].

[7]"Azure network round trip latency statistics", Docs.microsoft.com, 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/networking/azure-networklatency. [Accessed: 25- Feb- 2020].

[8]"Co-locate Windows Azure VMs - Azure Windows Virtual Machines", Docs.microsoft.com, 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/co-location. [Accessed: 25- Feb- 2020].

[9]"Mathematical optimization", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Mathematical\_optimization. [Accessed: 25- Feb- 2020].

[10]"Linear programming", En.wikipedia.org, 2020. [Online]. Available: <https://en.wikipedia.org/wiki/Linear_programming>. [Accessed: 18- Sep- 2020].

[11]"NP-hardness", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/NP-hardness. [Accessed: 18- Sep- 2020].

[12] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, April 2002, doi: 10.1109/4235.996017.

[13]"Kubernetes Scheduler", Kubernetes.io, 2020. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/. [Accessed: 25- Feb- 2020].

[14]"Horizontal Pod Autoscaler", Kubernetes, 2020. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. [Accessed: 18- Sep- 2020].

[15]M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi and G. Zavattaro, "Optimal and Automated Deployment for Microservices", Fundamental Approaches to Software Engineering, pp. 351-368, 2019. Available: 10.1007/978-3-030-167226\_21 [Accessed 25 February 2020].

[16]"Assign CPU Resources to Containers and Pods", Kubernetes, 2020. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/. [Accessed: 18- Sep- 2020].

[17]"Generalised logistic function", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Generalised\_logistic\_function. [Accessed: 18- Sep- 2020].

[18]"Database Master-Slave Replication in the Cloud | MariaDB", MariaDB, 2020. [Online]. Available: https://mariadb.com/resources/blog/database-master-slave-replication-in-the-cloud/. [Accessed: 18- Sep- 2020].

[19]"Genetic algorithm", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Genetic\_algorithm. [Accessed: 18- Sep- 2020].

[20] "Genetic Algorithms - Parent Selection - Tutorialspoint", Tutorialspoint.com, 2020. [Online]. Available: https://www.tutorialspoint.com/genetic\_algorithms/genetic\_algorithms\_parent\_selection.htm. [Accessed: 25- Sep- 2020].

[21]"Genetic Algorithms - Crossover - Tutorialspoint", Tutorialspoint.com, 2020. [Online]. Available: https://www.tutorialspoint.com/genetic\_algorithms/genetic\_algorithms\_crossover.htm. [Accessed: 18- Sep- 2020].

[22]"Brute Force Algorithms Explained", freeCodeCamp.org, 2020. [Online]. Available: https://www.freecodecamp.org/news/brute-force-algorithms-explained/. [Accessed: 20- Sep- 2020].

**Appendix**

**Appendix A: Implementation of NSGA II Algorithm**

import numpy as np  
import time  
from src.ga\_modules.constraint import \*  
from src.ga\_modules.crossover\_functions import \*  
from src.ga\_modules.fitness\_functions import \*  
from src.ga\_modules.non\_domination\_sorting import \*  
from src.tools.report\_generator import \*  
from src.models.solution\_metrics import SolutionMetrics  
from src.ga\_modules.choices import \*  
from src.tools.list\_generator import \*  
  
  
# from typing import \*  
  
  
class ClusterOptimizationAlgorithm:  
  
 def \_\_init\_\_(self, existing\_chromosome, initial\_population, cluster, number\_of\_generation, next\_generation\_size,  
 run\_time,  
 cpu\_limit\_precentage=0.9, memory\_limit\_precentage=0.8, storage\_limit\_precentage=0.8,  
 population\_effect\_mutation=0.1, chromosome\_effect\_mutaion=0.2, genes\_mutation\_max=4,  
 genes\_mutation\_min=0, fitness\_check={'performance': 1, 'availability': 1, 'scalability': 1},  
 limited\_scale=None):  
 self.existing\_chromosome = existing\_chromosome  
 self.initial\_population = initial\_population  
 self.cluster = cluster  
 self.number\_of\_generation = number\_of\_generation  
 self.next\_generation\_size = next\_generation\_size  
 self.run\_time = run\_time if run\_time != None else float('inf')  
 self.cpu\_limit\_precentage = cpu\_limit\_precentage  
 self.memory\_limit\_precentage = memory\_limit\_precentage  
 self.storage\_limit\_precentage = storage\_limit\_precentage  
 self.population\_effect\_mutation = population\_effect\_mutation  
 self.chromosome\_effect\_mutaion = chromosome\_effect\_mutaion  
 self.genes\_mutation\_max = genes\_mutation\_max  
 self.genes\_mutation\_min = genes\_mutation\_min  
 self.fitness\_check = fitness\_check  
 self.limited\_scale = limited\_scale  
 self.existing\_deployment\_metrics = None  
  
 @staticmethod  
 def generate\_initial\_population\_v3(cluster, constrained\_first\_generation\_size, run\_time,  
 required\_instances, cpu\_limit\_precentage=0.9, memory\_limit\_precentage=0.8,  
 storage\_limit\_precentage=0.8):  
 # set initial population function execution time ,when there is no inital population size present  
 run\_time = run\_time if run\_time != None else float('inf')  
 number\_of\_nodes = len(cluster.nodes)  
 number\_of\_microservices = len(cluster.pods)  
 current\_pop\_size = 0  
 np.random.seed(42)  
 random\_population\_list = []  
  
 # generate random numberlist  
 choices = generate\_random\_choice\_numbers(cluster, cpu\_limit\_precentage, memory\_limit\_precentage,  
 storage\_limit\_precentage, required\_instances)  
 choices = set(choices)  
 choices = list(choices)  
 choices.sort()  
 print('choices', choices)  
 chromosome\_size = number\_of\_microservices \* number\_of\_nodes  
 count = 0  
  
 # add probabilty for each random number  
 probability\_list = generate\_probability\_list(len(choices))  
  
 next = True  
  
 start\_time = time.perf\_counter\_ns()  
 while next:  
 chromosome = np.random.choice(choices, chromosome\_size, p=probability\_list)  
 # check resource constraint  
 valid = check\_constraint\_for\_chromosome(chromosome, cluster, cpu\_limit\_precentage,  
 memory\_limit\_precentage,  
 storage\_limit\_precentage)  
 count = count + 1  
 end\_time = time.perf\_counter\_ns()  
 if valid:  
 random\_population\_list.append(chromosome)  
 current\_pop\_size = current\_pop\_size + 1  
 current\_run\_time = (end\_time - start\_time) / (1000000000 \* 60)  
 next = current\_pop\_size <= constrained\_first\_generation\_size and current\_run\_time <= run\_time  
 # print size ,when the size is sequence of 50  
 if (current\_pop\_size % 50 == 0):  
 print('pop size', current\_pop\_size)  
  
 return np.array(random\_population\_list)  
  
 # calculte exisiting cluster fitness values  
 def \_\_calculate\_existing\_deployment\_metrics(self):  
 objectives = dict()  
 print('existing', self.existing\_chromosome)  
 if self.fitness\_check[PERFORMANCE\_STR] == 1:  
 objectives[PERFORMANCE\_STR] = calculate\_fitness\_performormance(self.existing\_chromosome, self.cluster)  
 if self.fitness\_check[AVAILABILITY\_STR] == 1:  
 objectives[AVAILABILITY\_STR] = calculate\_fitness\_availability\_v3(self.existing\_chromosome, self.cluster)  
 if self.fitness\_check[RATIO\_STR] == 1:  
 objectives[RATIO\_STR] = calculate\_fitness\_ratio(self.existing\_chromosome, self.cluster)  
 self.existing\_deployment\_metrics = SolutionMetrics(0, 0, self.existing\_chromosome[0], objectives)  
 if self.fitness\_check[SCALABILITY\_STR] == 1:  
 objectives[SCALABILITY\_STR] = calculte\_fitness\_scale\_v3(self.existing\_chromosome, self.cluster,  
 self.limited\_scale)  
 if self.fitness\_check[COST\_STR] == 1:  
 objectives[COST\_STR] = calculate\_fitness\_cost\_effective(self.existing\_chromosome,  
 self.cluster)  
  
 # check whether the resources constraint are satisfied  
 def \_\_check\_constraint(self, population):  
 invalid\_chromosome\_indexes = list()  
 for chromosome\_index in range(0, len(population)):  
 chromosome = population[chromosome\_index]  
 valid = check\_constraint\_for\_chromosome(chromosome, self.cluster, self.cpu\_limit\_precentage,  
 self.memory\_limit\_precentage,  
 self.storage\_limit\_precentage)  
 if (not valid):  
 invalid\_chromosome\_indexes.append(chromosome\_index)  
 population = np.delete(population, invalid\_chromosome\_indexes, axis=0)  
  
 return population  
  
 # reduce number of instances,when the instances are created over maximum limit  
 def \_\_reduce\_number\_of\_instances(self, population):  
  
 for chromosome\_index in range(0, len(population)):  
 population[chromosome\_index] = modify\_chromosome(population[chromosome\_index], self.cluster,  
 self.cluster.master\_pod\_indexes)  
  
 return population  
  
 # calculate one generation all solution fitness values  
 def \_\_cal\_pop\_fitness(self, population, generation):  
 solutions = list()  
 start\_time = time.perf\_counter\_ns()  
 for index, chromosome in enumerate(population):  
 objectives = dict()  
 if self.fitness\_check[PERFORMANCE\_STR] == 1:  
 objectives[PERFORMANCE\_STR] = calculate\_fitness\_performormance(chromosome, self.cluster)  
  
 if self.fitness\_check[AVAILABILITY\_STR] == 1:  
 objectives[AVAILABILITY\_STR] = calculate\_fitness\_availability\_v2(chromosome, self.cluster)  
  
 if self.fitness\_check[RATIO\_STR] == 1:  
 objectives[RATIO\_STR] = calculate\_fitness\_ratio(chromosome, self.cluster)  
  
 if self.fitness\_check[SCALABILITY\_STR] == 1:  
 objectives[SCALABILITY\_STR] = calculte\_fitness\_scale\_v3(chromosome, self.cluster, self.limited\_scale)  
  
 if self.fitness\_check[COST\_STR] == 1:  
 objectives[COST\_STR] = calculate\_fitness\_cost\_effective(chromosome, self.cluster)  
 solutions.append(SolutionMetrics(index, generation, chromosome, objectives))  
  
 end\_time = time.perf\_counter\_ns()  
 # print total fitness evalution execution time for one generation  
 print('solutions', len(population), 'time:', end\_time - start\_time)  
  
 return solutions  
  
 # rank all solution according to fitness values  
 def \_\_select\_mating\_pool(self, solutions: List[SolutionMetrics], population):  
 start\_time = time.perf\_counter\_ns()  
 # doing fast non-dominating sorting  
 ranked\_solutions = nd\_sort\_1d(solutions)  
 end\_time = time.perf\_counter\_ns()  
 print('generation:', 'function runtime:', end\_time - start\_time)  
  
 new\_ranked\_population = list()  
 for solutions in ranked\_solutions:  
 new\_ranked\_population.append(population[int(solutions.index)])  
  
 return new\_ranked\_population, ranked\_solutions  
  
 # single point crossover with Roulette Wheel Selection  
 def \_\_crossover\_v5(self, population):  
 children\_population = list()  
 population\_size = len(population)  
 children\_count = 0  
 np.random.seed(42)  
 # set probability values for Roulette Wheel Selection  
 index\_list = range(0, population\_size)  
 sum\_of\_index = (population\_size / 2) \* (1 + population\_size)  
 probability\_list = [(population\_size - index) / sum\_of\_index for index in index\_list]  
  
 # check whether the generation population size is exceeded or not  
 while children\_count < self.next\_generation\_size:  
 # select random parent by probability values  
 parents = np.random.choice(index\_list, 2, replace=False, p=probability\_list)  
  
 parent\_chromosome\_1 = population[parents[0]]  
 parent\_chromosome\_2 = population[parents[1]]  
 # single point crossover  
 child = chromosome\_half\_cross\_over(parent\_chromosome\_1, parent\_chromosome\_2, self.cluster)  
 # check whether new child has invlid genes  
 valid = check\_constraint\_for\_chromosome(child, self.cluster, self.cpu\_limit\_precentage,  
 self.memory\_limit\_precentage,  
 self.storage\_limit\_precentage)  
 if valid:  
 children\_population.append(child)  
 children\_population = np.unique(children\_population, axis=0).tolist()  
 children\_count = len(children\_population)  
  
 return np.array(children\_population)  
  
  
 # add some mutation to the chromosome  
 def \_\_mutation(self, population):  
 random\_chromosome = np.random.choice(len(population), int(len(population) \* self.population\_effect\_mutation),  
 replace=False)  
 # select new population for doing mutation  
 selected\_population = population[random\_chromosome]  
 for chromosome\_index, chromosome in enumerate(selected\_population):  
   
 # select random genes from chromosome  
 random\_genes\_index = np.random.choice(len(chromosome),  
 int(len(chromosome) \* self.chromosome\_effect\_mutaion),  
 replace=False)  
 # generate new numbers for doing mutation  
 genes\_new\_random\_values = np.random.choice(range(self.genes\_mutation\_min, self.genes\_mutation\_max),  
 len(random\_genes\_index))  
 # mutation  
 for i, genes\_index in enumerate(random\_genes\_index):  
 chromosome[genes\_index] = chromosome[genes\_index] + genes\_new\_random\_values[i]  
 valid = check\_constraint\_for\_chromosome(chromosome, self.cluster, self.cpu\_limit\_precentage,  
 self.memory\_limit\_precentage,  
 self.storage\_limit\_precentage)  
 if chromosome[genes\_index] < 0 or (not valid):  
 chromosome[genes\_index] = chromosome[genes\_index] - genes\_new\_random\_values[i]  
 # update population after mutation  
 population[random\_chromosome[chromosome\_index]] = chromosome  
   
 return population  
   
 # one iteration  
 def \_\_one\_generation(self, generation, current\_generation, report\_generator):  
  
 population = self.\_\_reduce\_number\_of\_instances(current\_generation)  
 fitness\_values = self.\_\_cal\_pop\_fitness(population, generation)  
  
 new\_ranked\_popualtion, ranked\_solution = self.\_\_select\_mating\_pool(fitness\_values, current\_generation)  
  
 report\_generator.add\_solution\_metrics\_list\_from\_gen(ranked\_solution)  
 children\_population = self.\_\_crossover\_v5(new\_ranked\_popualtion)  
 mutated\_popualtion = self.\_\_mutation(children\_population)  
 current\_generation = mutated\_popualtion  
  
 return current\_generation  
  
 def run\_algorithm(self, report\_generator):  
 #start counting time  
 start\_time\_algorithm = time.perf\_counter\_ns()  
   
 current\_generation = self.initial\_population  
 self.\_\_calculate\_existing\_deployment\_metrics()  
   
 #add existing deployment fitness values to the dataframe  
 report\_generator.add\_solution\_metrics\_list\_from\_gen([self.existing\_deployment\_metrics])  
 generation = 1  
 next = True  
 tot\_time\_in\_minute = 0  
   
   
 while next:  
 print('generation:', generation, ' running..')  
 start\_time = time.perf\_counter\_ns()  
 current\_generation = self.\_\_one\_generation(generation, current\_generation, report\_generator)  
 end\_time = time.perf\_counter\_ns()  
 print('generation:', generation, ' runtime:', end\_time - start\_time)  
 generation = generation + 1  
 tot\_time\_in\_minute = (end\_time - start\_time\_algorithm) / (1000000000 \* 60)  
 next = (tot\_time\_in\_minute <= self.run\_time and generation <= self.number\_of\_generation)  
 print('total time:', tot\_time\_in\_minute)

**Appendix B: Implementation of Fitness function**

import numpy as np  
  
  
# calculate instances distribution scale  
def calculte\_fitness\_scale\_v4(chromosome, cluster, limited\_scale):  
 number\_of\_nodes = len(cluster.nodes)  
 number\_of\_pods = len(cluster.pods)  
 # convert chromosome to 2d-array  
 deployment\_matrix = chromosome.reshape(number\_of\_pods, number\_of\_nodes)  
  
 # get each micro-service sum of instances  
 each\_pod\_instances = np.sum(deployment\_matrix, axis=1)  
  
 scale = 0  
  
 for i in range(0, number\_of\_pods):  
 pod\_instances = each\_pod\_instances[i]  
 pod\_expand = 0  
 pod\_scale = 0  
 for j in range(0, len(cluster.nodes)):  
 if deployment\_matrix[i][j] > 0:  
 pod\_expand = pod\_expand + 1  
 if limited\_scale is not None:  
 if limited\_scale < pod\_expand:  
 pod\_expand = limited\_scale  
 else:  
 limited\_scale = number\_of\_nodes  
 for j in range(0, len(cluster.nodes)):  
 pod\_scale = pod\_scale + (pod\_expand \* deployment\_matrix[i][j])  
  
 max\_pod\_scale = limited\_scale \* pod\_instances  
 pod\_scale\_ratio = pod\_scale / max\_pod\_scale  
  
 scale = scale + pod\_scale\_ratio  
  
 return scale  
  
  
# calculte fitness of requirement instances  
def calculate\_fitness\_availability\_v2(chromosome, cluster):  
 deployment\_matrix = chromosome.reshape(len(cluster.pods), len(cluster.nodes))  
 array = np.sum(deployment\_matrix, axis=1)  
 number\_of\_services = len(array)  
 fitness = 0  
 sum\_required\_instances = 0  
 for index, pod\_instances in enumerate(array):  
 required\_instances = cluster.microservice\_instances\_requirement[index]  
  
 # get the ratio between solution number of instances and required number of instances  
 ratio = pod\_instances / required\_instances  
  
 logistic\_value = generalised\_logistic\_function(ratio)  
  
 fitness = fitness + logistic\_value \* required\_instances  
 sum\_required\_instances = sum\_required\_instances + required\_instances  
  
 return fitness - (genralised\_logistic\_function(1) \* sum\_required\_instances)  
  
  
# calculate fitness performance  
def calculate\_fitness\_performormance(chromosome, cluster):  
 deployment\_matrix = chromosome.reshape(len(cluster.pods), len(cluster.nodes))  
  
 # determine total request  
 tot\_of\_req = calculate\_tot\_req(cluster.pod\_dependency\_map)  
 fitness = 0  
  
 for couple\_of\_pod, req in cluster.pod\_dependency\_map.items():  
 # determine 1/network latency of each dependency  
 fitness = fitness + (req / tot\_of\_req) / couple\_of\_pods\_avg\_latency(cluster, deployment\_matrix, couple\_of\_pod,  
 req)  
 return fitness  
  
  
# calculate average network latency of depencies  
def couple\_of\_pods\_avg\_latency(cluster, deployment\_matrix, couple\_of\_pod, req):  
 number\_of\_nodes = len(cluster.nodes)  
 tot\_latency = 0  
 tot\_req\_link\_count = 0  
 # iterate all nodes  
 for i in range(0, number\_of\_nodes):  
 # iterate all pods  
 for j in range(0, number\_of\_nodes):  
 number\_of\_pods\_p1 = deployment\_matrix[couple\_of\_pod[0], i]  
 number\_of\_pods\_p2 = deployment\_matrix[couple\_of\_pod[1], j]  
 # check instances are present on the genes  
 if number\_of\_pods\_p1 > 0 and number\_of\_pods\_p2 > 0:  
  
 # get RTT latency between couple of nodes  
 link\_latency = cluster.node\_latencies.get((i, j)) if cluster.node\_latencies.get(  
 (i, j)) != None else cluster.node\_latencies.get((j, i))  
 if link\_latency != None:  
 req\_link\_count = number\_of\_pods\_p2 \* number\_of\_pods\_p1  
 tot\_latency = tot\_latency + link\_latency \* int(req\_link\_count)  
  
 tot\_req\_link\_count = tot\_req\_link\_count + req\_link\_count  
  
 return (tot\_latency / tot\_req\_link\_count) if tot\_req\_link\_count != 0 else float('inf')  
  
  
def calculate\_tot\_req(pod\_dependency\_\_map):  
 tot\_req = 0  
 for req in pod\_dependency\_\_map.values():  
 tot\_req = tot\_req + req  
 return tot\_req  
  
  
def calculate\_fitness\_cost\_effective(chromosome, cluster):  
 return -(calculate\_cost(chromosome, cluster))  
  
  
def calculate\_cost(chromosome, cluster):  
 deployment\_matrix = chromosome.reshape(len(cluster.pods), len(cluster.nodes))  
 array = np.sum(deployment\_matrix, axis=0)  
 tot\_cost = 0  
 for index, node in cluster.nodes.items():  
 tot\_cost = tot\_cost + (node.cost \* (0 if array[index] == 0 else 1))  
  
 return tot\_cost  
  
  
# configured generalised logistic function  
def generalised\_logistic\_function(x, A=-1, C=1, Q=31, B=8, v=0.1, K=1):  
 z = A + (K - A) / pow((C + Q \* np.exp(-B \* x)), 1 / v)  
 return z  
  
  
# calculate instances ratio difference  
def calculate\_fitness\_ratio(chromosome, cluster):  
 # convert to 2-d array  
 deployment\_matrix = chromosome.reshape(len(cluster.pods), len(cluster.nodes))  
 sol\_pod\_instances = np.sum(deployment\_matrix, axis=1)  
 require\_pod\_instances = cluster.microservice\_instances\_requirement  
 return calculate\_ratio\_difference(require\_pod\_instances, sol\_pod\_instances) \* -1  
  
  
def calculate\_ratio\_difference(req, sol):  
 reqs = sum(req)  
 sols = sum(sol)  
 m = sols / reqs  
 t = 0  
 for i, s in enumerate(sol):  
 t = t + abs(s - req[i] \* m)  
 return t

**Appendix C: Implementation of Fast Non-Dominate Sorting Algorithm**

from src.models.solution\_metrics import SolutionMetrics  
from typing import \*  
from src.models.constant import \*  
  
  
# ranking non domination list of solution  
def fast\_non\_dominate\_sort(solutions: List[SolutionMetrics]):  
 fronts = dict()  
  
 for solution\_p in solutions:  
 for solution\_q in solutions:  
 if solution\_p.this\_dominates\_(solution\_q):  
 # if p dominate q ,add q to solution p domination list  
 solution\_p.domination\_list[solution\_q.index] = solution\_q  
 elif solution\_q.this\_dominates\_(solution\_p):  
 # if q dominate p add 1 for p domination counter  
 solution\_p.domination\_counter = solution\_p.domination\_counter + 1  
 if solution\_p.domination\_counter == 0:  
 solution\_p.rank = 1  
  
 if (fronts.get(1) == None):  
 fronts[1] = list()  
 fronts[1].append(solution\_p)  
 i = 1  
  
 while len(fronts[i]) != 0:  
 front = list()  
  
 # group in to pareto fronts  
 for solution\_p in fronts[i]:  
 for solution\_q in solution\_p.domination\_list.values():  
  
 solution\_q.domination\_counter = solution\_q.domination\_counter - 1  
 if solution\_q.domination\_counter == 0:  
 solution\_q.rank = i + 1  
 front.append(solution\_q)  
 i = i + 1  
  
 fronts[i] = front  
 del fronts[i]  
  
 return fronts  
  
  
# ranking with in the pareto front  
def crowding\_distance\_calculation(front: List[SolutionMetrics], objectives\_max, objectives\_min):  
 for objective\_index, objective in front[0].objectives.items():  
 front.sort(key=lambda x: x.objectives[objective\_index], reverse=True)  
 front[0].objectives\_cd[objective\_index] = front[len(front) - 1].objectives\_cd[objective\_index] = float('inf')  
  
 for index in range(1, len(front) - 1):  
 front[index].objectives\_cd[objective\_index] = float('inf') if (objectives\_max[objective\_index] -  
 objectives\_min[objective\_index]) == 0 else \  
 front[index].objectives\_cd[objective\_index] + (  
 front[index - 1].objectives[objective\_index] - front[index + 1].objectives[objective\_index]) / (  
 (objectives\_max[objective\_index] - objectives\_min[objective\_index]))  
  
 for solutions in front:  
 solutions.calculate\_tcd()  
  
  
# retrieving all the objective min and max values  
def min\_max\_objectives(solutions: List[SolutionMetrics]):  
 # interate all objectives  
 for objective\_index, value in OBJECTIVES.items():  
 if value == 1:  
 # retrieve each objective min and max values  
 min\_t = min(solutions, key=lambda x: x.objectives[objective\_index]).objectives[objective\_index]  
 SolutionMetrics.min\_objectives[objective\_index] = min\_t  
 max\_t = max(solutions, key=lambda x: x.objectives[objective\_index]).objectives[objective\_index]  
 SolutionMetrics.max\_objectives[objective\_index] = max\_t  
  
 return SolutionMetrics.min\_objectives, SolutionMetrics.max\_objectives  
  
  
# ranking solutions with non dominating sorting and crowding distance measurement  
def nd\_sort\_2d(solutions: List[SolutionMetrics]):  
 min, max = min\_max\_objectives(solutions)  
 fronts = fast\_non\_dominate\_sort(solutions)  
 for front in fronts.values():  
 crowding\_distance\_calculation(front, max, min)  
 front.sort(key=lambda x: x.tcd, reverse=True)  
 return fronts  
  
  
def nd\_sort\_1d(solutions: List[SolutionMetrics]) -> List[SolutionMetrics]:  
 fronts = nd\_sort\_2d(solutions)  
 ranked\_list = list()  
 for front in fronts.values():  
 for solution in front:  
 ranked\_list.append(solution)  
 return ranked\_list

**Appendix D: Implementation of Constraints functions**

import numpy as np  
# 0.8 denote by 80%  
  
def check\_constraint\_for\_chromosome(chromosome, cluster, cpu\_limit\_precentage, memory\_limit\_precentage,  
 storage\_limit\_precentage):  
 #check all the microservices are whether represent or not  
 deployment\_matrix = chromosome.reshape(len(cluster.pods), len(cluster.nodes))  
 for genes in chromosome:  
 if genes < 0 :  
 return False  
  
 if 0 in np.sum(deployment\_matrix,axis=1):  
 return False  
  
 #check whether the node resource power is enough or not  
 for nodeKey in cluster.nodes:  
 memory = 0  
 cpu = 0  
 storage = 0  
 for podkey in cluster.pods:  
 memory = memory + cluster.pods.get(podkey).memory \* deployment\_matrix[podkey][nodeKey]  
 cpu = cpu + cluster.pods.get(podkey).cpu \* deployment\_matrix[podkey][nodeKey]  
 storage = storage + cluster.pods.get(podkey).storage \* deployment\_matrix[podkey][nodeKey]  
 # return false when microservice resource consumption is greater than the resource power  
 if (cluster.nodes.get(nodeKey).memory \* memory\_limit\_precentage <= memory or cluster.nodes.get(  
 nodeKey).cpu \* cpu\_limit\_precentage <= cpu or cluster.nodes.get(  
 nodeKey).storage \* storage\_limit\_precentage <= storage):  
 return False  
 return True  
  
def check\_requirement\_instances\_for\_chromosome(chromosome, cluster):  
 deployment\_matrix = chromosome.reshape(len(cluster.pods), len(cluster.nodes))  
 np.sum(deployment\_matrix, axis=1)  
 return 0  
  
def check\_static\_number\_of\_instances(chromosome,cluster,existing\_chromosome,microservice\_auto\_scalement\_false\_index):  
 deployment\_matrix = chromosome.reshape(len(cluster.pods), len(cluster.nodes))  
 service\_replica\_array=np.sum(deployment\_matrix, axis=1)  
 for index in microservice\_auto\_scalement\_false\_index:  
 if service\_replica\_array[index] != existing\_chromosome[index]:  
 return False  
 return True  
  
#modify chromosome when exceed the maximum limitation of number of instances  
#this function is created for master service instances  
#master service always need only one service  
def modify\_chromosome(chromosome, cluster, master\_pod\_indexes):  
 number\_of\_nodes = len(cluster.nodes)  
 number\_of\_pods = len(cluster.pods)  
 deployment\_matrix = chromosome.reshape(number\_of\_pods, number\_of\_nodes)  
 each\_sum\_of\_pod\_instances = np.sum(deployment\_matrix, axis=1)  
  
 for pod\_index in master\_pod\_indexes:  
 each\_pod\_instances=deployment\_matrix[pod\_index][:]  
 probability = list()  
 for node\_index in range(0,number\_of\_nodes):  
 probability.append(each\_pod\_instances[node\_index]/each\_sum\_of\_pod\_instances[pod\_index])  
  
 selected\_index=np.random.choice(number\_of\_nodes, 1, replace=False, p=probability)  
 deployment\_matrix[pod\_index][:]=np.zeros((number\_of\_nodes,), dtype=int)  
 deployment\_matrix[pod\_index][selected\_index]=1  
  
 return deployment\_matrix.reshape(1,number\_of\_nodes \*number\_of\_pods)

**Appendix E: Implementation of Solution Service Class**

from codecs import ignore\_errors  
  
import simplejson as json  
from src.config\_runner import Config  
import pandas as pd  
from src.services.pandas\_query\_service import \*  
from src.models.constant import \*  
import numpy as np  
  
  
# this service is resposible for retrieve solution  
class SolutionService:  
 \_\_instance = None  
  
 # retrieve all solution  
 def read\_all\_solutions(self):  
 json\_str = None  
 with open(self.solution\_file\_path) as file:  
 data = json.load(file)  
 json\_str = json.dumps(data)  
 return json\_str  
  
 # retrieve solution according to option such as best\_performance,best\_availability  
 def solutions\_get\_from\_query(self, solution\_option):  
  
 with open(self.options\_file\_path) as json\_file:  
 data = json.load(json\_file)  
 print('data', data.keys())  
 solutions = data[solution\_option]  
 print('best', type(solutions))  
  
 if solutions is not None:  
 return json.dumps(solutions, ignore\_nan=True)  
 else:  
 return json.dumps([])  
  
 # retrieve solution by index  
 def solution(self, index):  
  
 json\_str = None  
 with open(self.options\_file\_path) as file:  
 data = pd.read\_json(file, orient='records')  
 tmp\_pd = data[data.index == int(index)]  
  
 json\_str = tmp\_pd.to\_json(orient='records')  
  
 return json\_str  
  
 # retrieve solution by generation and index within the generation  
 def solutiongen(self, index, generation):  
  
 json\_str = None  
 # search solution on cache file for retrieving  
 with open(self.solution\_cache\_file\_path) as file:  
 data = pd.read\_json(file, orient='records')  
 pqs = PandasQueryService(data)  
 # pqs.set\_query(SOLUTION\_GET\_BY\_GEN\_AND\_INDEX,gen=generation,index=index)  
 pqs.set\_query(SOLUTION\_GET\_BY\_GEN\_AND\_INDEX)  
 tmp\_pd = pqs.execute(gen=generation, index=index)  
 if tmp\_pd.shape[0] >= 1:  
 json\_str = tmp\_pd.to\_json(orient='records')  
 else:  
 with open(self.solution\_file\_path) as file:  
 data = pd.read\_json(file, orient='records')  
 pqs = PandasQueryService(data)  
 pqs.set\_query(SOLUTION\_GET\_BY\_GEN\_AND\_INDEX)  
 tmp\_pd = pqs.execute(gen=generation, index=index)  
 json\_str = tmp\_pd.to\_json(orient='records')  
  
 return json\_str  
  
 # retrieve solution by generation and index within the generation without cache  
 def solutiongen2d(self, index, generation):  
  
 with open(self.solution\_file\_path) as file:  
 data = pd.read\_json(file, orient='records')  
 pqs = PandasQueryService(data)  
 pqs.set\_query(SOLUTION\_GET\_BY\_GEN\_AND\_INDEX)  
 tmp\_pd = pqs.execute(gen=generation, index=index)  
  
 json\_str = tmp\_pd.to\_json(orient='records')  
  
 return json\_str  
  
 # save most important solution in local file  
 def save\_all\_solutions\_to\_file(self):  
 with open(self.solution\_file\_path) as file:  
 data = pd.read\_json(file, orient='records')  
 #  
 pqs = PandasQueryService(data)  
 solution\_option = {BEST\_PERFORMANCE\_OPTION\_STR: pqs.set\_query(BEST\_PERFORMANCE\_OPTION\_STR).execute().iloc[  
 0:self.max\_rows].to\_dict('records'),  
 BEST\_AVAILABILITY\_OPTION\_STR: pqs.set\_query(BEST\_AVAILABILITY\_OPTION\_STR).execute().iloc[  
 0:self.max\_rows].to\_dict('records'),  
 OPTIMAL\_OPTION\_STR: pqs.set\_query(OPTIMAL\_OPTION\_STR).execute().iloc[  
 0:self.max\_rows].to\_dict('records'),  
 COST\_EFFECTIVE\_OPTION\_STR: pqs.set\_query(COST\_EFFECTIVE\_OPTION\_STR).execute().iloc[  
 0:self.max\_rows].to\_dict('records'),  
 EXISTING\_DEPLOYMENT\_STR: pqs.set\_query(EXISTING\_DEPLOYMENT\_STR).execute().to\_dict(  
 'records')  
  
 }  
  
 solution\_list\_cache = list()  
 for items in solution\_option.values():  
 solution\_list\_cache.extend(items)  
 with open(self.solution\_cache\_file\_path, 'w') as outfile:  
 json.dump(solution\_list\_cache, outfile)  
 print("All Solution Saved on cache on ", self.solution\_cache\_file\_path)  
  
 with open(self.options\_file\_path, 'w') as outfile:  
 json.dump(solution\_option, outfile)  
 print("All Solution Saved on ", self.options\_file\_path)  
  
 def \_\_init\_\_(self):  
 *""" Virtually private constructor. """* if SolutionService.\_\_instance != None:  
 raise Exception("This class is a singleton!")  
 else:  
 config = Config()  
 self.solution\_file\_path = config.config\_data.get('file\_path', 'solutions')  
 self.options\_file\_path = config.config\_data.get('file\_path', 'options')  
 self.solution\_cache\_file\_path = config.config\_data.get('file\_path', 'cache')  
 self.max\_rows = 40  
 SolutionService.\_\_instance = self  
  
 @staticmethod  
 def get\_instance():  
  
 if SolutionService.\_\_instance == None:  
 return SolutionService()  
 else:  
 SolutionService.\_\_instance

**Appendix E: Implementation of Pandas Query Class**

from src.models.constant import \*  
  
  
# this class is used to genrate pandas dataframe queries  
class PandasQueryService:  
 def \_\_init\_\_(self, data\_frame):  
 self.data\_frame = data\_frame  
 self.query = None  
  
 def set\_query(self, option):  
 switcher = {  
 'best\_lowest\_latency': self.\_\_best\_lowest\_latency\_query,  
 BEST\_PERFORMANCE\_OPTION\_STR: self.\_\_best\_performance\_query,  
 'best\_availability\_latency': self.\_\_best\_availability\_latency\_query,  
 BEST\_AVAILABILITY\_OPTION\_STR: self.\_\_best\_availability\_query,  
 COST\_EFFECTIVE\_OPTION\_STR: self.\_\_cost\_effective\_query,  
 OPTIMAL\_OPTION\_STR: self.\_\_optimal\_query,  
 EXISTING\_DEPLOYMENT\_STR: self.\_\_existing\_solution\_query,  
 SOLUTION\_GET\_BY\_GEN\_AND\_INDEX: self.\_\_get\_solution\_by\_gen\_and\_index\_query,  
 'all': self.\_\_all  
 }  
 if switcher.get(option) is not None:  
 self.query = switcher.get(option)  
 # self.parameters = kwargs  
 else:  
 self.query = self.\_\_all  
 return self  
  
 def execute(self, \*\*kwargs):  
 if self.query is None:  
 raise Exception("No Query set!")  
 else:  
 func = self.query  
  
 return func(\*\*kwargs)  
  
 def \_\_best\_lowest\_latency\_query(self):  
 column = self.data\_frame["fitness\_performance"]  
 max\_value = column.max()  
 df = self.data\_frame[self.data\_frame.fitness\_performance >= max\_value]  
 return df  
  
 def \_\_best\_availability\_latency\_query(self):  
 column = self.data\_frame["fitness\_availability"]  
 max\_value = column.max()  
 tmp\_pd = self.data\_frame[self.data\_frame.fitness\_availability >= max\_value]  
 column\_ratio = tmp\_pd['fitness\_ratio']  
 max\_ratio\_value = column\_ratio.max()  
 df = tmp\_pd[tmp\_pd.fitness\_ratio >= max\_ratio\_value]  
 return df  
  
 def \_\_best\_performance\_query(self):  
 tmp\_pd1 = self.data\_frame[self.data\_frame.fitness\_availability >= 0]  
 column = tmp\_pd1["fitness\_performance"]  
 max\_value = column.max()  
 tmp\_pd2 = tmp\_pd1[self.data\_frame.fitness\_performance >= max\_value]  
 df = tmp\_pd2.sort\_values(by=['fitness\_ratio', 'fitness\_availability'], ascending=[False, False])  
 return df  
  
 def \_\_best\_availability\_query(self):  
 tmp\_pd1 = self.data\_frame[self.data\_frame.fitness\_availability >= 0]  
 df = tmp\_pd1.sort\_values(  
 by=['fitness\_performance', 'fitness\_ratio', 'fitness\_scale', 'fitness\_availability'],  
 ascending=[False, False, False, False])  
 return df  
  
 def \_\_cost\_effective\_query(self):  
 tmp\_pd1 = self.data\_frame[self.data\_frame.fitness\_availability >= 0]  
 column\_cost = tmp\_pd1["fitness\_cost"]  
 low\_cost = column\_cost.max()  
 tmp\_pd2 = tmp\_pd1[tmp\_pd1.fitness\_cost == low\_cost]  
 df = tmp\_pd2.sort\_values(by=['fitness\_ratio', 'fitness\_availability'], ascending=[False, False])  
 return df  
  
 def \_\_all(self):  
 return self.data\_frame  
  
 def \_\_optimal\_query(self):  
 tmp\_pd1 = self.data\_frame[self.data\_frame.fitness\_availability >= 0]  
 df = tmp\_pd1.sort\_values(by=['fitness\_ratio', 'fitness\_performance', 'fitness\_scale', 'fitness\_availability'],  
 ascending=[False, False, False, False])  
 return df  
  
 def \_\_existing\_solution\_query(self):  
 tmp\_pd = self.\_\_get\_solution\_by\_gen\_and\_index\_query(0, 0)  
 return tmp\_pd  
  
 def \_\_get\_solution\_by\_gen\_and\_index\_query(self, gen, index):  
 tmp\_pd1 = self.data\_frame[self.data\_frame['generation'] == gen]  
 tmp\_pd = tmp\_pd1[tmp\_pd1['index'] == index]  
 return tmp\_pd

**Appendix F: Implementation of HPA Component**

import math  
  
  
# HPA algorithm is used to predict desired replica set  
class HPAAlgorithm:  
 # cpu\_limit\_precentage, memory\_limit\_precentage should be between 0-1.  
 def \_\_init\_\_(self, cpu\_limit\_precentage, memory\_limit\_precentage):  
 self.cpu\_limit\_precentage = cpu\_limit\_precentage  
 self.memory\_limit\_precentage = memory\_limit\_precentage  
  
 # HPA algorithm  
 def \_individual\_pod\_desired\_replicas(self, current\_replicas, predicted\_metric\_value, desired\_metric\_value):  
 return math.ceil(current\_replicas \* (predicted\_metric\_value / desired\_metric\_value))  
  
 # calculate number of required instances of all services  
 def calculate\_requirement\_replicaset(self, pod\_dictionary\_map, predicted\_cpu\_metrics, predicted\_memory\_metrics,  
 type="cpu"):  
 requirement\_replicaset\_b\_on\_cpu = list()  
 requirement\_replicaset\_b\_on\_memory = list()  
 requirement\_replicaset\_b\_on\_cpu\_and\_memory = list()  
 for index, pod in pod\_dictionary\_map.items:  
 pod\_replica\_b\_on\_cpu = self.\_individual\_pod\_desired\_replicas(pod.current\_replica, predicted\_cpu\_metrics,  
 pod.cpu \* self.cpu\_limit\_precentage)  
 requirement\_replicaset\_b\_on\_cpu.append(pod\_replica\_b\_on\_cpu  
 )  
  
 pod\_replica\_b\_on\_memory = self.\_individual\_pod\_desired\_replicas(pod.current\_replica,  
 predicted\_memory\_metrics,  
 pod.cpu \* self.memory\_limit\_precentage)  
 requirement\_replicaset\_b\_on\_memory.append(pod\_replica\_b\_on\_memory)  
 max\_replica = pod\_replica\_b\_on\_cpu if pod\_replica\_b\_on\_cpu >= pod\_replica\_b\_on\_memory else pod\_replica\_b\_on\_memory  
 requirement\_replicaset\_b\_on\_cpu\_and\_memory.append(max\_replica)  
 switcher = {  
 "cpu": requirement\_replicaset\_b\_on\_cpu,  
 "memory": requirement\_replicaset\_b\_on\_memory,  
 "cpu\_memory": requirement\_replicaset\_b\_on\_cpu\_and\_memory  
 }  
 return switcher.get(type)