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

De Silva N.

IT17006880

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 AN OPTIMAL MICROSERVICE GOVERNANCE**

De Silva N.

IT17006880

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 my 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:**

|  |  |  |
| --- | --- | --- |
| **IT17006880** | **De Silva N.** |  |

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

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

**Date: …………………………….**

**Abstract**

One of the many advantages of using Kubernetes for microservice deployments is the ability to scale cluster resources based on traffic. This ability of Kubernetes to automatically scale cluster resources thereby enables developers and system administrators to reduce costs and make use of their deployed microservices more effectively and efficiently. However, even though the autoscaling tools such as the Horizontal Pod Autoscaler (HPA) provided by Kubernetes help developers and system administrators to effectively utilize their microservice deployments as previously stated, they also possess some inherent drawbacks as well. The main drawback among them being that these autoscaling policies adopted are primarily local and rule-based auto-scaling techniques which do not consider the overall impact of the auto-scaling policy in order to truly optimize the existing resources in a particular deployment. This research thereby aims to address this issue prevalent through aiming to provide a data science-based approach with the help of machine learning-based time series prediction measures along with centrality analysis performed on microservice co-dependency network to facilitate the development of an improved auto-scaling policy which takes into account the global importance of a particular microservice.

**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](#_Toc51619542)

[**Abstract** ii](#_Toc51619543)

[**Acknowledgement** iii](#_Toc51619544)

[**Table of Contents** iv](#_Toc51619545)

[**List of Figures** vi](#_Toc51619546)

[**List of Tables** vii](#_Toc51619547)

[**List of Abbreviations** viii](#_Toc51619548)

[**List of Appendices** ix](#_Toc51619549)

[**1.0** **INTRODUCTION** 1](#_Toc51619550)

[**1.1 Background and Literature** 3](#_Toc51619551)

[**1.1.1 Centrality Measures** 7](#_Toc51619552)

[**1.2 Research Gap** 9](#_Toc51619553)

[**2.0** **RESEARCH PROBLEM** 11](#_Toc51619554)

[**3.0** **OBJECTIVES** 12](#_Toc51619555)

[**3.1 Main Objective** 12](#_Toc51619556)

[**3.2 Specific Objectives** 12](#_Toc51619557)

[**4.0** **METHODOLOGY** 13](#_Toc51619558)

[**4.1 Requirement Gathering** 13](#_Toc51619559)

[**4.1.1 Past Research Analysis** 13](#_Toc51619560)

[**4.1.2 Identifying Existing Systems** 13](#_Toc51619561)

[**4.2 Feasibility Study** 14](#_Toc51619562)

[**4.2.1 Technical Feasibility** 14](#_Toc51619563)

[**4.2.2 Schedule Feasibility** 14](#_Toc51619564)

[**4.2.3 Economic Feasibility** 14](#_Toc51619565)

[**4.3 Requirement Analysis** 15](#_Toc51619566)

[**4.4 System Analysis** 16](#_Toc51619567)

[**4.5 System Development and Implementation** 20](#_Toc51619568)

[**4.5.1 Prediction of Load-based Inter - microservice Dependency Measures / Pod Resource Utilization Metrics** 21](#_Toc51619569)

[**4.5.3 Calculation of Centrality Measures** 23](#_Toc51619570)

[**4.6** **Project Requirements** 24](#_Toc51619571)

[**4.6.1 Functional requirements** 24](#_Toc51619572)

[**4.6.2 Non-Functional Requirements** 24](#_Toc51619573)

[**4.7 Commercialization** 25](#_Toc51619574)

[**5.0** **TESTING AND IMPLEMENTATION RESULTS AND DISCUSSION** 26](#_Toc51619575)

[**5.1 Testing** 26](#_Toc51619576)

[**5.1.1 Selection of Optimal Prediction Model** 26](#_Toc51619577)

[**5.1.2 Testing of Developed Solution** 29](#_Toc51619578)

[**5.2 Test Results** 31](#_Toc51619579)

[**5.2.1 Prediction of Resource Utilization Metrics** 31](#_Toc51619580)

[**5.2.2 Prediction of Load - based Inter- microservice Link Dependency Measures** 34](#_Toc51619581)

[**5.2.3 Centrality Evaluation** 37](#_Toc51619582)

[**5.2.3 Determination of Optimal Microservice Instance Levels to the Creation of Improved Autoscaling Policies through Integration with the Optimization Algorithm** 38](#_Toc51619583)

[**5.3 Research Findings and Discussion** 40](#_Toc51619584)

[**6.0 CONCLUSION** 41](#_Toc51619585)

[**References** 42](#_Toc51619586)

[**Appendix** 45](#_Toc51619587)

[**Appendix A: Coding Solution for Prediction of Load-based Inter - microservice Dependency Measures / Pod Resource Utilization Metrics** 45](#_Toc51619588)

[**Appendix B: Coding Solution for Creation of Current Co – dependency Map and Evaluation of Centrality Measures** 52](#_Toc51619589)

[**Appendix C: Coding Solution for Creation of Predicted Co – dependency Map and Evaluation of Centrality Measures** 54](#_Toc51619590)

**List of Figures**

|  |  |  |
| --- | --- | --- |
|  |  | Page |
| Figure 1.1 | MAPE control loop | 4 |
| Figure 4.1 | Load prediction and centrality analysis component in the context of the developed governance model | 16 |
| Figure 4.2 | Load prediction and centrality analysis component | 17 |
| Figure 4.3 | High-level diagram of load prediction and centrality analysis component | 19 |
| Figure 4.4 | Overview of the prediction process for load-based inter-microservice link dependency measures/pod resource utilization metrics | 21 |
| Figure 4.5 | Processes followed in the calculation of centrality measures | 23 |
| Figure 5.1 | Evaluated prediction models | 28 |
| Figure 5.2 | Structure of dataset containing CPU resource utilization metrics | 30 |
| Figure 5.3 | Structure of dataset containing load-based inter-microservice link dependency measures | 30 |
| Figure 5.4 | MAPE, RMSE, SMAPE and MASE values - resource utilization prediction | 33 |
| Figure 5.5 | 1-hour forecast of CPU utilization metrics | 33 |
| Figure 5.6 | MAPE, RMSE, SMAPE and MASE values - inter-microservice link dependency measures | 34 |
| Figure 5.7 | 1-hour forecast of load-based inter-microservice link dependency measures | 35 |
| Figure 5.8.1 | Predicted microservice co-dependency network at time t+12 | 35 |
| Figure 5.8.2 | Microservice co-dependency network at time t | 36 |
| Figure 5.9 | Figure of the table depicting the comparison of current and forecasted centrality measures of co-dependency network | 37 |
| Figure 5.10 | Structure of sample JSON input | 39 |
| Figure 5.11 | Figure of the table depicting the comparison of current and optimal microservice instance numbers | 39 |

**List of Tables**

|  |  |  |
| --- | --- | --- |
|  |  | Page |
| Table 4.1 | HPA Algorithm Term Definitions | 19 |
| Table 4.2 | Tools and technology | 20 |
| Table 5.1 | MAPE and RMSE values of evaluated prediction models | 27 |
| Table 5.2 | Model Parameters – Resource utilization prediction | 32 |
| Table 5.3 | Model Parameters – Inter-microservice link dependency measures | 34 |

**List of Abbreviations**

|  |  |
| --- | --- |
| Abbreviation | Description |
| AKS | Azure Kubernetes Services |
| ARIMA | Auto-Regressive Integrated Moving Average |
| AWS | Amazon Web Service |
| CSV | Comma Separated Values |
| EMD | Empirical Mode Decomposition |
| ES-RNN | Empirical Smoothing – Recurrent Neural Network |
| HPA | Horizontal Pod Autoscaler |
| JSON | JavaScript Object Notation |
| MAPE 1 | Monitor Analyze Plan and Execute |
| MAPE 2 | Mean Absolute Percentage Error |
| MASE | Mean Absolute Scaled Error |
| QoS | Quality of Service |
| RMSE | Root Mean Square Error |
| SLA | Service Level Agreement |
| SMAPE | Scaled Mean Absolute Percentage Error |
| TCN | Temporal Convolutional Network |
| VM | Virtual Machine |
| VPA | Vertical Pod Autoscaler |
|  |  |

**List of Appendices**

|  |  |  |
| --- | --- | --- |
| Appendix | Description | Page |
| Appendix A | Coding Solution for Prediction of Load-based Inter - microservice Dependency Measures / Pod Resource Utilization Metrics | 45 |
| Appendix B | Coding Solution for Creation of Current Co – dependency Map and Evaluation of Centrality Measures | 52 |
| Appendix C | Coding Solution for Creation of Predicted Co – dependency Map and Evaluation of Centrality Measures | 54 |

1. **INTRODUCTION**

The introduction of Kubernetes brought about a revolutionary approach to the deployment of microservice applications throughout the world by providing an effective solution to the orchestration containerized microservices. As a result, in conjunction with the vast array of services and features such as service discovery and load balancing, storage orchestration, and self-healing mechanisms [1] offered through the Kubernetes framework, many organizations throughout the world are increasingly adopting Kubernetes as an effective solution to govern their microservice deployments, thereby establishing Kubernetes among the foremost microservice deployment tools currently available.

In this regard, among the many services offered by Kubernetes, the ability to automatically scale based on workloads is most prominent. This process, known as autoscaling, refers to the automatic scaling of computational resources based on the required workloads. This feature is particularly crucial in the context of microservices, primarily since the decentralized and modular design approach to microservices entails that workloads across microservices are dynamic, where at a specific instance of time, a particular microservice may require the need to face varied workload intensities compared to its counterpart services. Hence, in order to achieve optimal performance of a microservice deployment, an effective autoscaling policy must be in effect such that users could make optimal use of resources while also ensuring minimal costs are incurred.

However, the above-mentioned autoscaling feature primarily adopts local and rule-based auto-scaling techniques to dynamically manage the number of microservice resources in a particular deployment. These rule-based techniques primarily utilize a limited amount of infrastructure-level metrics such as CPU utilization to determine thresholds for autoscaling policies which may, in turn, lead to a lack of global awareness when making effective autoscaling decisions due to the use of a limited amount of metrics considered in autoscaling processes which in turn, may cause under or over-provisioning of resources and ultimately result in ineffective autoscaling. Therefore, in order to truly optimally utilize the existing resources in a microservice-based deployment, it may be necessary to have a holistic perspective of how each microservice is utilized. In this regard, the presence of a holistic autoscaling policy is a prime requirement in the creation of such a holistic perspective.

This research thereby utilizes a data-science based approach thorough incorporation of load prediction and evaluation of centrality measures performed on microservice networks in conjunction with a developed optimization algorithm to the creation of holistic autoscaling policies on microservice deployments.

The proceeding sections in this document discuss key aspects in the development of the above-mentioned solution. In this regard, this document provides a thorough analysis pertaining to key topics which include the background, research gap as well as the research problem addressed by the developed solution, along well an in-depth discussion and analysis of results obtained. Lastly, the document also discusses future work and suggested improvements to the developed solution along with an overall conclusion of the research conducted.

**1.1 Background and Literature**

According to popular cloud providers such as AWS, auto-scaling is defined as a cloud computing service feature that allows AWS users to automatically launch or terminate virtual instances based on defined policies, health status checks, and schedules [2] whereas other publications such as [3] define autoscaling as a mechanism of dynamically acquiring or releasing resources to meet QoS requirements. However, one of the most in-depth descriptions of the definition of autoscaling is provided by the publication [4]. This publication manages to summarize the key features of autoscaling, which can be listed as follows.

* Ability to scale out (addition of extra unused resources during increased demand) and scale in (removal of extra unused resources during reduced demand)
* Capability of setting rules for scaling in and out
* Automatically detect and remove unhealthy instances

Due to these features, autoscaling is widely adopted in a variety of cloud platforms as well in other technologies to scale a wide variety of resources ranging from VMs to other resources such as pods in orchestration tools such as Kubernetes. Additionally, due to the elastic nature which allows resources to be provisioned more conservatively, cloud providers can serve more customers with the same infrastructure.

Furthermore, the autoscaling process for the resources mentioned above can be performed using a variety of techniques, as described in [5]. However, among them, rule-based autoscaling is the technique most commonly used by many cloud vendors today [6] as well as in other orchestration platforms such as Kubernetes, evident through its HPA and VPA autoscaling tools.

Rule-based autoscaling involves defining the conditions under which capacity will be added to or removed from a cloud-based system, in order to satisfy the objectives of the application owner [6]. Therefore, for a rule-based autoscaling approach to be effective, the application provider has to specify upper and lower bounds, which are usually defined through a performance metric such as CPU utilization. This approach to rule-based approach to autoscaling has therefore defined rule-based autoscaling as a more reactive approach to provision resources since the autoscaling process occurs when the defined thresholds and bounds set, are exceeded.

Another key characteristic that is present in most rule-based autoscaling policies is the adaptation of the MAPE 1 (Monitor, Analyze, Plan, and Execute) control loop reference model used in various autonomic computing systems. This is reference model is primarily adopted in orchestration tools such as Kubernetes in the creation of guidelines of self-adaptive software systems [7].



Figure 1.1: MAPE 1 control loop [5]

However, there are some issues prevalent in the adaptation of this model along with the rule-based autoscaling policies adopted. These include issues ranging from the lack of adaptability to dynamic workloads faced, which result in under or over-provisioning of resources and loss in the QoS, to issues such as the response delay faced in resource creation. As a result, throughout the years, various research publications have proposed numerous approaches ranging from proactive autoscaling through resource prediction to other approaches such as performance modeling, in order to combat these prevalent issues.

Therefore, before moving on with the implementation of this research, it is vital a thorough analysis is conducted on some of the existing research carried out in order to solve some of the current issues found these local and rule-based (reactive) autoscaling policies adopted in Kubernetes as well as other similar autoscaling technologies.

The research conducted by A. Zhao and his team [8], uses a resource prediction model based on Kubernetes container auto-scaling technology and makes use of a combination of ARIMA model as well as Empirical Mode Decomposition (EMD) to predict the resource usage and thereby proactively auto-scale the number of pod replicas. This research was primarily conducted aiming to solve the response delay found on the existing autoscaling strategy in Kubernetes as well as manages to provide a detailed insight into some of the current issues found in the Kubernetes HPA and its implementation of rule-based autoscaling policies. However, even though the use of this combination of the above-mentioned techniques proved to produce more accurate forecasting compared to the traditional ARIMA model used, the use of EMD can be quite an inefficient and time-consuming process, as discussed in publications such as [9].

The research conducted by H. Zhao and his team [10] makes use of a double exponential smoothing algorithm for predicting the resource consumption and thereby determining the predicted pods. However, a key point to note in the research conducted is the fact that the predictive algorithm developed, only makes use of the predicted values in the scaling-up process and does not utilize the prediction algorithm in the scaling down process.

Y. Meng and his team [11] makes use of a data science-based approach to predict resource utilization. This research proposes the use of CRUPA, a resource utilization prediction algorithm based on the ARIMA time series analysis model combined with docker container techniques. This prediction algorithm was quite successful and had a high prediction accuracy with an average forecast error of about 6.5%. However, a key fact noted in this approach used is that this model developed, is primarily based on microservice containers, and does not make use of the existing auto-scaling tools provided by Kubernetes for the scaling process.

Publications such as [12] propose a system architecture for Docker containerized applications with an auto-scaler based on a machine learning model. However, here too, the proposed auto-scaler is primarily based on microservice containers and does not integrate with existing Kubernetes autoscaling tools similar to [11]. Other researches based on proactive containerized auto-scaling techniques similar to the previously mentioned researches stated include researches such as [13] and [14].

However, there have also been some other approaches proposed for improved autoscaling policies rather than the proactive solution-based approaches mentioned above, in order to solve the issues of rule-based autoscaling policies. Key researches include the dynamic multi-level autoscaling method proposed by the publication [15] which makes use of infrastructure as well as application-level monitoring data to dynamically specify thresholds for autoscaling. Other researches in this category include the heterogeneity-aware auto-scaling strategy [16] and the performance modeling-based approach for the rule-based auto-scaling technique proposed by the authors of [6].

Some other key related researches in this regard include the framework for autoscaling proposed by Al-Sharif [17] used in the provisioning of sufficient VMs to address the changing resource requirements in cloud environments, the master-slave autoscaling architecture for containerized applications proposed by Kukade and Kale [18], the container-based elastic cloud platform named Do Cloud proposed by Kan [19] as well as the Platform Insights framework proposed in [20] which makes use of a combination of proactive and reactive models in order to scale resources.

From the above-mentioned researches, it is clear that even though the solutions for improved autoscaling strategies proposed by these researches were quite effective, they were only focused on providing effective solutions for the localized rule-based autoscaling policies which were primarily based on infrastructure level metrics such as CPU utilization.

**1.1.1 Centrality Measures**

The concept of centrality stems from graph theory and network analysis and typically deals with the identification of the level of importance nodes in a given network. Centrality analysis is thereby a key player in many industrial and practical applications and utilized in a multitude of fields such as computer science, medicine, and economics.

A key focus of this research pertains to the modeling and evaluating of microservice networks through network analysis-based approaches. In this regard, the research conducted also focusses on the evaluation of centrality measures performed on microservice networks. The following sections below provide a brief overview of key centrality measures that could be performed on graph networks.

* + - 1. **Degree Centrality**

Degree centrality can be defined as one of the simplest centrality measures that can be evaluated in a graph network. It is defined by the number of links that incident on a node in a network and is a key centrality measure that can be utilized in the identification of highly connected nodes in a network. In simple terms, the higher the number of links connected to a particular node, the higher the degree centrality score.

* + - 1. **Closeness Centrality**

Closeness centrality is a centrality measure that defines the average length of the shortest path amongst a given node and the remaining nodes in a graph. Thereby closeness centrality is essential in the identification of the level of influence of a given node with respect to its closeness to other nodes.

* + - 1. **Betweenness Centrality**

Betweenness centrality quantifies the number of times a given node lies on the shortest path between other nodes in a graph network. Betweenness centrality can be a key centrality measure that helps in the identification of nodes that shapes the flow in information in a graph network.

* + - 1. **Eigenvector Centrality**

Eigenvector centrality quantifies the level of influence of a node is based on the number of connections (links) a node has to other nodes in a graph network. In this regard, eigenvector centrality can be defined as an extension of degree centrality through incorporating the level of connections a node possesses along with the number of links possessed by the other connected nodes.

**1.2 Research Gap**

In the identification of the research gap, the research publication [7] can be stated one the main publications that manage to provide insight into some of the key issues present in existing rule-based autoscaling strategies (particularly with respect to Kubernetes), as well as the some of the influential factors that should be considered in the development of an optimal auto-scaling strategy for Kubernetes based deployments. This publication is therefore quite influential in the identification of the research gap this research aims at fulfilling. Moreover, according to such research publications, although existing rule-based auto-scaling methods may be suitable for cloud-based applications they may result in undesirable Quality of Service (QoS) or poor resource utilization with certain dynamic workloads. This publication then goes on to highlight the fact that ensuring a favorable performance in microservice-based applications governed by the existing reactive auto-scaling rules specifically is currently a challenging issue.

Publications such as [4] also clearly state some of the key challenges that need to be addressed in current autoscaling services. According to this publication, there is a lack of autoscaling studies focusing on the service-level of autoscaling and the use of service level metrics (e.g.: - requests/transactions per unit time). Additionally, issues such as the lack of monitoring tools and aggregating metrics at the platform level and service level to support autoscaling decisions are also clearly described in this regard.

Furthermore, as described in the previous section, although many research publications provide various solutions in which to minimize issues present in rule-based autoscaling policies through the use of various measures, a majority of the proposed researches make use of a limited amount of infrastructure-level metrics (such as CPU utilization) in order to define autoscaling policies. This, in turn, may result in ineffective autoscaling since other service-level metrics that can help in the realization of a more holistic view regarding the importance of a particular microservice in a deployment are not utilized in the creation of the autoscaling policy.

Through analysis of these research publications, the apparent gap in research related to the creation and development of auto-scaling policies that make use of the global importance of a particular microservice along with the use of service-level metrics is realized.

This research thereby aims to fulfill this research gap through the inclusion of both infrastructure-level metrics as well as service-level metrics derived from monitoring solutions, for a load prediction based approach along with centrality based evaluation performed on microservice co-dependency networks, which will ultimately aid in the identification of key microservices in a particular deployment and thereby aid in the creation of a more globally aware autoscaling policy.

1. **RESEARCH PROBLEM**

The auto-scalers of container orchestration tools such as Kubernetes follow a broadly accepted reference model named MAPE 1 (Monitor, Analyze, Plan, and Execute) used in various autonomic computing systems [3,5,7]. This MAPE 1 model uses monitoring-based mechanisms to analyze relevant thresholds and thereby scale resources.

However, this approach is rule-based and, at times, due to the dynamic nature of workload that microservices have to experience, the current rule-based auto-scaling mechanisms offered are unable to adapt to these workload intensities. This prevalent issue results in over-provisioning or under-provisioning of allocated resources and ultimately results in a lower Quality of Service (QoS) experienced by users.

Issues such the response delay caused during resource creation and initialization [8,21] and the difficulties posed in configuring rule-based auto-scaling due to lack of knowledge and expertise as well as the vast configuration space involved during this process which in turn make the selection of optimal parameters and variables virtually impossible [6], further complicates the creation of effective rule-based auto-scaling policies.

In addition, due to the dynamic nature of the workload faced by microservices, it becomes necessary to make use of a wide variety of metrics in order to maintain the required SLAs [4]. Furthermore, this process is quite inefficient and usually results in ineffective resource provisioning due to the fact that most auto scalers make use of a limited amount of infrastructure-level metrics such as CPU utilization to infer autoscaling decisions. This lack of inclusion of other service-level metrics (For example, transactions per unit time) prevents and hinders effective autoscaling decision making and does not provide the necessary insight in order to visualize and understand how each microservice is utilized. Evidence regarding the importance of using higher-level metrics in the creation of autoscaling policies can be seen from reading research publications such as [15].

1. **OBJECTIVES**

**3.1 Main Objective**

The primary objective of this research is to facilitate the development of an improved auto-scaling policy for a Kubernetes - based microservice deployment, based on load prediction and centrality measures.

**3.2 Specific Objectives**

The specific objectives for the research project are as follows.

* To develop an algorithm to predict the load of a microservice and evaluate centrality measures on microservice co-dependency networks.
* To develop an auto-scaling policy without considering factors such as the location of instance creation, based on the global importance of a particular microservice, measured using the centrality measures (such as degree, betweenness, etc.) applied to the derived co-dependency network.
* To evaluate and explore the effect of centrality evaluation-based mechanisms in the creation of an optimal deployment strategy for Kubernetes deployments.

1. **METHODOLOGY**

**4.1 Requirement Gathering**

Requirement gathering was primarily performed through the analysis of published research papers as a variety of online sources. A key focus was given on the identification of existing or similar systems developed and the methodology used.

**4.1.1 Past Research Analysis**

Past research analysis was primarily performed through reading research publications mainly focused on key areas such as resource utilization prediction, short term load prediction, time series analysis, proactive auto-scaling, cloud elasticity, centrality evaluation, and machine learning models.

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 load forecasting and resource utilization prediction.

**4.1.2 Identifying Existing Systems**

Existing systems were primarily identified through referring research publications as well as referring a variety of online sources. A key focus was given in the identification of the existing feature they offered as well as the potential drawbacks in the technology and methodology used.

**4.2 Feasibility Study**

**4.2.1 Technical Feasibility**

Technical feasibility was a key factor considered in the requirements analysis phase of this research project since this project mainly focused on the development of a load prediction algorithm using machine learning technology. A key focus was given in the identification of potential system requirements as well as the required tools and technologies that may be used in the development of the prediction algorithm.

**4.2.2 Schedule Feasibility**

The schedule feasibility was also a key factor considered throughout this research. A key focus was given in the identification of possible time periods and duration to develop and implement the load prediction algorithm using the possible development tools and technology within an implementation period of about five months.

**4.2.3 Economic Feasibility**

The economic feasibility was another key factor considered before the development of this prediction algorithm. A key focus was given in the identification of the possible costs that might be incurred in the development process, as well as the costs that may be incurred in the use of the planned development tools and technologies.

**4.3 Requirement Analysis**

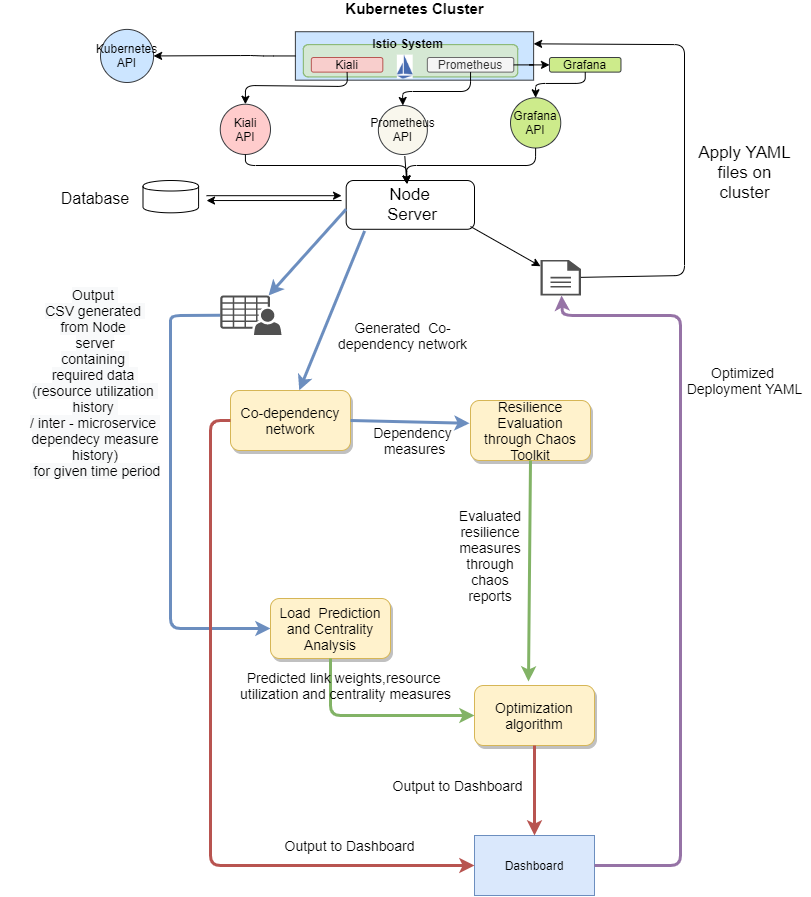
The requirement analysis phase was one of the keys phases in this research project since it enabled in the identification of a variety of factors that should be considered in the implementation process of this research.

During this process, the information gathered from the various sources during the requirement gathering phase was analyzed. As a result, the key factors related to the possible challenges that may be encountered as well as insight into the methodology and also a clear understanding of the use of possible and tools and technology were also able to be easily identified.

Furthermore, a clear idea of the scope of the research, as well as the feasibility of the project, was also able to be identified during this phase. Requirement analysis also helped in the determination of the existing research gaps as well as provide insight into the identification underlying research problem as the research.

**4.4 System Analysis**

The load prediction and centrality analysis component is a key constituent of the developed governance model and responsible for the prediction of load and evaluation and calculation of centrality measures in microservice co-dependency networks. It is incorporated into the developed governance model as depicted in Fig. 4.1 below.



Load Prediction and Centrality Analysis component

Figure 4.1: Load prediction and centrality analysis component in the context of the developed governance model

As per the figure above, the load prediction and centrality analysis component receives input from the K8Advisor server and forwards the resulting outputs to the optimization algorithm. The communication among the components in the developed governance model is facilitated through the utilization API endpoints, and in the case of the load prediction and centrality analysis component, a flask server, as depicted in Fig. 4.2 is incorporated along with the load prediction and centrality analysis component in order to expose the necessary API endpoints required by the optimization algorithm.

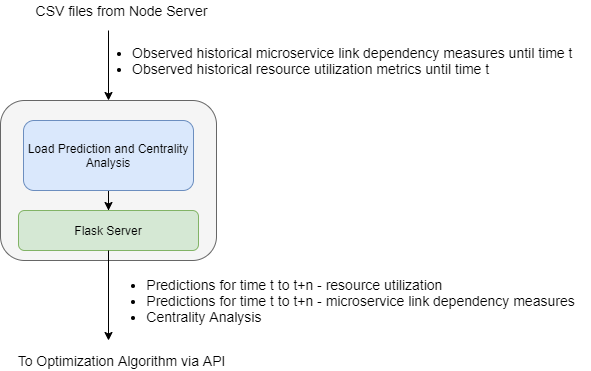


Figure 4.2: Load prediction and centrality analysis component

The primary goal of the load prediction and centrality analysis component is to facilitate the creation of holistic autoscaling and optimized deployment policies in conjunction with the optimization algorithm, through the prediction of load-based metrics and evaluation of centrality measures performed on microservice co-dependency networks.

In this regard, in order to perform these tasks, the load prediction and centrality analysis component requires two key inputs measures, as listed below.

* Historical pod CPU and memory utilization values.
* Historical inter-microservice link dependency measures in the microservice co-dependency network.

The above-listed input measures are obtained in the form of CSV files through the K8Advisor Node server. The two CSV files obtained in this regard, contain a record of observed historic metric and microservice load-based inter-microservice link dependency data pertaining to a specific time period, gathered via the monitoring solutions deployed in the Istio service mesh in the Kubernetes cluster.

Once, the required CSV files are obtained by the load prediction and centrality analysis component, the process of load prediction and centrality analysis is initiated, and the resulting outputs are forwarded to the optimization algorithm via APIs. These include;

* Predicted future resource utilization values (primarily CPU) in the form of a JSON file response.
* Predicted future microservice load-based inter-microservice link dependency measures (link weight) in the form of a JSON file response.
* Centrality measures of microservices utilizing predicted and existing microservice link dependency measures in the form of a JSON response.

The outputs above are then utilized by the optimization algorithm for the completion of the following crucial tasks.

* Proactively auto-scale the cluster based on predicted future resource utilization values.
* Determine the optimal placement of microservices utilizing the predicted future microservice load-based link dependency measures and centrality measures of microservices.

To facilitate the creation of the globalized autoscaling policy, the predicted microservice link dependency measures calculated will be utilized as a primary input to the optimization algorithm facilitate the determination of the optimal number of microservice instances whereas as previously mentioned, the predicted resource utilization measures will be used to proactively auto-scale the cluster via the optimization algorithm component of the developed governance model.

In this regard, the proactive autoscaling process utilizes a slightly altered version of the below-depicted Kubernetes HPA replica calculation algorithm [22] described as in formula (1) to predict replicas required in the future by replacing the “*currentMetricValue*” term with the "*predictedMetricValue*" term as in formula (2). Table 4.1 below describes the terms utilized in the formula (1) and (2).

Table 4.1 HPA Algorithm Term Definitions

|  |  |
| --- | --- |
| 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)

(2)

In summary, the key functionalities provided by the load prediction and centrality analysis component can be depicted as per Fig. 4.3 below.



Figure 4.3: High-level diagram of load prediction and centrality analysis component

**4.5 System Development and Implementation**

The load prediction and centrality analysis component of the developed governance model utilizes a data-science based approach in the fulfillment of its core functionalities. Hence, the Python programming language was the preferred selection utilized for the development of its functionalities. This is primarily due to the immense flexibility and adaptability possessed by the Python programming language that supports data augmentation along with the provision of added benefits such as the presence of a mature, well-developed collection of libraries that facilitate enhanced machine learning and deep learning-based programming functionalities. In this regard, Table 4.2 provided below provides a summary of key development tools and python utilized in conjunction with the Python programming language.

Table 4.2: Tools and technology

|  |  |
| --- | --- |
| Tools | * Anaconda * Jupyter Notebook |
| Python libraries | * Numpy * Scikit-learn * Keras * TensorFlow * Pandas * Matplotlib * NetworkX * Flask |

**4.5.1 Prediction of Load-based Inter - microservice Dependency Measures / Pod Resource Utilization Metrics**

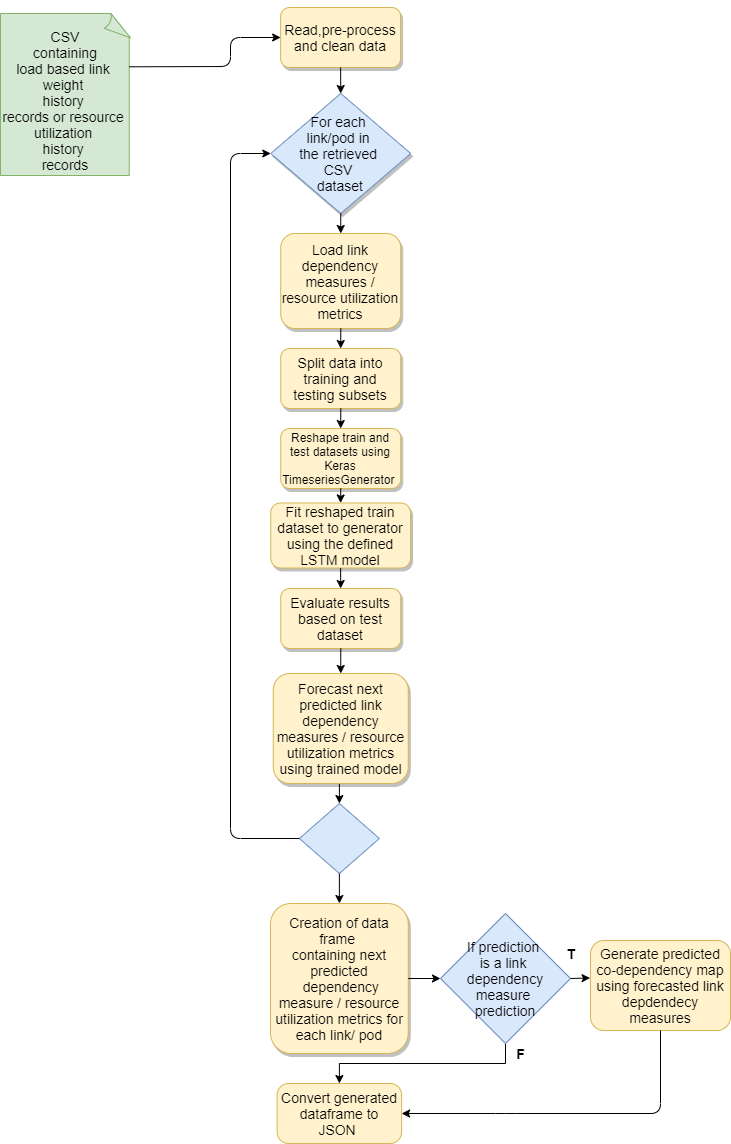


Figure 4.4: Overview of the prediction process for load-based inter-microservice link dependency measures/pod resource utilization metrics

**Note**: - See Appendix A for coding solution

The process utilized for the prediction load-based inter-microservice link dependency measures and resource utilization metrics is quite similar, due to the similarities in functional requirements. Hence, the prediction process of load-based inter-microservice link dependency measures and resource utilization metrics can be depicted as per Fig.4.4 above. However, as evident in the diagram above, the primary variation that exists among the prediction process utilized in the prediction of load-based inter-microservice link dependency measures and resource utilization metrics is the fact that there exists an additional step pertaining to the generation of the predicted microservice co-dependency map through the utilization of predicted link dependency measures.

The prediction of load-based inter-microservice link dependency measures and resource utilization metrics is performed through the application of a Long Short-Term Memory (LSTM) network in which a particular number of time steps (configured as per user requirements), are used to predict future link dependency measures/resource utilization values pertaining to the next time period. Through the application of the forecasted measures derived through the prediction process an accurate estimation of the load/resource utilization that is expected to be received by microservices in the cluster can be estimated. Furthermore, in conjunction with the optimization algorithm of the developed governance model, the load prediction and centrality analysis component facilitates the identification of key potential microservices that highly manipulate microservice placement decisions and the realization of optimal cluster performance.

**4.5.3 Calculation of Centrality Measures**

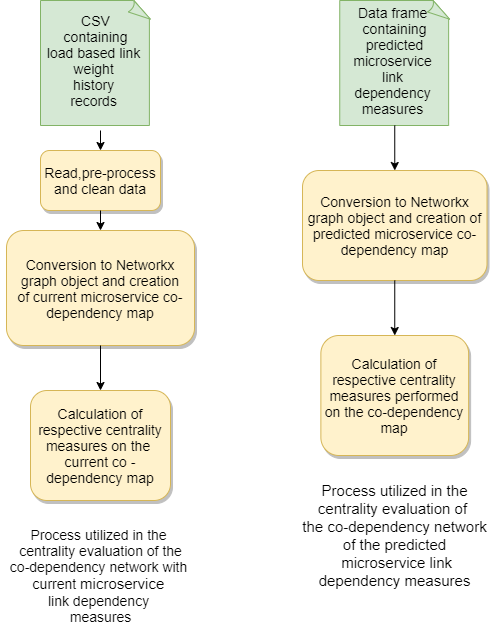


Figure 4.5: Processes followed in the calculation of centrality measures

**Note**: - See Appendix B and C for coding solution

Calculation of microservice centrality measures is also be performed within the load prediction component. Here, the microservices in the co-dependency network can be evaluated on several centrality measures to facilitate the identification of influential microservices in the cluster utilizing the current load-based microservice link dependency measures or through the predicted load-based inter-microservice link dependency measures.

In addition, the calculated centrality measures obtained in this regard, are then forwarded as inputs to the optimization algorithm, to infer autoscaling decisions through the determination of required service instance levels. In this regard, the developed governance model is expected to make use of the key centrality measures such as degree, betweenness, closeness as well as eigenvector centrality measures to facilitate the identification process of influential microservices. Figure 4.5 above depicts the process followed in the calculation of centrality measures of microservices in the co-dependency network.

* 1. **Project Requirements**

**4.6.1 Functional requirements**

The primary functional requirements aimed at fulfilling during the implementation process in this research are as follows.

* Dependency metrics for prediction should be obtained from the co-dependency network developed
* Predictions should be made ahead of time. (at least 1 – 12 hours in advance)
* The developed solution should enable the identification of highly dependent microservices based on load-based metrics, identified using centrality measures.
* Prediction should be based on a time series prediction model.
* The auto scaler should be configured to scale based on the forecasted predictions.

**4.6.2 Non-Functional Requirements**

The following are the non-functional requirements that are primarily being focused on during this research.

* Availability – The system should be able to function on a scheduled basis (e.g.: once per hour) throughout the day
* Efficiency – The system should be as efficient as possible and make use of minimal resources in the prediction process such that it does not affect the performance of the overall cluster.
* Performance – The system should be able to handle the vast amount of data it receives and be able to process the data without affecting the system performance.
* Interoperability – The system should be able to interact and communicate with the other components developed in this research and receive inputs as well as forward outputs to the desired components.

**4.7 Commercialization**

This developed load prediction and centrality analysis component is primarily designed as a supplementary component as part of the developed governance model. However, this component could also be developed as a standalone APM tool since it could be used to provide users a variety of benefits not seen in most autoscaling tools developed today. These benefits include the ability to:

* Integrate with monitoring solutions in order to retrieve load-based metrics and predict future loads based on the received metrics.
* Evaluate and provide solutions to visualize a vast variety of centrality measures based on loads.
* Automatically configure and auto-scale Kubernetes autoscaling tools based on inputs derived from the predicted load and calculated centrality measures.

This tool will be primarily targeted for system administrators and cloud engineers who manage and maintain cloud-based microservice deployments. Due to the heavy competition in the APM market space which primarily includes a vast variety of opensource APM tools, the initial marketing strategy will be to adopt an opensource marketing strategy to easily enter the marketspace and capture users. However, since this autoscaling tool is quite a unique tool that offers some useful benefits as mentioned above, adopting a freemium marketing strategy for this tool for users for additional supplementary features will the most appropriate approach once this tool becomes more popular in the future.

1. **TESTING AND IMPLEMENTATION RESULTS AND DISCUSSION**

**5.1 Testing**

**5.1.1 Selection of Optimal Prediction Model**

A key stage crucial to the development of the load prediction and centrality analysis component was the selection of the optimal prediction model the forecasting of load-based inter- microservice link dependency measures and resource utilization metrics. Hence, prior to the development of the finalized prediction model, initial testing was conducted on a multitude of time series prediction models which consisted of statistical and data-centric approaches to others which included machine learning-centric approaches. In this regard, the following time series prediction models were evaluated on a sample dataset containing resource utilization and performance metrics of VMs obtained from a distributed datacenter [23].

* ARIMA
* XGBoost
* Holt-Winters
* LSTM
* Tpot Auto ML

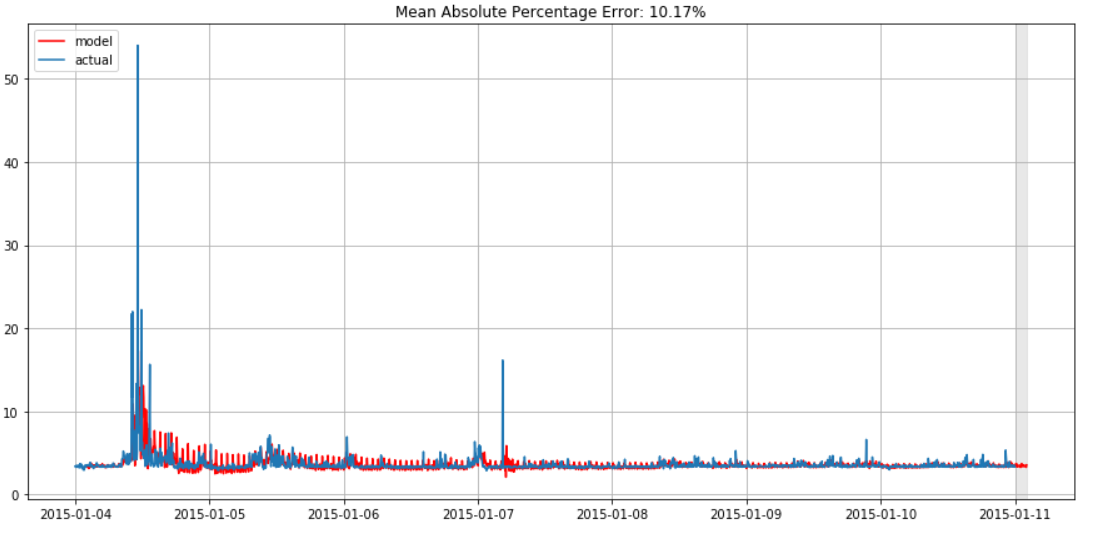
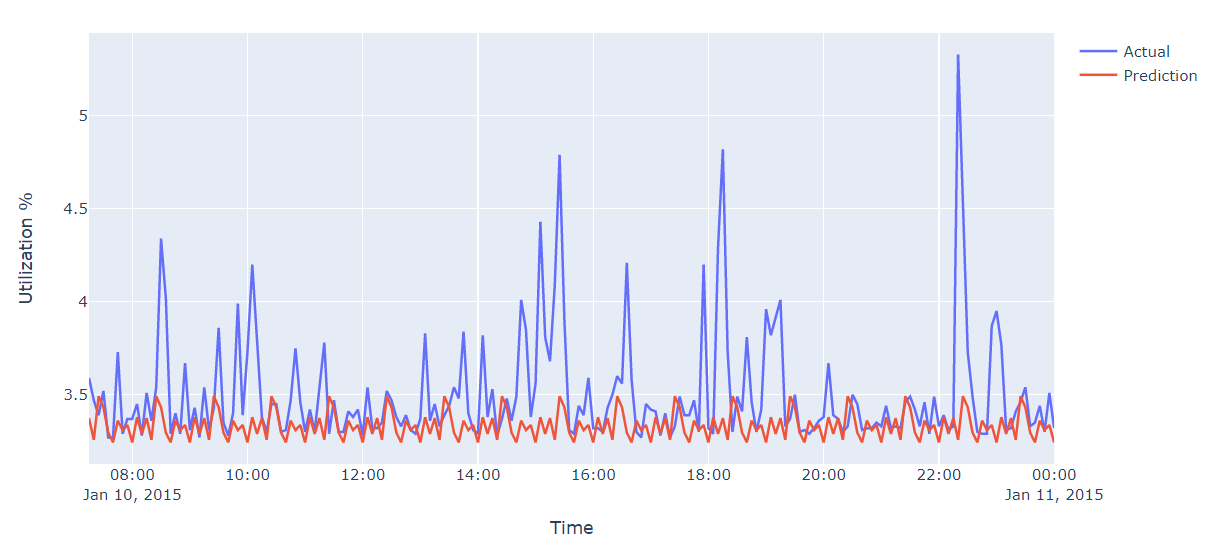
Moreover, as per the functional requirements of the prediction model required by the load prediction component, the above-listed prediction models were evaluated based on the following criteria.

* Ability to conduct an efficient prediction process with minimal data augmentation and pre-processing
* The prediction model should possess minimal model training time
* Ability to obtain a high prediction accuracy
* Ability to easily optimize the prediction model
* Easy parameter selection with a minimal number of parameters
* The prediction model should possess the ability to capture the history of data accurately
* Prediction model run time should be minimal

During the initial testing and model selection process developed through the utilization of Jupyter notebook scripts, the sample dataset was altered to create a sub-dataset in which each the time interval between records in the dataset was set to 5 minutes. In this regard, while focusing on the above-listed evaluation criteria, the “CPU Utilization Percentage” attribute of the dataset was evaluated for prediction. Note, for the evaluated machine learning models, XGBoost, LSTM, and Auto ML, information captured from 1 previous time step was utilized in the prediction of the subsequent record, whereas for statistical approaches which include the ARIMA and Holt-Winters models, a seasonal period 12 was used. Table 5.1 below summarizes the MAPE 2 (Mean Absolute Percentage Error) and RMSE results thus obtained through the evaluated prediction models and Fig. 5.1depicts the train vs test plots obtained through the application of the evaluated prediction models.

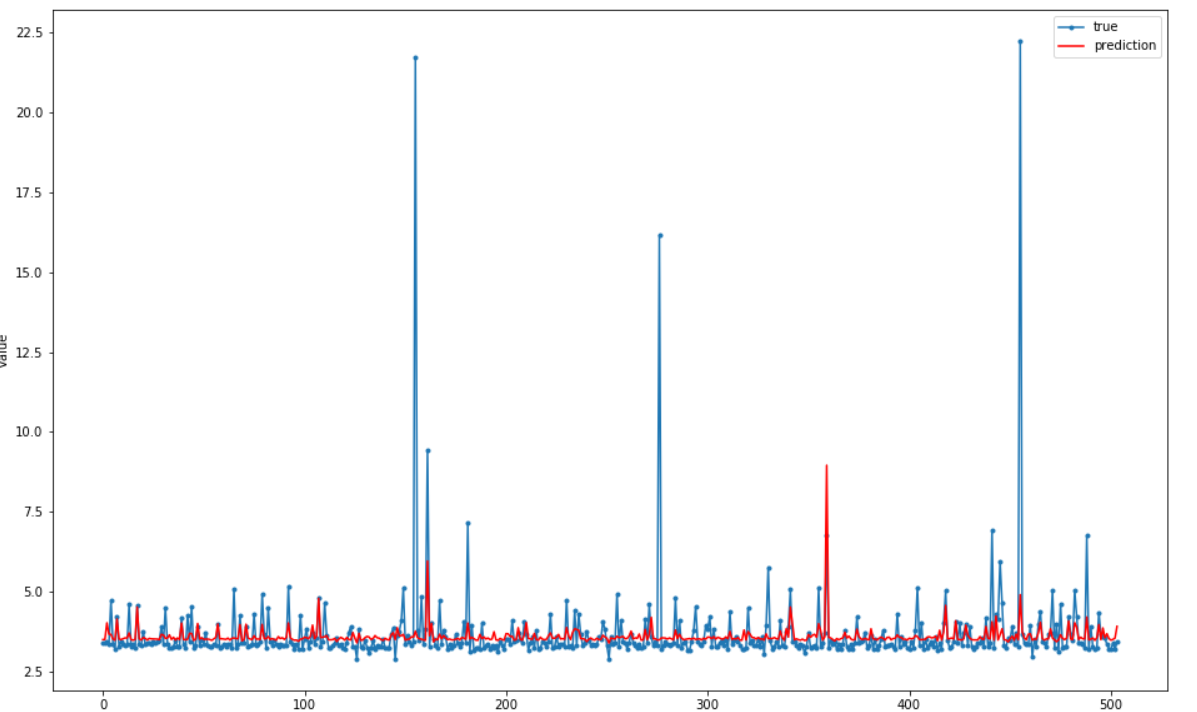
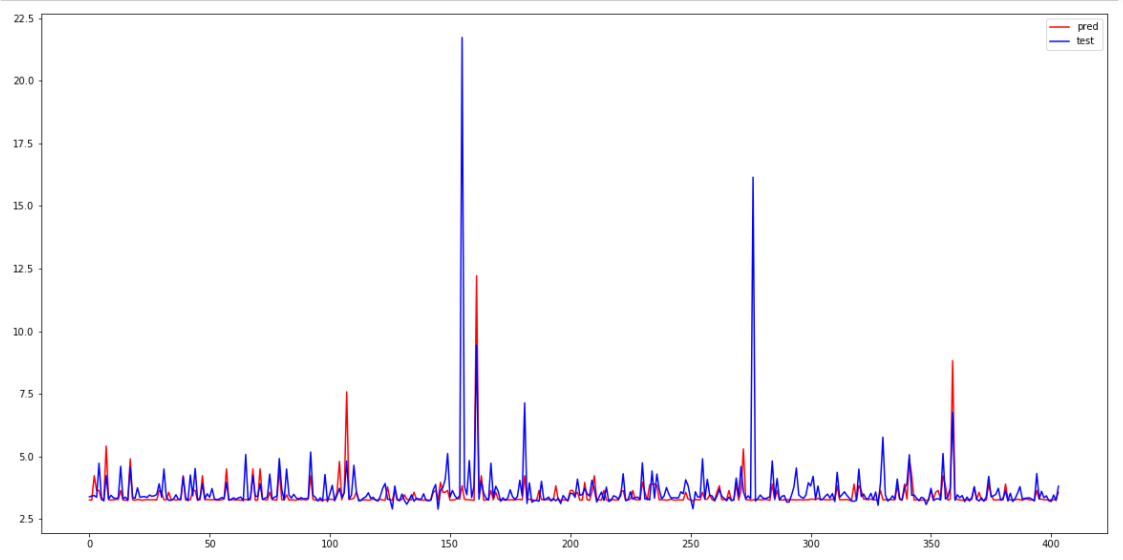
Table 5.1: MAPE 2 and RMSE values of evaluated prediction models

|  |  |
| --- | --- |
| **Model** | **MAPE/RMSE** |
| ARIMA | MAPE – 10.17%  RMSE – 1.414 |
| Holt-Winters | MAPE - 5.76%  RMSE - 0.346 |
| Tpot Auto ML | MAPE - 8.13%  RMSE - 1.334 |
| XGBoost | MAPE - 6.88 %  RMSE - 1.194 |
| LSTM | MAPE - 5.10%  RMSE - 0.304 |

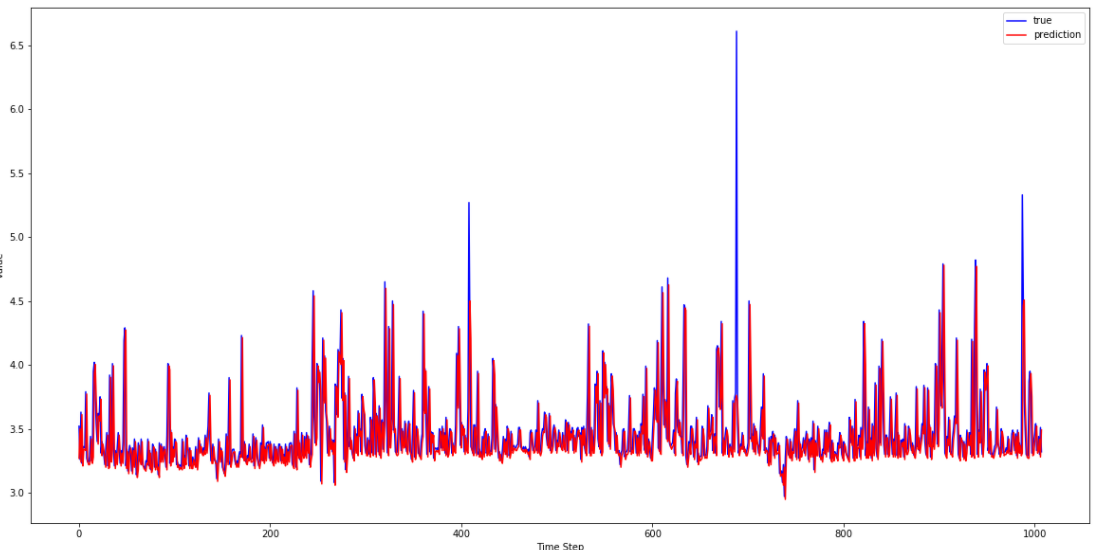
Holt Winters

ARIMA

Tpot Auto ML

XGBoost



LSTM

Figure 5.1: Evaluated prediction models

The initial testing process revealed several key findings. In this regard, a notable aspect observed throughout the testing process is the fact that amongst the prediction models evaluated, the LSTM prediction model was superior, possessing the least MAPE 2 and RMSE values. Moreover, other essential factors such as the minimal model runtime and parameter selection process, efficient model optimization process as well as the fact that the prediction process requires minimal data pre-processing, further proved the superiority of the LSTM prediction model in contrast to the remainder of the evaluated prediction models. Hence, based on these factors, the LSTM model proved to be the optimal prediction model to be utilized for the prediction of load-based inter- microservice link dependency measures and resource utilization metrics

**5.1.2 Testing of Developed Solution**

**5.1.2.1 Testing Process for Prediction of Load based Inter- microservice Link Dependency Measures and Resource Utilization Metrics**

On completion of the prediction model, a thorough testing process was conducted to evaluate the performance and accuracy of the developed prediction model. Moreover, through the utilization of a sample of 6 microservices\* deployed in a Kubernetes cluster hosted via AKS, along with the utilization of a load generation script for the generation of sample traffic among the microservices deployed, two sample datasets which contain recorded data pertaining to a given time period of the sample Kubernetes cluster, were obtained, These datasets were then utilized for evaluation of the developed prediction model. In this regard, the two datasets thus obtained are listed below.

* A dataset containing CPU resource utilization metrics of all pods in the Kubernetes cluster.
* A dataset containing load-based inter-microservice link dependency measures among microservices in the Kubernetes deployment.

**Note \***: - 6 microservices not including, pods allocated for metrics server. Also note, this research assumes one microservice is deployed on a single pod in the Kubernetes cluster as per industry standards and practices.

Figures 5.2 and 5.3 depict the structure of the resulting CSV format of the above-listed datasets utilized for prediction. Moreover, similar to the sample datasets utilized during the initial model selection process, the generated datasets listed above also contain data recorded within 5-minute time intervals. In this regard, the CPU resource utilization dataset contains 35 records each recorded within 5-minute intervals and the load-based inter-microservice link dependency dataset contains 144 records recorded within a 5-minute interval pertaining to load (requests per second) on each link in the microservice network deployed in the Kubernetes cluster.

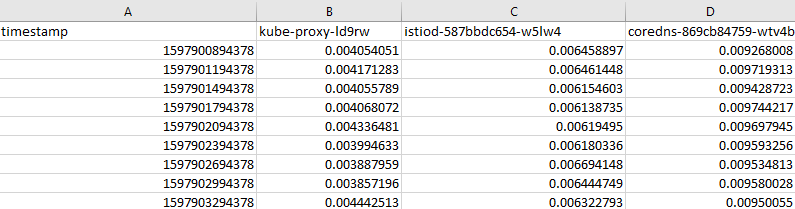


Figure 5.2: Structure of dataset containing CPU resource utilization metrics

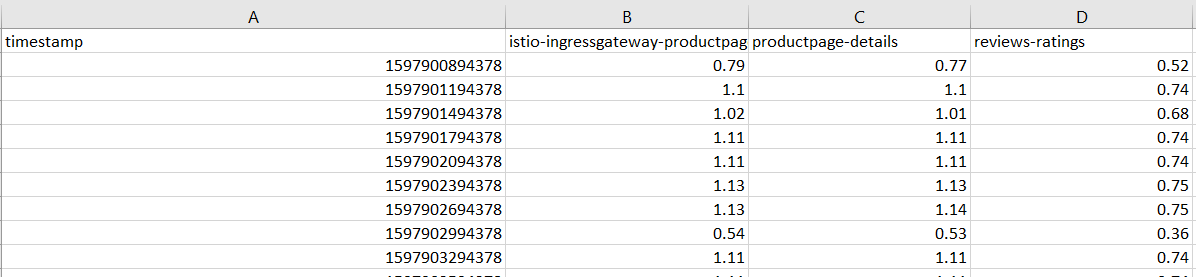


Figure 5.3: Structure of dataset containing load-based inter-microservice link dependency measures

**5.1.2.2 Testing Process for Centrality Evaluation**

The testing process observed during the centrality evaluation process was quite a straightforward process since the majority of the required centrality evaluation functions were available through the NetworkX python library. Similar to the testing process undertaken to the load prediction process, the centrality evaluation of microservices was also performed on the sample Kubernetes cluster deployed on AWS. In this regard, the testing process was performed to ensure centrality calculation could be performed on the existing microservice co-dependency network gathered from the CSV containing current load-based inter-microservice link dependency measures among microservices in the Kubernetes deployment as well as the predicted load-based inter-microservice link dependency measure, derived through the prediction model.

**5.2 Test Results**

In accordance with the testing process as described in the section above, the load-based metrics (resource utilization measures and inter-microservice link dependency measures) are predicted utilizing the developed LTMS prediction model. The following sections provide an overview of the prediction results thus obtained as well as an overview of key performance evaluation metrics of the developed prediction model.

**5.2.1 Prediction of Resource Utilization Metrics**

Through the evaluation of the developed prediction model on the sample CPU resource utilization metric dataset, the MAPE 2, RMSE, SMAPE, and MASE performance measures were evaluated. In this case, a past history of 1 hour (t-12; where “t” is the last recorded time of the time-series dataset and a time “t-1” represents the time 5 minutes prior to the last recorded time, “t”) was utilized for the prediction of CPU resource utilization values 1 hour in advance (t+12).

In this regard, Table 5.2 depicts the relevant parameter values set for the prediction model during the prediction process whereas Fig. 5.4 and Fig. 5.5 depict the results of key model performance evaluation metrics along with the forecasted results respectively.

Table 5.2: Model Parameters – Resource utilization prediction

|  |  |  |
| --- | --- | --- |
| Parameter | Parameter Description | Value |
| RES\_PAST\_HISTORY | Lookback period | 12 |
| RES\_NUM\_FEATURES | Number of features considered | 1 |
| RES\_NUM\_EPOCHS | Number of epochs | 100 |
| RES\_SPLIT\_SIZE | Split size for train and test subsets | 0.5 |
| RES\_TRAIN\_BATCH\_SIZE | Train subset batch size | 12 |
| RES\_TEST\_BATCH\_SIZE | Test subset batch size | 1 |
| RES\_NUM\_PRED | Number of forecast steps | 12 |

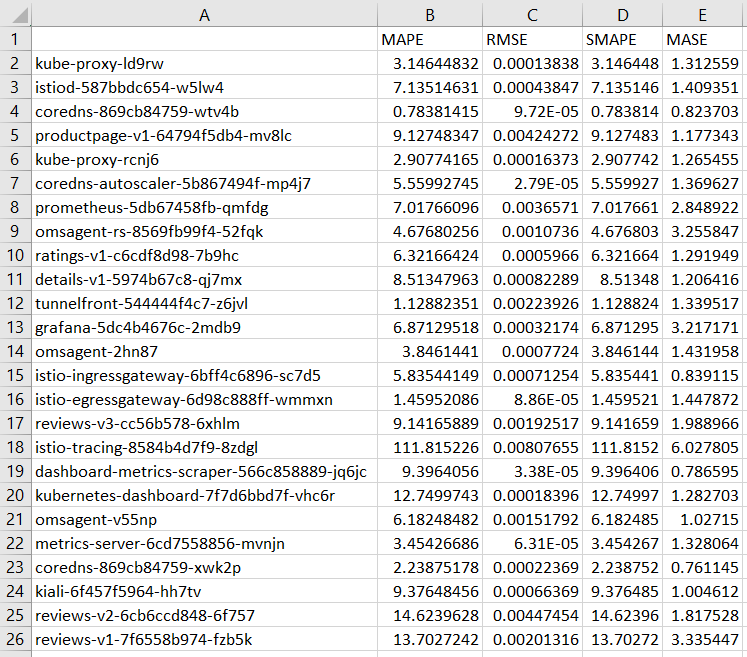


Figure 5.4: MAPE 2, RMSE, SMAPE and MASE values - resource utilization prediction (JSON result converted to CSV for improved readability)

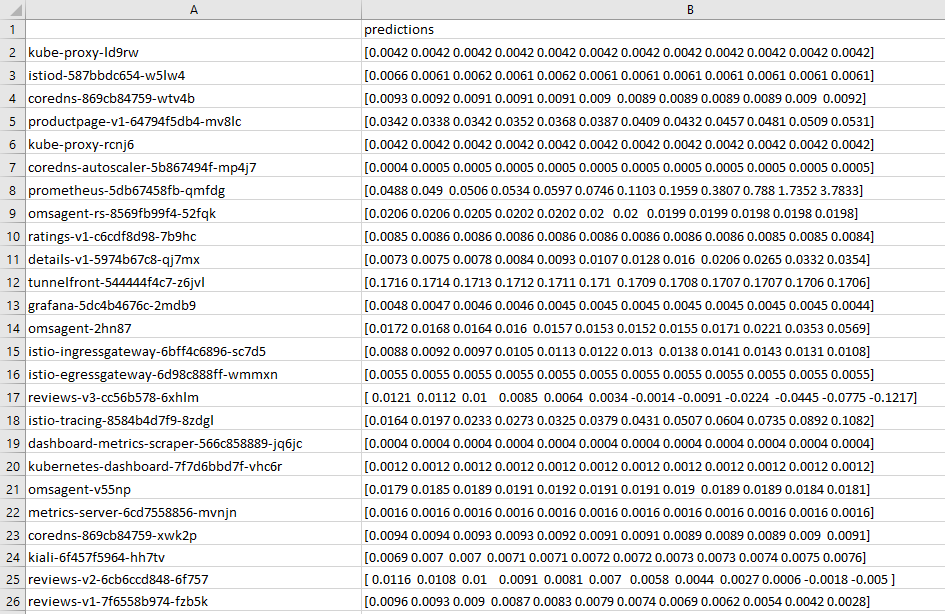


Figure 5.5: 1-hour forecast (t+12) of CPU utilization metrics (JSON result converted to CSV for improved readability)

**Note \***: - Above predictions (Fig. 5.5) depict predictions at 5-minute intervals for the next 1 hour.

**5.2.2 Prediction of Load - based Inter- microservice Link Dependency Measures**

Through evaluation of the developed prediction model on the dataset containing load-based inter-microservice link dependency measures, MAPE 2, RMSE, SMAPE, and MASE performance measures were evaluated. In this case, a past history of 1 hour (12 \* 5 min) was utilized for the prediction of microservice link dependency measures 1 hour in advance, with a time interval of 5 minutes between each successive prediction. Table 5.3 depicts the performance evaluation scores of the developed prediction model in the prediction of the results along with the relevant parameter values set for the prediction model.

Table 5.3: Model Parameters – Inter-microservice link dependency measures

|  |  |  |
| --- | --- | --- |
| Parameter | Parameter Description | Value |
| DEP\_PAST\_HISTORY | Lookback period | 12 |
| DEP\_NUM\_FEATURES | Number of features considered | 1 |
| DEP\_NUM\_EPOCHS | Number of epochs | 100 |
| DEP\_SPLIT\_SIZE | Split size for train and test subsets | 0.7 |
| DEP\_TRAIN\_BATCH\_SIZE | Train subset batch size | 1 |
| DEP\_TEST\_BATCH\_SIZE | Test subset batch size | 1 |
| DEP\_NUM\_PRED | Number of forecast steps | 12 |

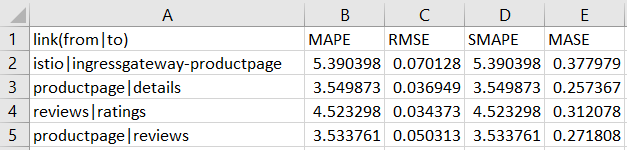


Figure 5.6: MAPE 2, RMSE, SMAPE and MASE values - inter-microservice link dependency measures (JSON result converted to CSV for improved readability)

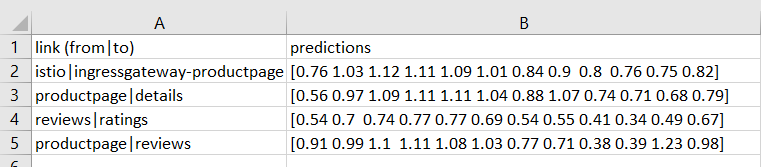
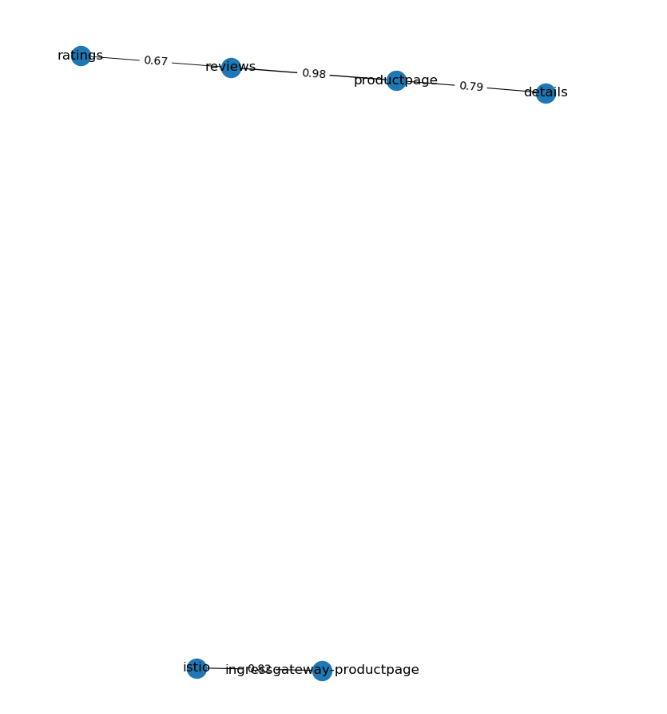


Figure 5.7: 1-hour forecast (t+12) of load-based inter-microservice link dependency measures (JSON result converted to CSV for improved readability)

**Note \***: - Above predictions (Fig. 5.7) depict predictions at 5-minute intervals for the next 1 hour.



0.82

Predicted link dependency measures at time t+12

Figure 5.8.1: Predicted microservice co-dependency network at time t+12 (1 hour in advance)

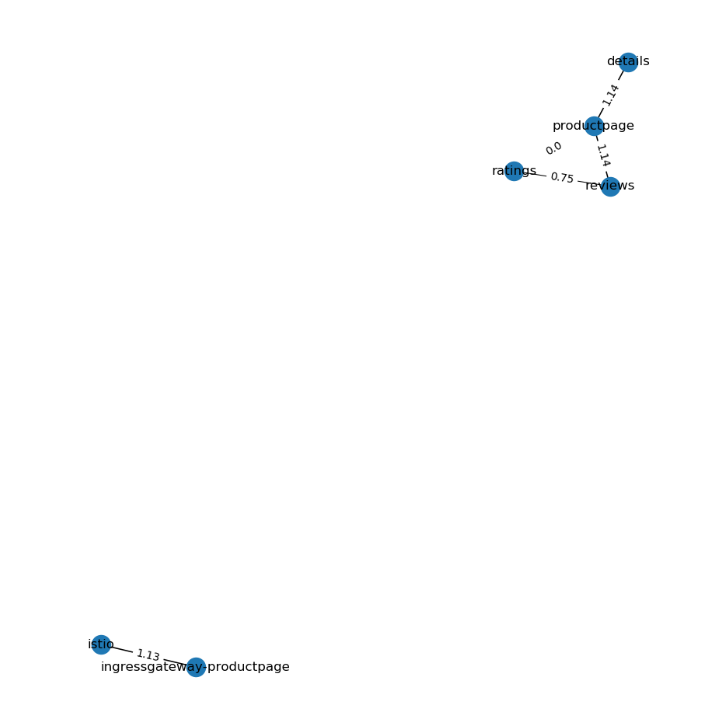
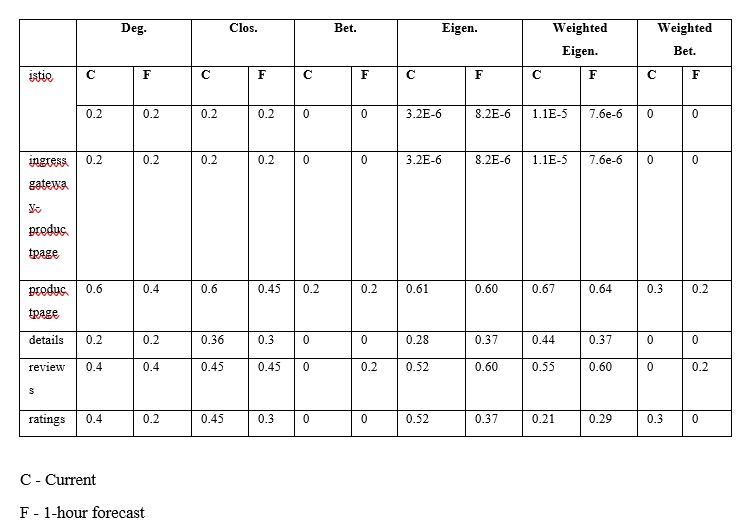


Figure 5.8.2: Microservice co-dependency network at time t

**Note \***: - In the above diagram, a dependency measure of 0 between a two microservices indicates no network traffic was recorded passing through the given link throughout the observed period, until time t.

**5.2.3 Centrality Evaluation**

Centrality evaluation measures were performed on the existing (Fig. 5.8.1) and predicted representation (Fig. 5.8.2) of the microservice co-dependency network. Table 5.9 below depicts a comparison of key evaluated centrality measures (rounded off to two decimal places) pertaining to the current and forecasted representation of the microservice co-dependency network.



**C** – Centrality measure for **Deg.** – Degree Centrality

co – dependency network at time t **Clos.** – Closeness Centrality

**F** – Centrality measure for predicted **Bet.** – Betweenness Centrality

co - dependency network at time t+12 **Eigen.** – Eigenvector Centrality

**Mic.** – Microservice

**Mic.**

Figure 5.9: Figure of the table depicting the comparison of current and forecasted centrality measures of co-dependency network

As per the figure provided above, it is evident there are variations in the centrality measures of microservices in the current co-dependency network at time t and predicted co-dependency network at time t +12. In this regard, prominent highlights to the changes in centrality measures between the two co-dependency measures include, the change in degree centrality measures in the “productpage” and “ratings” microservice, the change in closeness centrality measures in the “productpage”, “detail” and “ratings” and microservices, the change in the betweenness centrality in the “reviews” microservice as well as other changes in the centrality measures of microservices with respect to the remaining eigenvector, weighted eigenvector, and weighted betweenness centrality.

**5.2.3 Determination of Optimal Microservice Instance Levels to the Creation of Improved Autoscaling Policies through Integration with the Optimization Algorithm**

Testing was also conducted to evaluate the application of predicted link dependency measures and to facilitate the determination of optimal microservice instance levels required for autoscaling. For evaluation purposes, the JSON (JavaScript Object Notation) representation of a sample cluster dataset containing the predicted microservice link dependency measures along with the additional input information required, is provided to the developed optimization algorithm in order to compute the optimized solutions.

In this regard, the determination optimal microservice instances, required for the creation of autoscaling policies were evaluated pertaining to a given set of optimization objectives as listed below.

* Deployment with best cluster performance
* Deployment with highest cluster availability
* Most cost-effective deployment

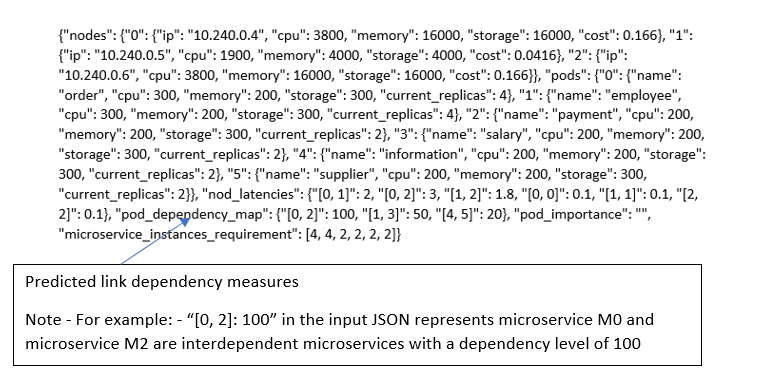


Figure 5.10: Structure of sample JSON input



**S** - Existing / current instance number as per the sample JSON input

R - Optimal number of instances as determined by the optimization algorithm

Figure 5.11: Figure of the table depicting the comparison of current and optimal microservice instance numbers

**5.3 Research Findings and Discussion**

As per the results highlighted above, it is evident there is a possibility of improving the prediction model. Moreover, through evaluation of MAPE 2, RMSE, MASE, and SMAPE scores of the developed prediction model, it is evident, at times, the prediction model fails to effectively predict and forecast certain microservice link dependency measures and resource utilization metrics of certain pods in the microservice deployment. Regardless, it is also crucial to note the fact that a majority of predictions performed through the utilization of the developed prediction model are within an acceptable prediction accuracy.

In light of the findings obtained, future work would include improvements to the current prediction model utilized in the prediction process. In this regard, possible future work would include experimentation with varying model orientations and architectures with varied layers. Moreover, in future work, it is also expected to perform additional testing utilizing more advanced prediction models such as ES-RNN and TCN.

With regard, to the application of measures derived from the load prediction and centrality analysis component to the development of holistic autoscaling and deployment policies through integration with the optimization algorithm. It is evident additional testing should be performed on microservice clusters containing a larger microservice count as larger microservice clusters may highlight hidden complexities previously unknown.

Moreover, as per the testing process carried out in conjunction of the optimization algorithm, it was evident the current iteration of the optimization algorithm developed in this research, makes use of limited centrality measures in the determination of optimized deployment policies, hence in future work, the optimization algorithm will be modified to incorporate the use of additional centrality measures derived via the load prediction and centrality analysis component.

**6.0 CONCLUSION**

The current implementation of MAPE 1 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 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. In this regard, this document provided a detailed analysis regarding the key concepts, ideologies, and methodologies adopted in the development of the finalized solution, as well as a detailed analysis of key findings obtained.

Analysis of results reveals the potential for further optimization of key aspects in the developed solution pertaining to the efficiency and accuracy of the prediction process. Moreover, future work pertaining to the research would also include steps to enhance the inclusion of a wider array of centrality evaluation measures to further facilitate the creation of optimal deployment policies.

**References**

[1] “What is Kubernetes,” Kubernetes. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>. [Accessed: 13-Feb-2020].

[2] AWS Auto Scaling", *Amazon Web Services, Inc.*, 2020. [Online]. Available: <https://aws.amazon.com/autoscaling/>. [Accessed: 20- Feb- 2020].

[3] C. Qu, R. N. Calheiros, and R. Buyya, “Auto-Scaling Web Applications in Clouds,” *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–33, 2018.

[4] H. Alipour, A. Hamou-Lhajd and X. Liu, “Analyzing Auto-scaling Issues in Cloud Environments”, 2014

[5] P. Singh, P. Gupta, K. Jyoti, and A. Nayyar, “Research on Auto-Scaling of Web Applications in Cloud: Survey, Trends and Future Directions,” *Scalable Computing: Practice and Experience*, vol. 20, no. 2, pp. 399–432, Feb. 2019.

[6] A. Evangelidis, D. Parker, and R. Bahsoon, “Performance Modelling and Verification of Cloud-Based Auto-Scaling Policies,” *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[7] S. Taherizadeh and M. Grobelnik, “Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications,” *Advances in Engineering Software*, vol. 140, p. 102734, 2020.

[8] Zhao, A., Huang, Q., Huang, Y., Zou, L., Chen, Z., & Song, J. “Research on Resource Prediction Model Based on Kubernetes Container Auto-scaling Technology.” IOP Conference Series: Materials Science and Engineering, 2019.

[9] Yamin Wang, L. Wu and Shouxiang Wang, "Challenges in applying the empirical mode decomposition based hybrid algorithm for forecasting renewable wind/solar in practical cases," 2016 IEEE Power and Energy Society General Meeting (PESGM), Boston, MA, 2016, pp. 1-5

[10] H. Zhao, H. Lim, M. Hanif and C. Lee, "Predictive Container Auto-Scaling for Cloud-Native Applications," 2019 International Conference on Information and Communication Technology Convergence (ICTC), Jeju Island, Korea (South), 2019, pp. 1280-1282.

[11] Y. Meng, R. Rao, X. Zhang and P. Hong, "CRUPA: A container resource utilization prediction algorithm for auto-scaling based on time series analysis," *2016 International Conference on Progress in Informatics and Computing (PIC)*, Shanghai, 2016, pp. 468-472

[12] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, “Machine learning-based auto-scaling for containerized applications,” *Neural Computing and Applications*, Aug. 2019.

[13] W.-Y. Kim, J.-S. Lee, and E.-N. Huh, “Study on proactive auto scaling for instance through the prediction of network traffic on the container environment,” *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication - IMCOM 17*, 2017.

[14] T. Ye, X. Guangtao, Q. Shiyou, and L. Minglu, “An Auto-Scaling Framework for Containerized Elastic Applications,” *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, 2017.

[15] S. Taherizadeh and V. Stankovski, “Dynamic Multi-level Auto-scaling Rules for Containerized Applications,” *The Computer Journal*, vol. 62, no. 2, pp. 174–197, Aug. 2018.

[16] J. Sahni and D. P. Vidyarthi, “Heterogeneity-aware adaptive auto-scaling heuristic for improved QoS and resource usage in cloud environments,” *Computing*, vol. 99, no. 4, pp. 351–381, 2016.

[17] Z. A. Al-Sharif, Y. Jararweh, A. Al-Dahoud, and L. M. Alawneh, “ACCRS: autonomic based cloud computing resource scaling,” *Cluster Computing*, vol. 20, no. 3, pp. 2479–2488, 2016.

[18] Kukade P.P and Kale G, “Auto-scaling of micro-services using containerization”, Int J SciRes(IJSR) 2015;4(9):1960–3.

[19] C. Kan, "DoCloud: An elastic cloud platform for Web applications based on Docker," 2016 18th International Conference on Advanced Communication Technology (ICACT), Pyeongchang, 2016, pp. 478-483.

[20] L.R. Moore, K. Bean and T. Ellahi, “A Coordinated Reactive and Predictive Approach to Cloud Elasticity.” *CLOUD 2013*, 2013

[21] W.-Y. Kim, J.-S. Lee, and E.-N. Huh, “Study on proactive auto scaling for instance through the prediction of network traffic on the container environment,” *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication - IMCOM 17*, 2017.

[22] "Horizontal Pod Autoscaler", Kubernetes, 2020. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. [Accessed: 10- Sep- 2020].

[23] “GWA-T-13 Materna.” http://gwa.ewi.tudelft.nl/datasets/gwa-t-13-materna [Accessed Sep. 10, 2020].

.

**Appendix**

**Appendix A: Coding Solution for Prediction of Load-based Inter - microservice Dependency Measures / Pod Resource Utilization Metrics**

import networkx as nx  
from centrality.run\_centrality import prediction\_to\_centrality, get\_centrality\_measures  
from load\_prediction.univariate\_LSTM\_model import univariate\_LSTM\_model  
from load\_prediction.util\_functions import \*  
from load\_prediction.evaluation\_functions import \*  
from statsmodels.tools.eval\_measures import rmse  
  
# initialize array lists to store prediction and error scores (MAPE,RMSE,SMAPE,MASE)  
# for link prediction and resource utilization prediction  
link\_prediction\_list = []  
link\_mape\_prediction\_error = []  
link\_rmse\_prediction\_error = []  
link\_smape\_prediction\_error = []  
link\_mase\_prediction\_error = []  
  
resource\_prediction\_list = []  
resource\_mape\_prediction\_error = []  
resource\_rmse\_prediction\_error = []  
resource\_smape\_prediction\_error = []  
resource\_mase\_prediction\_error = []  
  
  
# PARAMETERS  
#################  
# dataset\_path - path to retrieved CSV dataset  
# split\_size - train and test subset split ratio  
# past\_history - number of previous time steps considered for performing predictions  
# train\_batch\_size - train batch size of TimeSeriesGenerator  
# test\_batch\_size - test batch size of TimeSeriesGenerator  
# num\_epochs - maximum number of epochs to be iterated  
# num\_pred - number of forecast steps required  
# num\_features - number of features of dataset  
# pred\_type - defines whether prediction is a link dependency prediction or resource utilization prediction  
  
def get\_json\_prediction(dataset\_path, split\_size, past\_history, train\_batch\_size, test\_batch\_size, num\_epochs, num\_pred,  
 num\_features,  
 pred\_type):  
 # formatting a pre-processing of dataset  
 formatted\_dataset = read\_and\_pre\_process\_dependency\_pred\_csv(dataset\_path)  
  
 # call run\_prediction function  
 # populates array lists with predicted link dependency / resource utilization values and error scores  
 run\_prediction(formatted\_dataset, split\_size, past\_history, train\_batch\_size, test\_batch\_size, num\_epochs, num\_pred,  
 num\_features,  
 pred\_type)  
  
 # check if prediction performed is a link dependency prediction  
 if pred\_type == "link":  
  
 # create a dictionary with error scores  
 error\_dict = {'MAPE': link\_mape\_prediction\_error, 'RMSE': link\_rmse\_prediction\_error,  
 "SMAPE": link\_smape\_prediction\_error, "MASE": link\_mase\_prediction\_error}  
  
 # creates a dataframe from populated link\_prediction\_list  
 prediction\_df = pd.DataFrame(link\_prediction\_list, index=formatted\_dataset.columns)  
  
 # converts error\_dict to dataframe  
 error\_df = pd.DataFrame(error\_dict, index=formatted\_dataset.columns)  
  
 # if prediction performed is a resource utilization prediction  
 else:  
  
 # create a dictionary with error scores  
 error\_dict = {'MAPE': resource\_mape\_prediction\_error, 'RMSE': resource\_rmse\_prediction\_error,  
 "SMAPE": resource\_smape\_prediction\_error, "MASE": resource\_mase\_prediction\_error}  
  
 # creates a dataframe from populated resource\_prediction\_list  
 prediction\_df = pd.DataFrame(resource\_prediction\_list, index=formatted\_dataset.columns)  
  
 # converts error\_dict to dataframe  
 error\_df = pd.DataFrame(error\_dict, index=formatted\_dataset.columns)  
  
 print("######################")  
 print(prediction\_df)  
  
 if pred\_type == "link":  
 # creation of predicted co-dependency map if prediction performed is a link dependency measure prediction  
 # returns networkx graph of predicted co - dependency map  
 G = prediction\_to\_centrality(prediction\_df)  
  
 print(error\_df)  
 print("######################")  
  
 # converts the prediction\_df to a JSON format  
 out = format\_dependency\_df(prediction\_df)  
 json\_out = "[" + out + "]"  
 # print(json\_out)  
  
 # if prediction performed is a link prediction, return the converted JSON output  
 # and networkx graph G  
 if pred\_type == "link":  
  
 return json\_out, G  
  
 # if prediction performed is a resource utilization prediction , return JSON output  
 else:  
  
 return json\_out  
  
  
def run\_prediction(dataset, split\_size, time\_steps, train\_batch\_size, test\_batch\_size, epochs, prediction\_steps,  
 feature\_num, type,  
 is\_scaled=True):  
 # iterate through each column in the dataset  
 for col in dataset.columns:  
 print("current link / pod ", col)  
  
 # create a dataset copy  
 dataset\_copy = dataset.copy()  
  
 # set the column to be predicted  
 nodelink = col  
 feature\_needed = [nodelink]  
  
 # split into train and test subsets  
 train, test = split\_train\_test(split\_size, dataset)  
  
 # reshape train values and test values  
 y\_train = train[nodelink].values.reshape((-1, 1))  
 y\_test = test[nodelink].values.reshape((-1, 1))  
  
 # if scaling is required (default = True)  
 if is\_scaled:  
  
 # perform scaling of train and test values  
 sc = MinMaxScaler(feature\_range=(-1, 1))  
 y\_train\_set\_scaled = sc.fit\_transform(y\_train)  
 y\_test\_set\_scaled = sc.fit\_transform(y\_test)  
  
 # create train\_generator and test\_generator for time series prediction using scaled values  
 train\_generator, test\_generator = build\_timeseries\_generator\_sets(y\_train\_set\_scaled, y\_test\_set\_scaled,  
 time\_steps=time\_steps,  
 train\_batch\_size=train\_batch\_size,  
 test\_batch\_size=test\_batch\_size)  
  
 # if scaling is not required  
 else:  
  
 # create train\_generator and test\_generator for time series prediction using unscaled values  
 train\_generator, test\_generator = build\_timeseries\_generator\_sets(y\_train, y\_test, time\_steps=time\_steps,  
 train\_batch\_size=train\_batch\_size,  
 test\_batch\_size=test\_batch\_size)  
  
 # defining LSTM model  
 lstm\_model = univariate\_LSTM\_model(time\_steps, feature\_num, epochs, train\_generator)  
  
 # pass to build\_model\_and\_fit function to build and fit model  
 fitted\_model, model\_history = lstm\_model.build\_model\_and\_fit()  
  
 # retrieve prediction for test data  
 predictions = fitted\_model.predict\_generator(test\_generator)  
  
 # if scaling is performed, rescale back to original values  
 if is\_scaled:  
  
 # get original values from scaled predictions  
 predicted\_rescaled = sc.inverse\_transform(predictions)  
  
 # get test y values from test generator  
 test\_eval = get\_y\_from\_generator(test\_generator)  
  
 # get original values from scaled test values  
 test\_eval\_rescaled = sc.inverse\_transform(test\_eval)  
  
 # get train y values from test generator  
 train\_eval = get\_y\_from\_generator(train\_generator)  
  
 # get original values from scaled train values  
 train\_eval\_rescaled = sc.inverse\_transform(train\_eval)  
  
 # visualize train and test by plotting graph  
 plot\_graph(test\_eval\_rescaled, predicted\_rescaled)  
  
 # compute error score MAPE,RMSE,SMAPE and MASE comparing expected and predicted values  
 mape\_error = calculate\_mape(test\_eval\_rescaled, predicted\_rescaled)  
 rmse\_error = rmse(test\_eval\_rescaled, predicted\_rescaled)  
 smape\_error = calculate\_smape(test\_eval\_rescaled, predicted\_rescaled)  
 mase\_error = calculate\_mase(train\_eval\_rescaled, test\_eval\_rescaled, predicted\_rescaled)  
  
 # if scaling was not performed  
 else:  
  
 # get test y values from test generator  
 test\_eval = get\_y\_from\_generator(test\_generator)  
  
 # visualize train and test by plotting graph  
 plot\_graph(test\_eval, predictions)  
  
 # compute error score MAPE,RMSE,SMAPE and MASE comparing expected and predicted values  
 mape\_error = calculate\_mape(test\_eval, predictions)  
 rmse\_error = rmse(test\_eval, predictions)  
 smape\_error = calculate\_smape(test\_eval, predictions)  
 mase\_error = calculate\_mase(train\_eval, test\_eval, predictions)  
  
 # Forcasting process starts here  
  
 # create the required history of records from dataset copy to perform the required forecasting  
 df = dataset\_copy[-time\_steps:]  
  
 # select the column forecasting is needed to be performed  
 val = df[feature\_needed]  
  
 # if scaling was performed  
 if is\_scaled:  
  
 # scaling  
 scaled\_data\_to\_predict = sc.fit\_transform(val.values.reshape((-1, 1)))  
  
 # perform forecasting process for future time steps  
 # returns forecasted values  
 list\_val = forecast\_future(scaled\_data\_to\_predict, fitted\_model, prediction\_steps, time\_steps)  
  
 # reshape forecasted values for inverse scaling (convert to original values)  
 vals = list\_val.reshape((-1, 1))  
  
 # perform inverse scaling  
 inverse\_preds = sc.inverse\_transform(vals)  
 inverse\_preds = inverse\_preds.reshape((-1))  
  
 # if forecast was performed was for a link  
 if type == 'link':  
  
 # rounding off inverse\_preds to two decimal places  
 inverse\_preds = np.round(inverse\_preds, 2)  
  
 # populating link\_prediction\_list array list with forecasted values  
 link\_prediction\_list.append({"predictions": inverse\_preds[1:]})  
  
 # populating error scores array list initialized above with relevant error scores  
 link\_mape\_prediction\_error.append(mape\_error)  
 link\_rmse\_prediction\_error.append(rmse\_error[0])  
 link\_smape\_prediction\_error.append(smape\_error)  
 link\_mase\_prediction\_error.append(mase\_error)  
  
 # if forecast was performed was resource utilization  
 else:  
  
 # rounding off inverse\_preds to four decimal places  
 inverse\_preds = np.round(inverse\_preds, 4)  
  
 # populating resource\_prediction\_list array list with forecasted values  
 resource\_prediction\_list.append({"predictions": inverse\_preds[1:]})  
  
 # populating error scores array list initialized above with relevant error scores  
 resource\_mape\_prediction\_error.append(mape\_error)  
 resource\_rmse\_prediction\_error.append(rmse\_error[0])  
 resource\_smape\_prediction\_error.append(smape\_error)  
 resource\_mase\_prediction\_error.append(mase\_error)  
  
 # if scaling was not performed  
 else:  
  
 # perform forecasting process for future time steps  
 # returns forecasted values  
 list\_val = forecast\_future(val.values, fitted\_model, prediction\_steps, time\_steps)  
  
 # if forecast was performed was for a link  
 if type == 'link':  
  
 # populating link\_prediction\_list array list with forecasted values  
 link\_prediction\_list.append({"predictions": list\_val[1:]})  
  
 # populating error scores array list initialized above with relevant error scores  
 link\_mape\_prediction\_error.append(mape\_error)  
 link\_rmse\_prediction\_error.append(rmse\_error[0])  
 link\_smape\_prediction\_error.append(smape\_error)  
 link\_mase\_prediction\_error.append(mase\_error)  
  
 # if forecast was performed was resource utilization  
 else:  
  
 # populating resource\_prediction\_list array list with forecasted values  
 resource\_prediction\_list.append({"predictions": list\_val[1:]})  
  
 # populating error scores array list initialized above with relevant error scores  
 resource\_mape\_prediction\_error.append(mape\_error)  
 resource\_rmse\_prediction\_error.append(rmse\_error[0])  
 resource\_smape\_prediction\_error.append(smape\_error)  
 resource\_mase\_prediction\_error.append(mase\_error)

**Appendix B: Coding Solution for Creation of Current Co – dependency Map and Evaluation of Centrality Measures**

# function to calculate a centrality measures for current co - dependency map using CSV dataset  
def get\_centrality\_for\_type(centrality\_type, path):  
 dataset = convert\_dataset\_to\_centrality\_df(path)  
 G = draw\_graph\_from\_node\_dataset(dataset, 'From\_Node', 'To\_Node', edge\_attr='Current\_Weight')  
 return get\_centrality\_measures(centrality\_type, G)

# converted to networkx graph format towards the creation of the current co - dependency map  
def convert\_dataset\_to\_centrality\_df(file\_path):  
 column\_arr = []  
  
 dataset = pd.read\_csv(file\_path)  
 last\_row\_df = dataset.tail(1)  
 last\_row\_df = last\_row\_df.drop("timestamp", axis=1)  
 last\_row\_df\_columns = last\_row\_df.columns  
  
 for col in last\_row\_df\_columns:  
 column\_arr.append(col.split('|'))  
  
 from\_column = [item[0] for item in column\_arr]  
 to\_column = [item[1] for item in column\_arr]  
 weight\_column = last\_row\_df.values.reshape(-1)  
 d = {'From\_Node': from\_column, 'To\_Node': to\_column, 'Current\_Weight': weight\_column}  
 centrality\_df = pd.DataFrame(d)  
 return centrality\_df

# convert link dependency dataset to co - dependency map  
# creation of current co - dependency map  
def draw\_graph\_from\_node\_dataset(dataset, from\_node, to\_node, edge\_attr):  
 # convert dataset to networkx graph objecT  
 G = nx.from\_pandas\_edgelist(dataset, from\_node, to\_node, edge\_attr=edge\_attr)  
  
 # get edge weights  
 durations = [i[edge\_attr] for i in dict(G.edges).values()]  
  
 # get node labels  
 labels = [i for i in dict(G.nodes).keys()]  
 labels = {i: i for i in dict(G.nodes).keys()}  
  
 # plotting co - dependency map  
 fig, ax = plt.subplots(figsize=(10, 10))  
 pos = nx.spring\_layout(G, scale=0.1, k=0.1)  
 nx.draw\_networkx\_nodes(G, pos, ax=ax, labels=True)  
 weights = nx.get\_edge\_attributes(G, "Current\_Weight")  
 nx.draw\_networkx\_edges(G, pos, width=durations, ax=ax)  
 \_ = nx.draw\_networkx\_labels(G, pos, labels, ax=ax)  
 nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=weights)  
 plt.show()  
  
 return G

# function to calculate a centrality measures given required centrality type  
def get\_centrality\_measures(centrality\_type, G):  
 if centrality\_type == "degree":  
 return str(get\_degree\_centrality(G))  
 elif centrality\_type == "closeness":  
 return str(get\_closeness\_centrality(G))  
 elif centrality\_type == "betweenness":  
 return str(get\_betweenness\_centrality(G))  
 elif centrality\_type == "load":  
 return str(get\_load\_centrality(G))  
 elif centrality\_type == "eigenvector":  
 return str(get\_eigenvector\_centrality(G))  
 elif centrality\_type == "w-eigenvector":  
 return str(get\_weighted\_eigenvector\_centrality(G, weight='Current\_Weight'))  
 elif centrality\_type == "w-betweenness":  
 return str(get\_weighted\_betweenness\_centrality(G, weight='Current\_Weight'))

**Appendix C: Coding Solution for Creation of Predicted Co – dependency Map and Evaluation of Centrality Measures**

if pred\_type == "link":  
 # creation of predicted co-dependency map if prediction performed is a link dependency measure prediction  
 # returns networkx graph of predicted co - dependency map  
 G = prediction\_to\_centrality(prediction\_df)

# function to convert predicted link dependency measures to predicted co - dependency map  
# returns predicted co - dependency map networkx graph object  
def prediction\_to\_centrality(predicted\_df):  
  
 # converting predicted measures to predicted co - dependency map  
 df = prediction\_df\_to\_centrality\_dataframe(predicted\_df)  
 G = draw\_graph\_from\_node\_dataset(df, 'From\_Node', 'To\_Node', edge\_attr='Current\_Weight')  
 return G

# function to convert dataframe containing predicted dependency measures to a dataframe format that can be  
# converted to networkx graph format  
def prediction\_df\_to\_centrality\_dataframe(prediction\_df):  
 df = []  
 weights = []  
 for link in prediction\_df.index:  
 df.append(link.split('|'))  
  
 from\_cols = [item[0] for item in df]  
 to\_cols = [item[1] for item in df]  
  
 for index, row in prediction\_df.iterrows():  
 vals = row['predictions']  
 weights.append(vals[-1])  
  
 d = {'From\_Node': from\_cols, 'To\_Node': to\_cols, 'Current\_Weight': weights}  
 predicted\_df = pd.DataFrame(d)  
 return predicted\_df

# convert link dependency dataset to co - dependency map  
# creation of current co - dependency map  
def draw\_graph\_from\_node\_dataset(dataset, from\_node, to\_node, edge\_attr):  
 # convert dataset to networkx graph objecT  
 G = nx.from\_pandas\_edgelist(dataset, from\_node, to\_node, edge\_attr=edge\_attr)  
  
 # get edge weights  
 durations = [i[edge\_attr] for i in dict(G.edges).values()]  
  
 # get node labels  
 labels = [i for i in dict(G.nodes).keys()]  
 labels = {i: i for i in dict(G.nodes).keys()}  
  
 # plotting co - dependency map  
 fig, ax = plt.subplots(figsize=(10, 10))  
 pos = nx.spring\_layout(G, scale=0.1, k=0.1)  
 nx.draw\_networkx\_nodes(G, pos, ax=ax, labels=True)  
 weights = nx.get\_edge\_attributes(G, "Current\_Weight")  
 nx.draw\_networkx\_edges(G, pos, width=durations, ax=ax)  
 \_ = nx.draw\_networkx\_labels(G, pos, labels, ax=ax)  
 nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=weights)  
 plt.show()  
  
 return G

# function to evaluate centrality measures of the networkx graph of the predicted co - dependency map  
def get\_predicted\_centrality(centrality\_type, G):  
 return get\_centrality\_measures(centrality\_type, G)

# function to calculate a centrality measures given required centrality type  
def get\_centrality\_measures(centrality\_type, G):  
 if centrality\_type == "degree":  
 return str(get\_degree\_centrality(G))  
 elif centrality\_type == "closeness":  
 return str(get\_closeness\_centrality(G))  
 elif centrality\_type == "betweenness":  
 return str(get\_betweenness\_centrality(G))  
 elif centrality\_type == "load":  
 return str(get\_load\_centrality(G))  
 elif centrality\_type == "eigenvector":  
 return str(get\_eigenvector\_centrality(G))  
 elif centrality\_type == "w-eigenvector":  
 return str(get\_weighted\_eigenvector\_centrality(G, weight='Current\_Weight'))  
 elif centrality\_type == "w-betweenness":  
 return str(get\_weighted\_betweenness\_centrality(G, weight='Current\_Weight'))