A Network Science-Based Approach for an Optimal Microservice Governance

Gihan Saranga Siriwardhana[1], Nishitha De Silva[2], Liyanage Sanjaya Jayasinghe[3],
Lakshitha Vithanage[4], Dharshana Kasthurirathna[5]
Department of Software Engineering, Sri Lanka Institute of Information Technology,
Malabe, Sri Lanka.
[1]*gihanrcg1997@gmail.com*, [2]*nishithadesilva123@gmail.com*, [3]*sanjayajayasinghe54@gmail.com*,
[4]*lmvithanage@gmail.com*, [5]*dharshana.k@sliit.lk*

***Abstract* - With the introduction of microservice architecture for the development of software applications, a new breed of tools, platforms, and development technologies emerged that enabled developers and system administrators to monitor, orchestrate and deploy their containerized microservice applications more effectively and efficiently. Among these vast arrays of technologies, Kubernetes has become one such prominent technology widely popular due to its ability to deploy and orchestrate containerized microservices. Nevertheless, a common issue faced in such orchestration technologies is the employment of vast arrays of disjoint monitoring solutions that fail to portray a holistic perspective on the state of microservice deployments, which in turn, inhibit the creation of more optimized deployment policies. In response to this issue, this publication proposes the use of a network science-based approach to the creation of a microservice governance model that incorporates the use of dependency analysis, load prediction, centrality analysis, and residency evaluation to effectively construct a more holistic perspective on a given microservice deployment. Furthermore, through analysis of the factors mentioned above, the research conducted, then goes on to create an optimized deployment strategy for the deployment with the aid of a developed optimization algorithm. Analysis of results revealed the developed governance model aided through the utilization of the developed optimization algorithm proposed in this publication, proved to be quite effective in the generation of optimized microservice deployment policies.**

Keywords: Auto-scaling, Chaos Engineering, Container, Docker, Kubernetes, Machine Learning, Microservices, Time Series

I. INTRODUCTION

The term "microservices" was first introduced in 2011 [1] and was considered as a specialized implementation of Service-Oriented Architecture (SOA), coined to denote the common architectural approach of decomposing applications into smaller self-contained, loosely coupled services. The microservice architectural style was later widely adopted in place of the traditional monolithic architecture by many leading companies such as Amazon, Netflix, LinkedIn, and SoundCloud due to the capability to develop loosely coupled services possessing the ability to be independently deployed, versioned and scaled, while ensuring in benefits such as faster delivery, more excellent performance, and greater autonomy [1].

The shift in architectural style from the traditional monolithic architecture to microservice architecture also brought forth the creation of a set of new methodologies and approaches that established the policies, standards and best practices for the adoption of microservices, designed for the agile IT environment, known as "Microservices Governance" [2]. This approach to governance was entirely dissimilar to the traditional governance policies followed in monolithic applications mainly since governance in microservices followed a decentralized approach, whereas governance in monoliths followed a centralized approach where decisions were made "top-down" [2]. Although the decentralized approach of governance of microservice provided advantages such as the freedom to develop applications using different technology stacks, a downside of this approach was that more steps should be taken to ensure effective governance is maintained, since typical applications required interconnections between a vast number of microservices where business process workflows were continuously introduced. Consequently, organizations required the service of a variety of tools, ranging from monitoring and autoscaling to others such as configuration management, service discovery, and fault tolerance, that facilitated the multitude of tasks required to ensure effective microservice governance was in effect.

In addition to the tools mentioned above, new deployment strategies that facilitated the newly developing microservice infrastructure were introduced. Among them, containerization of microservices became one of the most effective ways to deploy microservice applications due to its ability to efficiently package microservices by encompassing all the required libraries and dependencies needed during runtime. This procedure separated the application from the underlying infrastructure and enabled developers to run the application in an isolated environment, ensuring performance and functionality. As a result, propelled by services such as Docker, containerization became the preferred approach for effectively deploying microservices, in contrast to the traditional virtualization-based approach previously adopted. However, in the case when the number of microservices of a particular application increased, it became increasingly difficult to coordinate, schedule, monitor, and maintain the required containerized microservices, especially in times where utmost application performance was required. In response to this issue, the Kubernetes framework was introduced in 2014 [3] to allow organizations to run distributed systems more resiliently by providing effective solutions for load balancing, storage, orchestration, automated rollouts, and self-healing mechanisms [4]. The unique characteristics offered by Kubernetes in this regard, thereby made it one of the essential microservice-based technologies available for organizations to deploy their vast arrays of microservice applications in production-grade environments.

The introduction of Kubernetes ushered in a new era of microservice governance through the introduction of container orchestration. Nevertheless, as evident throughout this publication, despite its immense use in orchestrating microservice applications, Kubernetes is still not able to provide a perfect governance solution to most modern

microservice applications, as there are still prevalent issues that need to be addressed in Kubernetes particularly concerning the policies followed in the deployment of interdependent microservices.

A primary reason for the existence of inefficient optimization policies in Kubernetes based microservice deployments is the lack of the tools and services to obtain a holistic view of Kubernetes deployments and thereby optimize cluster performance. The current tools and services offered by Kubernetes often have to be pre-configured to the existing pre-conceived knowledge of the developers in contrast to the actual real-time utilization. Although implementing such solutions may be of use in the short term, it maybe it may be difficult to further improve upon the performance of the microservice cluster in the long term due to the lack of a holistic view on the interaction of the interdependent microservices in real-time use. Hence, it should be realized that if a particular microservice deployment is to be optimized for performance, a clear understanding regarding the relationships among the interdependent microservices during runtime is required. However, if a microservice deployment is to be truly optimized for optimal performance, it may also be necessary to take into account factors such as the resilience among the interdependent microservices, the effect of autoscaling in addition to a clear understanding on the interactions of interdependent microservices. Regardless, even though there are several monitoring solutions available for such purposes, such as Prometheus, Istio, and Chaos toolkit, their disjoint nature prevents them from allowing users to obtain a holistic perspective on the state of their deployed microservices. Furthermore, in cases such as fault management, error handling, and performance monitoring, due to the disjoint nature of these monitoring solutions, users are often unable to gain insight into possible solutions as to why a particular problem or bottleneck has occurred even though they are made aware of the presence of a particular problem by these monitoring solutions.

In addition to the above-mentioned issues, these monitoring solutions are also often and plagued with other challenges such as the difficultly in successfully configuring and integrating these monitoring tools with the existing tools used by organizations [5]. The issues mentioned above may also further complicate the already complicated management and configuration process prevalent in Kubernetes and, in turn, may confuse inexperienced developers and system administrators, ultimately leading towards misallocation of cluster resources and degradation of cluster performance.

In response to the issues stated above, this publication proposes a novel approach to the creation of a unified governance model that can be used by developers and system administrators to effectively oversee the performance of their microservice deployments factoring in dependency analysis, load prediction, centrality analysis, and residency evaluation to determine the optimal placement of microservices and thereby create an optimized deployment plan for a given microservice deployment. Thus, through the application of the proposed governance model, users would be able to obtain a more holistic view of their deployment, resulting in a greater understanding of the runtime behavior of the deployed microservices, thereby enabling greater optimization possibilities. Through application of the approach proposed in this publication, the authors wish to provide key insight to the contribution of a new set of microservice deployment optimization methodologies, which factor in the impact of key factors such as dependency among deployed microservices, autoscaling policies as well as resilience measures in microservice deployments.

The governance model proposed in this publication is comprised of four main components, each aimed at capturing a particular dimension of the microservice deployment with the ultimate goal of achieving a more holistic view of a given microservice deployment. Accordingly, the key components of the proposed model are as follows.

I. A generated microservice co-dependency map which is aimed at obtaining a clear perspective about the dependencies between each microservice and the importance of the deployment plan.

II. A load prediction and centrality analysis component for the prediction of the level of interdependency among co-dependent microservices, the resource utilization of pods in the cluster as well as performing the task of the calculation of centrality measures of microservices in the co-dependency network.

III. A resilience evaluation component to determine the resiliency of microservices in the cluster.

IV. An optimal placement algorithm to determine the optimum placement of microservices in the Kubernetes cluster based on the above stated predicted loads, resiliency, and centrality measures.

The rest of this publication is organized as follows. Section II discusses the background and the related work literature referenced in the development of this optimization model. Section III discusses the architecture of the proposed model, along with an in-depth view of its components. Section IV discusses the methodology followed in the development of the proposed model. Section V discusses the results obtained using the developed model and, finally, the conclusion of this publication, along with directions for future work, is outlined in Section VI.

II. Background and Literature

The apparent need for improved microservice governance modeling strategies, along with some of the prevalent issues in current microservice governance methodologies, have been highlighted in several publications throughout the years. The authors of [6] highlight the need for new modeling strategies that capture the recent advances in deployment technology such as Kubernetes. The publication [7] states the inability of monitoring frameworks to measure microservice performance level metrics would lead to the creation of several new research topics, which include the development of holistic techniques for collecting and integrating monitoring data from microservices and datacenter resources. In contrast, publications such as [1] highlight the use of past actions and events to better inform resource management decisions in microservice environments along with the challenges such as the overloading of monitoring events faced in resource monitoring and management processes.

In addition, several publications have also proposed performance modeling strategies for Kubernetes deployments. In this regard, [8] an architectural approach that federates Kubernetes clusters using a TOSCA-based cloud orchestration tool. In contrast, research publications such as [9] proposed a tool named Terminus to solve the problem of finding the best-suited resources for the microservice to be deployed so that the whole application achieves the best

performance while minimizing the resource consumption. Other researches include the reference net-based model for pod & container lifecycle in Kubernetes proposed by the authors of [10] and the generative platform for benchmarking performance and resilience engineering approaches in microservice architectures as proposed in [11].

The approaches suggested in the publications stated above are all approaches that aim at performance optimization of Kubernetes deployments. However, a key aspect to note in this regard is the fact that the methodologies stated in the publications mentioned above, fail to capture critical dimensions such as the dependent relationships between microservices, the effect of autoscaling policies, resiliency to determine the optimal placement of a particular microservices concerning its global significance. Therefore, to our knowledge, there is no current solution proposed, that takes into consideration an integrated modeling strategy, factoring key elements essential to the optimization of microservice deployments such as co-dependencies present as well resilience and centrality measures among microservices when developing a holistic governance policy for Kubernetes based microservice deployments, as proposed in this research.

The governance model proposed in this publication is primarily aimed at resolving the key issues in present microservice governance methodologies. Hence, a pre-requisite knowledge regarding the nature of these issues was vital in the development of the proposed governance model. The following sub-sections provide an in-depth insight into these key issues as well as the methodologies proposed by fellow publications in the discovery of practical solutions.

A. Microservice Monitoring

Even though Kubernetes and all other tools resolve various problems and improve the functionality of the microservices, there are still some issues and performance bottlenecks either these tools cannot solve, or the tools introduce. This is evident from previous researches such as [12] which describes the drawbacks of Kubernetes when it comes to containerized microservices.

The research conducted by authors such as [13] highlights the importance of having a monitoring solution to monitor the workload and the performance of the cluster. In addition, publications such as [14] describe some of the key challenges faced in the deployment of microservices and the need for Application Performances Monitoring tools, especially those deployed in containers to include additional measures to monitor microservices such that they could use as input for resilience mechanisms and creation of auto-scaling policies.

Even though there are numerous researches about Kubernetes and service mesh, there were negligible publications considering the dependency between microservices as a whole. The available publications go on to describe the auto-deployment facilities of Kubernetes [15]; however, they do not describe how that can affect the network latency between microservice are deployed automatically in the available nodes.

In simple terms, the governance model proposed in this publication will incorporate the calculation of the number of requests to measure the dependency level between each microservice by obtaining a quantifiable value, which then can be used to compare and order the dependency. The obtained values can be used to generate dependency maps in service, pod, and workload levels. The implemented system will use several metric query engines to obtain necessary metrics regarding hardware aspects of pods, nodes, and the cluster and develop a collection of APIs to generate necessary data sources for other components on demand.

B. Rule-based Autoscaling

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. This process is in contrast to scaling in traditional monolithic applications in which, during intense workloads, the entire application stack is scaled, leading to a misallocation of resources. Therefore, as part of its orchestration policy, the Kubernetes framework makes use of autoscaling in order to ensure microservices can adapt to dynamic workload intensities while allowing resources to be provisioned more conservatively.

Autoscaling of resources could be achieved through following a variety of techniques, as stated in [16] however, the autoscaling process in Kubernetes is primarily achieved through the use of Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). Although the governance model proposed in this publication primarily makes use of the HPA, due to the model being primarily focused on the use of existing cluster infrastructure, a common characteristic that these tools possess is the fact that they primarily adopt local and rule-based auto-scaling techniques to dynamically manage the number of microservice resources in a particular deployment.

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 [17]. 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. Furthermore, most rule-based autoscaling policies adapt the MAPE (Monitor, Analyze, Plan, Execute) control loop reference model used in various autonomic computing systems. This MAPE reference model is primarily adopted in orchestration tools such as Kubernetes in the creation of guidelines of self-adaptive software systems [18]. 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 undesirable Quality of Service (QoS) and reduced resource utilization [18], to issues such as the response delay faced in resource creation. Moreover, the lack of autoscaling studies focusing on the service-level of autoscaling and the use of service level metrics, as well as the lack of monitoring tools and aggregating metrics at the platform level and service level to support autoscaling decisions [19], also manage to aggravate the issues mentioned above.

In this regard, numerous publications have proposed a wide array of approaches ranging from proactive autoscaling through resource prediction, to other approaches such as performance modeling, in order to combat some of the issues prevalent in autoscaling process in platforms such as Kubernetes. Fundamental researches include resource consumption prediction models such as those proposed by the

authors [20]–[24], approaches to the creation of improved autoscaling policy as stated in publications such as [25], [26] as well as improved autoscaling frameworks proposed by publications such as [27]–[30]. Regardless, it is evident that even though the solutions for improved autoscaling strategies proposed by these researches were quite effective, they were mostly focused on the creation of localized autoscaling policies without taking int account the overall impact of globally aware autoscaling policy based on importance and the optimal placement of microservices, in contrast to the governance model introduced in this publication.

C. Resilience Evaluation

Failures are inevitable; even the most robust platforms with concrete operations infrastructure could face outages in production when the system's threshold to withstand turbulent conditions go out of control. There is no single reason why a system fails, and it is not possible to immediately address a failure without prior knowledge on why and when that specific failure might occur, the same implies to the widely used Kubernetes platform. Even when all of the individual services in a Kubernetes environment are functioning correctly, the interactions between those services can cause unpredictable outcomes [31].

The concept of chaos engineering was brought up by Netflix to identify its system flaws. Failure injection testing laid the foundation for the emergence of tools such as chaos monkey, chaos toolkit, which allows users to inject failures carefully to the system and examine the behavior [32]. With the use of Kubernetes for microservice deployments, it provided a friendly environment towards chaos engineering practices as it provided native features for resiliency [33]. Regardless, publications such [6] describe some of the critical challenges faced in the deployment of microservices and the need for APM tools, especially those deployed in containers to include additional measures to monitor microservices such that they could be used as input for resilience mechanisms and creation of auto-scaling policies.

When analyzing the optimal deployments of microservices, the fact that resiliency has not been considered as an essential factor [34], Even though fault tolerance and resiliency evaluations have been performed on microservices, the results obtained are only used in the identification of the weaknesses of the system. Resilience evaluation is therefore vital in the determination of critical services in microservice deployment and aid in providing key insight into the determination of optimized deployment strategies.

D. Optimal Microservice Placement

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. 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.

There have been few pieces of research conducted related to microservices optimal deployment, which give prominence to optimal microservice placement. The research conducted by the authors of [35] propose a solution for determining the optimal microservice placement in microservices through analysis of historical values and microservice dependencies, similar to that proposed in this publication; 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, and although the approach suggested in the 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.

III. Proposed Model

As previously stated, the proposed governance model is primarily comprised of four primary components, each aimed at capturing a particular dimension of a given microservice deployment. Fig. 1, given below, depicts a high-level view of the core components of the proposed governance model.

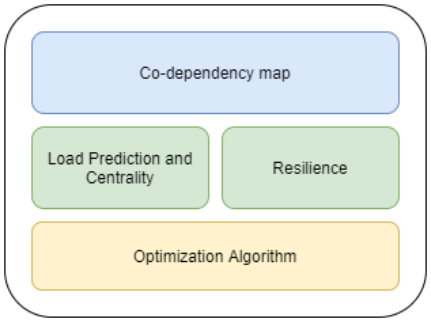


Fig. 1. Key components of the proposed governance model

A. Microservice Co-dependency Network

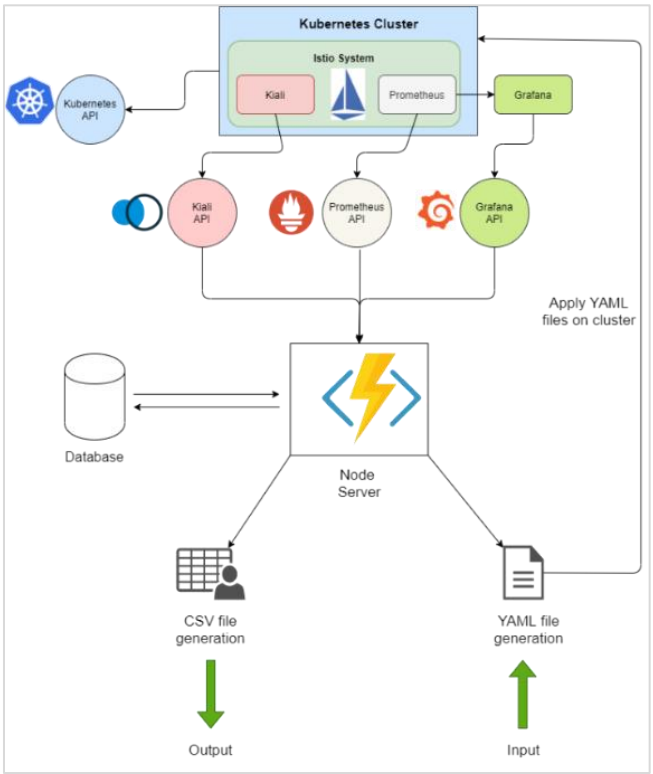


Fig. 2. High-level diagram of components utilized in the creation of the co-dependency network

The microservice co-dependency network component is developed through the combination of the three sub-components as depicted in Fig.2 above.

a. An Istio service mesh platform that incorporates Kiali and Prometheus monitoring solutions.

b. A backend Node server for integration with metric APIs provided by monitoring solutions.

c. A database solution for the storage of gathered metric data.

Istio is an open-source service mesh platform that can be used to monitor and control the behavior of microservices when it comes to sharing data with one another [36]. Istio is thereby be configured on top of the Kubernetes environment to facilitate the creation of the dependency map.

The proposed model will make use of the services and APIs that Istio provides to develop a backend server to query metrics from the Kubernetes cluster. Based on those metrics, the proposed model makes use of a generated dependency map that is utilized to get a holistic perspective of the microservice network. Also, the collected metrics from the cluster is stored in the configured database solution to facilitate the creation of load-based prediction models, as highlighted in the sections below. Furthermore, the proposed model is able to generate a set of YAML files [37] based on the optimal deployment plan and administer the generated YAML files to the cluster with the consent of the user.

B. Load prediction and Centrality Analysis

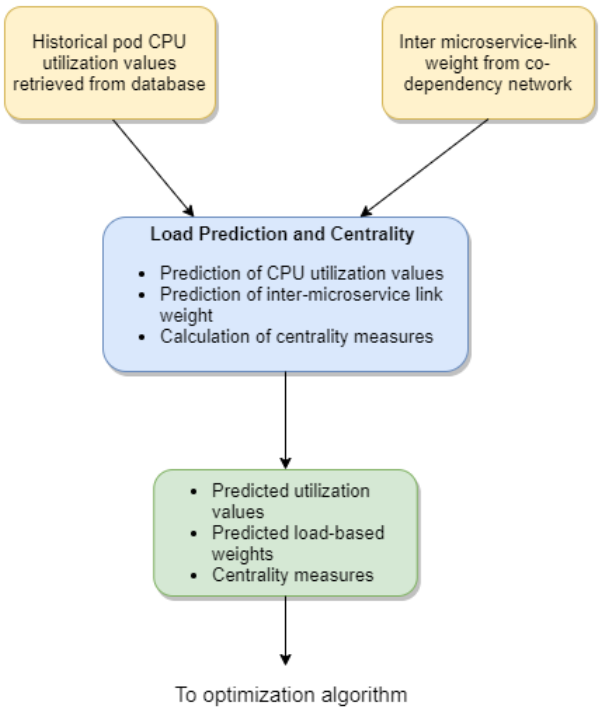


Fig. 3. High-level diagram of load prediction and centrality analysis component

The load prediction component and centrality analysis component are primarily responsible for performing the following essential tasks, as depicted in Fig. 3 above.

- Prediction of future resource utilization values (primarily CPU and memory) based on historical pod resource utilization data.
- Prediction of inter-microservice link weight (dependency measures), based on historical link weight data derived from the load based metrics in the co-dependency network.
- Calculation of centrality measures of microservices in the co-dependency network.

The resource utilization prediction process is performed through performing a time series-based prediction on pod utilization metrics, in which predicted CPU and memory utilization values for a particular period are forecasted. Furthermore, the prediction process for resource utilization is performed through the application of a Short-Term Long Memory (LSTM) network in which a particular number of time steps of utilization metrics are used to predict future utilization values. Here, the LSTM model has been chosen as the preferred prediction model due to its inherent ability to learn long term dependencies with a high degree of accuracy compared to similar time series prediction models. Once predictions are made, the predicted utilization values for a particular period (e.g. - 24 hours in advance) are passed through to the optimization algorithm to infer autoscaling decisions.

The process of inter-microservice link weight prediction is primarily a network-based time-series prediction in which the inter microservice link weights derived through load based metrics, as described in the previous section, are forecasted such that the next predicted weight for the links in the co-dependency map is determined. The forecasted weights determined through the use of an LSTM prediction model could then be used to provide an accurate estimation of the load that is expected to be received by microservices in the cluster, enabling the identification of key potential microservices which may in turn, highly manipulate microservice placement decisions and the realization of optimal cluster performance

Calculation of microservice centrality measures will also be performed within the load prediction component. Here, the microservices in the co-dependency network are evaluated on several centrality measures to facilitate the identification of influential microservices in the cluster. These calculated centrality measures are then sent to the optimization algorithm as inputs, to infer autoscaling decision through determination of required service instance levels. In this regard, the proposed 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.

C. Resiliency Evaluation

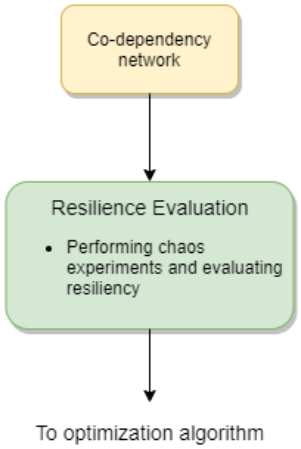


Fig. 4. High-level diagram of resilience evaluation component

The resilience evaluation component is particularly based on chaos engineering principles. Utilizing the co-dependency network, most prominent nodes will be targeted for conducting chaos experiments. In short, this component will primarily be responsible for performing chaos experiments to identify system weaknesses as depicted in Fig.4 above.

Identifying system weaknesses is done using a chaos experimenting tool referred to as chaos toolkit. Here, a set of actions is performed on each node in order to examine how the system behaves according to the changes performed. For instance, a selected pod is terminated using the "terminate pod" action, and the system is examined to see whether the system remains available or the other services remain healthy. Furthermore, by using chaos toolkit extensions such as Istio Fault Injection, delays can be created to investigate how the system responds.

To obtain metrics such as CPU usage, disk space, and bandwidth, we could use the chaos toolkit-Prometheus extension. With the use of this extension, the above-mentioned metrics can be extracted while applying different conditions to the system. Moreover, various experiments are done to create different conditions for the system such that the resilience of the system can be determined precisely.

D. Optimization Algorithm

The optimization algorithm utilized in the proposed governance model is predominantly based on the NSGA-II (Non-Dominant Sorting Genetic Algorithm) algorithm. The algorithm generates a multitude of optimized solutions that enables the user to infer optimization decisions predicated on three key optimization categories, which are as follows.

- Solutions optimized for best performance and availability, thereby maintaining a balance between reduced latency and number of instances.
- Solutions optimized for optimal performance based on the reduction of latency.
- Solutions optimized for highest availability based on the maximization of the number of instances.

These optimized solutions are generated following four main input parameters utilized by the optimization algorithm.

1. Predicted microservice dependency measures from the load prediction and centrality analysis component.
2. Node latency map generated from the Node Server.
3. Required number of microservice instances derived from centrality measures and predicted resource utilization metrics from the load prediction and centrality analysis component.
4. General cluster infrastructure information gathered from monitoring solutions.

The sub-sections below provide an in-depth insight into the manner these input parameters are utilized in the developed algorithm as well as their impact on the creation of holistic optimization policies.

I. Predicted Microservice Dependency Measures

In microservice deployments, although factors such as latency cannot be completely eliminated, dependent microservices can be deployed in nearby nodes or the same node in order to reduce the overall latency of an application. Therefore, making use of this approach while intending to solve low availability and sub-optimal performance issues, as well as to aid in the creation of autoscaling policies, the developed optimization algorithm makes use of the predicted load-based link weights obtained from the load prediction component. This is done such that optimal placement and scaling decisions could be performed ahead of time, establishing a future deployment strategy such that users such as DevOps engineers would be able to make use of the gathered information to create an optimized microservice deployment plan. In addition, making use of the predicted dependency measures (load-based link weight), optimal placement decisions are determined through the application of formula (1) and (2), as defined below, which calculates the average latency among the microservice instances, based on the dependency measures and as the node latency map obtained from the Node server.

TABLE I. AVERAGE LATENCY 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

$$D_j = \frac{\sum_{j=1}^{i=m} L_i}{n} \quad (1)$$

$$Minimize\ TL = \sum_{j=1}^{j=m} W_j \times D_j \quad (2)$$

II. *Node Latency Measures*

The main objective of the optimization algorithm is the maximization of performance through the minimization of latency among microservices. Therefore, the developed optimization algorithm also utilizes a developed node latency map obtained from the Node Server, to evaluate the fitness of generated solutions.

III. *Required Microservice Instances*

In the process of fitness calculation, the first step is the calculation of the required number of instances per microservices. Here, the calculation of the required number of microservices instances is performed by utilizing the predicted resource utilization values derived from the load prediction component, applied on the Horizontal Pod Autoscaling algorithm. Also, the centrality measures derived from the co-dependency network will be utilized to infer the optimum microservice instance levels, particularly in cases where historical information of the cluster is unknown. The required microservice instance levels are also utilized in availability fitness calculation measures, aided through the use of a generalized logistic function [38] to avoid giving high scoring fitness values from resources that require low resource consumption and are of low instance levels, thereby establishing a fairer scoring method. In this regard, the fitness is calculated as defined through the formula (3) given below.

TABLE II. FITNESS CALCULATION

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

$$Maximize\ TA = \sum_{i=1}^{i=n} R_i \times generalizedLogisticFunction\left(\frac{S_i}{R_i}\right) \quad (3)$$

The fitness function also makes use of a scoring system based on the distribution of the number of instances deployed on cluster node resources known as the scale value. In this regard, a higher number of instances distributed among cluster nodes throughout the deployment are given a higher score than localized instances deployed within a single node. This task is performed to avoid convergence of dependent services into one node and affecting availability. These scale values are then utilized to infer performance and availability decisions.

IV. *General Cluster Information*

The optimization algorithm also makes use of the general cluster infrastructure information such as the resource power consumption of nodes and node labels names. The information gathered in this regard is primarily utilized in the definition of constraints utilized by the optimization algorithm.

IV. METHODOLOGY

Fig. 5 below presents an overview of the implemented governance model along with its key components.

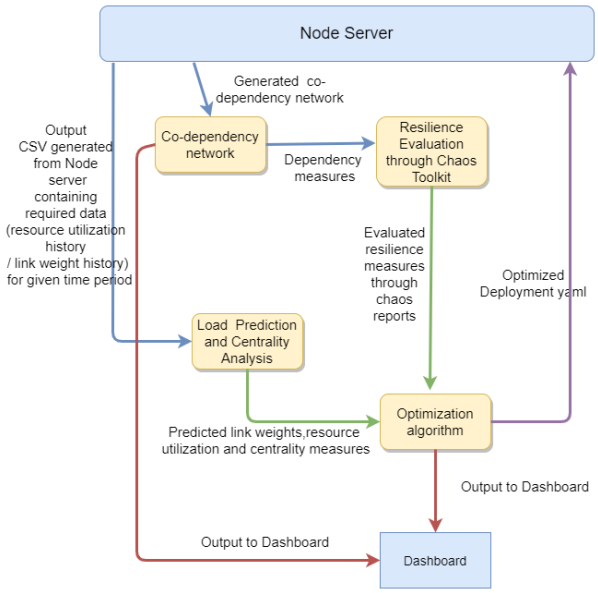


Fig. 5. Overview of the implemented solution

A. *Developing the Microservice Co-dependency Map*

This component primarily queries metrics from a Kubernetes cluster and saves the gathered metrics in order to create a dataset such that it could be used to build a co-dependency network. As previously stated, the Istio service mesh is installed and configured in order to get data from the cluster, in the app, pod, and node levels. Istio also comes with a set of pre-installed services. In this regard, Kiali, Prometheus, and Jaeger are some of the primary services utilized in the development process of this component. Grafana is also configured using Prometheus as a data source. Each of these services exposes APIs that can be used to gather information via HTTP requests. Also, Kubernetes itself exposes an API that can be used to query data about the cluster and the environment.

A Node server is also developed to combine all these APIs to query metrics from a single endpoint. Node JS is used as the development framework for this server in order to maintain the necessary speed and flexibility required. The server can be configured to query metrics and trigger the process based on a scheduler. The default option will is to run the process once every three hours. The collected metrics are then stored in a No SQL database to maintain simplicity. In addition, a timestamp is stored with every record to create a time series dataset that will be used in training machine learning models for more accurate results. The developed Node server is also capable of generating CSV (Comma Separated Values) files on demand of the other components by reading the above stated No SQL database. The server will also expose an endpoint that can be accessed via an HTTP request in order to trigger required functions on demand. All the data stored in the database is maintained within the same Kubernetes cluster without exposing it to the public in order to maintain the privacy of user data. Lastly, in addition to the above, the Node server is also responsible for the creation of a node latency map through evaluating the latencies between the nodes in the cluster. Here, the Round-Trip Time (RTT) of

network calls between nodes in the cluster is evaluated and, through the use of a developed shell script, the average latency measures between cluster nodes are obtained and forwarded to the optimization algorithm.

As mentioned earlier, one of the critical components, which is the generation of the microservice dependency map, will be done by using Kiali to record the requests and responses between each service for a specified period. That measurement can then be divided by the time to quantify the dependency level. The generated CSV files and the quantified dependency map is then used as inputs in all other components in this research. Furthermore, the dependency map is displayed in the final dashboard to provide a more holistic view of the microservice network to the user.

As the server developed in this component has a direct connection with the Kubernetes cluster, the optimal deployment plans, which are given to the optimization algorithm component of this research, are used to generate a set of YAML files that can be applied to the cluster directly in order to change the deployment plan of the cluster. This will not be an automated process, and the user will be given a choice to apply these deployment changes to the cluster or not.

B. Developing the Load Prediction and Centrality Analysis Component

As previously stated, this component is primarily developed for the prediction of load-based link weights, prediction resource utilization metrics, as well as the calculation of centrality measures on the developed co-dependency network. Thereby, a key objective of this component is the utilization of historical data and centrality measures to aid in the optimization of microservice deployments and the creation of holistic autoscaling policies.

The entirety of the above-stated tasks is performed through the use of the Python programming language mainly due to its immense flexibility and adaptability 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. Hence, the LSTM prediction model employed for the resource utilization prediction and link-weight prediction purposes in this component is primarily developed through the utilization of Keras and TensorFlow python libraries. In contrast, the NetworkX python library is utilized for the calculation of centrality measures on the co-dependency network.

For the prediction process, a time series-based approach is adopted to acquire the knowledge present in historical data. Therefore, an LSTM prediction model is utilized for this purpose. Furthermore, since the prediction of time series-based values using LSTM models requires a data-science based approach to obtain the expected predictions, a pre-requisite data manipulation process is required to be performed to obtain the most accurate prediction results. Fig. 6, below depicts the process followed in the prediction of resource utilization metrics, whereas Fig. 7 given below depicts the process followed in the prediction process for inter microservice-link weights.

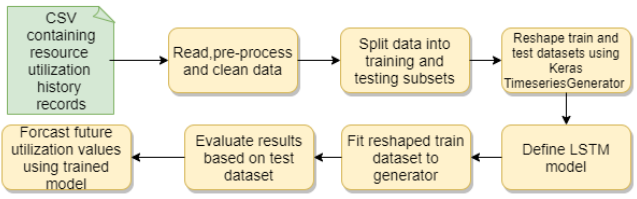


Fig. 6. Prediction process for resource utilization

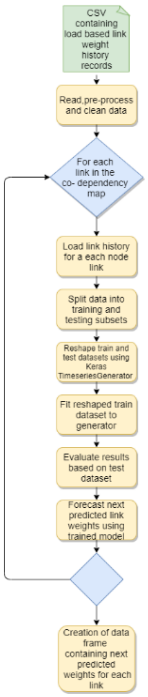


Fig. 7. Prediction process for inter microservice link weight (dependency)

The process of the calculation of centrality is also performed through a data science-based approach in which a CSV containing microservice-link data generated from the Node Server is utilized in the process of centrality calculation. Fig. 8 given below depicts the process followed in the calculation of centrality measures of microservices in the co-dependency network.

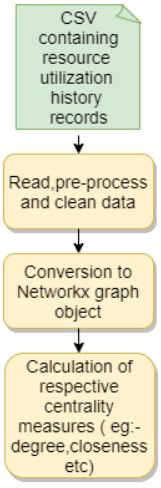


Fig. 8. The process followed in the calculation of centrality measures

C. Developing the Resilience Evaluation Component

As noted earlier, this component mainly focuses on evaluating the resilience level of the system. The process is performed under the principles of chaos engineering which are as follows.

1. Building a hypothesis around steady-state behavior
2. Vary real-world events
3. Run experiments in production
4. Automate experiments to run continuously
5. Minimize blast radius

The main tool used for conducting these experiments is the chaos toolkit. Experiments are scripted in JSON or YAML formats, in which various actions and probes are defined to create different conditions in the system. A python virtual environment is used to host the chaos toolkit and after the configuration of the Kubernetes context, the chaos experiments are executed by running the scripts. The outputs of these experiments are configured as needed to exploit the needed metrics, and the results are generated as CSV or pdf files.

D. Developing the Optimization Algorithm

The entirety of the development of the NSGA-II based optimization algorithm is performed through the use of the Python code scripting. Fig. 9 given below depicts an overview of the process utilized by the optimization algorithm in the determination of optimal solutions.

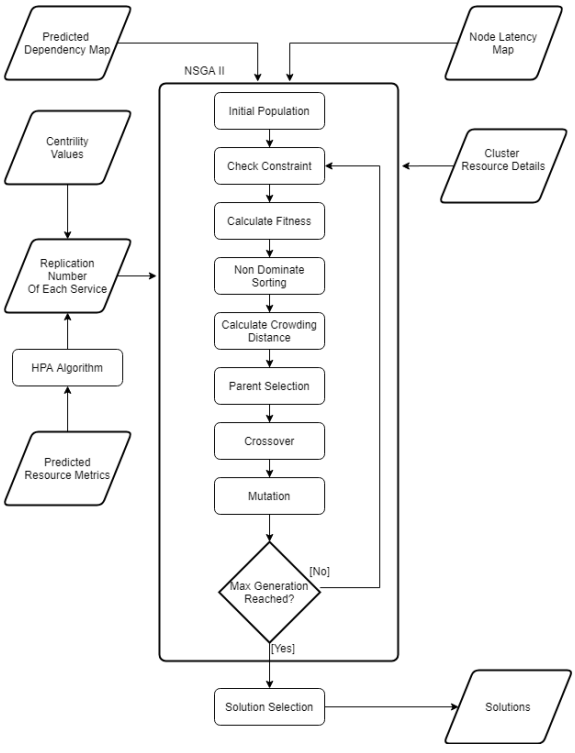


Fig. 9. Overview of the optimization algorithm

As depicted in Fig. 9, once all input parameters are retrieved by the optimization algorithm, the first step in the process is the generation of the initial population. Here, the initial population is generated through the use of a random number generator which inserts data into a pre-defined two-dimensional array in which the microservice instances and their respective nodes are represented by the row and columns respectively. Fig. 10 depicts a sample overview of the structure of the generated two-dimensional array.

Services|Nodes	*Node0*	*Node1*	*Node2*
Microservice 0	6	0	0
Microservice 1	0	0	5
Microservice 2	4	0	0
Microservice 3	0	2	2
Microservice 4	0	4	0
Microservice 5	0	6	0

Fig. 10. Overview of the structure of the generated two-dimensional array

After the creation of the initial population, constraints are applied to the initial population to eliminate the invalid solutions generated. Next, fitness is computed from the remaining solutions and the superior parent chromosomes are selected based on the evaluated fitness measures. The selected parent chromosomes are then ranked using non-dominant sorting, and crowding distance measures and the highest-ranked parent chromosomes are utilized for crossover. The above-described process is then run iteratively until the maximum generation count is reached. Lastly, the solutions obtained through the above-described process are then saved in a python data frame such that the optimized solutions based on three key optimization categories, as stated in the previous section, could be retrieved when needed.

V. Results and Discussion

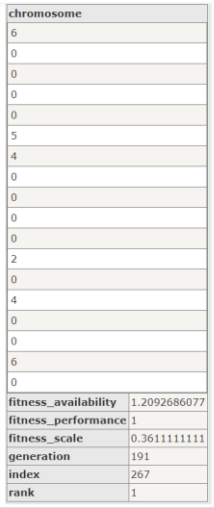
The developed optimization model was evaluated on a sample microservice cluster dataset containing 3 nodes and 6 microservices. For evaluation purposes, the JSON (JavaScript Object Notation) representation of this cluster dataset, along with the additional information required by the optimization algorithm 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.11 below depicts the structure of the sample input JSON provided to the optimization algorithm.

{"nodes": {"0": {"ip": "", "cpu": 4000, "memory": 8000, "storage": 500000}, "1": {"ip": "", "cpu": 2000, "memory": 4000, "storage": 500000}, "2": {"ip": "", "cpu": 4000, "memory": 8000, "storage": 500000}}, "pods": {"0": {"name": "", "cpu": 300, "memory": 200, "storage": 300}, "1": {"name": "", "cpu": 300, "memory": 200, "storage": 300}, "2": {"name": "", "cpu": 200, "memory": 200, "storage": 300}, "3": {"name": "", "cpu": 200, "memory": 200, "storage": 300}, "4": {"name": "", "cpu": 200, "memory": 200, "storage": 300}, "5": {"name": "", "cpu": 200, "memory": 200, "storage": 300}}, "nod_latencies": {"[0, 1]": 20, "[0, 2]": 30, "[1, 2]": 20, "[0, 0]": 1, "[1, 1]": 1, "[2, 2]": 1}, "pod_dependency_map": {"[0, 2]": 1000, "[1, 3]": 500, "[4, 5]": 200}, "pod_importance": "", "microservice_instances_requirement": [4, 4, 2, 2, 2, 2]}

Fig. 11. Structure of sample JSON provided as input to the developed optimization algorithm

Once the optimization algorithm is executed, a set of optimized solutions are obtained. In this regard, two optimized solutions are obtained once the algorithm is executed; one solution represents the cluster orientation with the highest cluster performance as depicted in Fig.12, whereas the second solution obtained depicts the solution that represents the cluster orientation with the highest cluster

availability as depicted in Fig.13. For added clarification, the tabular format of the representation is given alongside the resulting solutions.

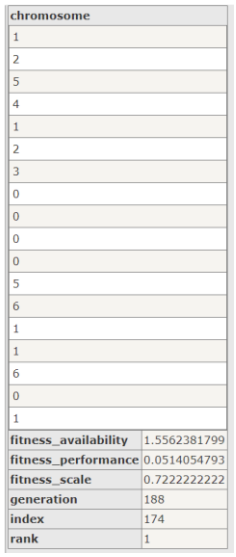


chromosome
6
0
0
0
0
5
4
0
0
0
0
2
0
4
0
0
6
0

fitness_availability	1.2092686077
fitness_performance	1
fitness_scale	0.3611111111
generation	191
index	267
rank	1

Micro-services	Nodes		
	Node 0	*Node 1*	*Node 2*
M0	6	0	0
M1	0	0	5
M2	4	0	0
M3	0	0	2
M4	0	4	0
M5	0	6	0

Fig. 12. Resulting solution representing cluster orientation with the highest performance



chromosome
1
2
5
4
1
2
3
0
0
0
0
5
6
1
1
6
0
1

fitness_availability	1.5562381799
fitness_performance	0.0514054793
fitness_scale	0.7222222222
generation	188
index	174
rank	1

Micro-services	Nodes		
	Node 0	*Node 1*	*Node 2*
M0	1	2	5
M1	4	1	2
M2	3	0	0
M3	0	0	5
M4	6	1	1
M5	6	0	1

Fig. 13. Resulting solution representing cluster orientation with the highest availability

Note the fact that in the tabular format depicted in Fig. 12 and Fig. 13, each cell in the table represents the optimal number of instances of a given microservice that should be present in order to achieve the required optimization goal (highest performance or highest availability).

With regard to the resulting solution obtained that represents the cluster orientation with the highest performance, the fact that the optimization algorithm has successfully managed to determine the cluster orientation with the highest performance is evident mainly due to the fact that the highest dependent services as provided in the input JSON have been determined to be placed on the same node by the optimization algorithm. This fact is determined through comparing the keys of the key-value pair sets in the "pod_dependency_map" feature of the input JSON which represents inter-dependent sets of microservices (For example: - " [0, 2]: 1000 " in the input JSON represents microservice M0 and microservice M2 are inter-dependent microservices with a dependency level of 1000), with the tabular representation of the resulting optimal performance solution, that also depicts the inter-dependent microservices as described in the input JSON (such as M0 and M2) placed on the same node.

Similarly, through comparing the "microservices_instances_requirement" feature of input JSON which represents the required number of instances required for each of the six microservices respectively, with the resulting instance levels obtained from resulting highest availability solution, it is evident that the optimization algorithm has also ensured highest availability of microservices through the allocation of a higher number of microservice instances than the required instances. (For example - Microservice M0 requires the presence 4 instances and optimization algorithm has allocated 8 instances of M0 as determined through its optimization process)

VI. Conclusion

This publication suggests the application of a network-science based microservice governance model in an attempt to aid in the creation of optimized microservice deployment policies currently hindered due to the employment of disjoint monitoring solutions prevalent in microservice-based governance methodologies that fail to portray a holistic perspective regarding the status of microservice deployments. In this regard, the proposed model seeks the creation of a holistic perspective of microservice deployments, through the incorporation of dependency analysis, load prediction measures, centrality measures as well as resilience measures. Furthermore, through the incorporation of the above measures, the research conducted utilizes the application of an optimization algorithm to determine an optimal deployment strategy for a given microservice deployment.

The publication also discusses the core architecture along with the methodologies followed in the development of the proposed governance model as well as the results obtained through the application of the proposed governance model. Analysis of the results suggests the developed governance model proved to be effective in determining the optimized cluster representations pertaining to the highest performance and availability. However, current research suggests considering the inner workings of applications deployed in a Kubernetes cluster so as to increase the accuracy of the prediction models and resiliency analysis components such that more optimized deployment policies can be established.

Acknowledgment

Gihan Siriwardhana, Nishitha De Silva, Sanjaya Jayasinghe, Lakshitha Vithanage would like to thank the Sri Lanka Institute of Information Technology (SLIIT) and the Department of Software Engineering for providing the opportunity to conduct this research as well as Dr. Dharshana Kasthurirahtna for guiding and inspiring the development of this research.

References

[1] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," IEEE Software, vol. 35, no. 3, pp. 24–35, 2018, doi: 10.1109/MS.2018.2141039.

[2] "Microservices Governance: A Detailed Guide." https://www.leanix.net/en/blog/microservices-governance (accessed Jun. 16, 2020).

[3] "Kubernetes. – Wikipedia." https://en.wikipedia.org/wiki/Kubernetes. (accessed Jun. 16, 2020).

[4] "What is Kubernetes? | Kubernetes." https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (accessed Jun. 16, 2020).

[5] "Kubernetes: The Challenge of Deploying & Maintaining." https://techolution.com/kubernetes-challenges/ (accessed Jun. 16, 2020).

[6] R. Heinrich et al., "Performance engineering for microservices: Research challenges & directions," ICPE 2017 – Companion of the 2017 ACM/SPEC International Conference on Performance Engineering, pp. 223–226, 2017, doi: 10.1145/3053600.3053653.

[7] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open Issues in Scheduling Microservices in the Cloud," IEEE Cloud Computing, vol. 3, no. 5, pp. 81–88, 2016, doi: 10.1109/MCC.2016.112.

[8] D. Kim, H. Muhammad, E. Kim, S. Helal, and C. Lee, "TOSCA-based and federation-aware cloud orchestration for Kubernetes container platform," Applied Sciences (Switzerland), vol. 9, no. 1, 2019, doi: 10.3390/app9010191.

[9] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," ICPE 2019 – Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, pp. 25–32, 2019, doi: 10.1145/3297663.3310309.

[10] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui, "Modelling performance & resource management in Kubernetes," Proceedings – 9th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2016, pp. 257–262, 2016, doi: 10.1145/2996890.3007869.

[11] T. F. Düllmann and A. van Hoorn, "Model-driven generation of microservice architectures for benchmarking performance & resilience engineering approaches," ICPE 2017 – Companion of the 2017 ACM/SPEC International Conference on Performance Engineering, pp. 171–172, 2017, doi: 10.1145/3053600.3053627.

[12] A. Modak, S. D. Chaudhary, P. S. Paygude, and S. R. Ldate, "Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes?," Proceedings of the International Conference on Inventive Communication and Computational Technologies, ICICCT 2018, no. Icicct, pp. 7–12, 2018, doi: 10.1109/ICICCT.2018.8473104.

[13] "Comprehensive Container-Based Service Monitoring with Kubernetes and Istio | Circonus." https://www.circonus.com/2018/06/comprehensive-container-based-service-monitoring-with-kubernetes-and-istio/ (accessed Jul. 13, 2020).

[14] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service Mesh: Challenges, state of the art, and future research opportunities," Proceedings – 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, 10th International Workshop on Joint Cloud Computing, JCC 2019 and 2019 IEEE International Workshop on Cloud Computing in Robotic Systems, CCRS 2019, pp. 122–127, 2019, doi: 10.1109/SOSE.2019.00026.

[15] "One year using Kubernetes in production: Lessons learned." https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned (accessed Jul. 13, 2020).

[16] 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, vol. 20, no. 2, pp. 399–432, 2019, doi: 10.12694/scpe.v20i2.1537.

[17] A. Evangelidis, D. Parker, and R. Bahsoon, "Performance modelling and verification of cloud-based auto-scaling policies," Future Generation Computer Systems, vol. 87, pp. 629–638, 2018, doi: 10.1016/j.future.2017.12.047.

[18] 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, no. September 2019, p. 102734, 2020, doi: 10.1016/j.advengsoft.2019.102734.

[19] A. Hanieh, L. Yan, and H.-L. Abdelwahab, "Analyzing Auto-scaling Issues in Cloud Environments," Proceedings of 24th Annual International Conference on Computer Science and Software Engineering. IBM Corp., no. January, pp. 75–89, 2014.

[20] A. Zhao, Q. Huang, Y. Huang, L. Zou, Z. Chen, and J. Song, "Research on Resource Prediction Model Based on Kubernetes Container Auto-scaling Technology," IOP Conference Series: Materials Science and Engineering, vol. 569, no. 5, 2019, doi: 10.1088/1757-899X/569/5/052092.

[21] H. Zhao, H. Lim, M. Hanif, and C. Lee, "Predictive Container Auto-Scaling for Cloud-Native Applications," ICTC 2019 – 10th International Conference on ICT Convergence: ICT Convergence Leading the Autonomous Future, pp. 1280–1282, 2019, doi: 10.1109/ICTC46691.2019.8939932.

[22] Y. Meng, R. Rao, X. Zhang, and P. Hong, "CRUPA: A container resource utilization prediction algorithm for auto-scaling based on time series analysis," PIC 2016 – Proceedings of the 2016 IEEE International Conference on Progress in Informatics and Computing, pp. 468–472, 2017, doi: 10.1109/PIC.2016.7949546.

[23] 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 2017, 2017, doi: 10.1145/3022227.3022243.

[24] T. Ye, X. Guangtao, Q. Shiyou, and L. Minglu, "An Auto-Scaling Framework for Containerized Elastic Applications," Proceedings – 2017 3rd International Conference on Big Data Computing and Communications, BigCom 2017, pp. 422–430, 2017, doi: 10.1109/BIGCOM.2017.40.

[25] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," Computer Journal, vol. 62, no. 2, pp. 174–197, 2019, doi: 10.1093/comjnl/bxy043.

[26] 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, 2017, doi: 10.1007/s00607-016-0530-9.

[27] 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, 2017, doi: 10.1007/s10586-016-0682-6.

[28] P. P. Kukade and G. Kale, "Auto-Scaling of Micro-Services Using Containerization," International Journal of Science and Research (IJSR), vol. 4, no. 9, pp. 1960–1964, 2013.

[29] C. Kan, "DoCloud: An elastic cloud platform for Web applications based on Docker," International Conference on Advanced Communication Technology, ICACT, vol. 2016-March, pp. 478–483, 2016, doi: 10.1109/ICACT.2016.7423440.

[30] L. R. Moore, K. Bean, and T. Ellahi, "A Coordinated Reactive and Predictive Approach to Cloud Elasticity," CLOUD COMPUTING 2013, The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization, no. c, pp. 87–92, 2013.

[31] "Improving Kubernetes Resiliency with Chaos Engineering | by Gokul Chandra | FAUN | Medium." https://medium.com/faun/failures-are-inevitable-even-a-strongest-platform-with-concrete-operations-infrastructure-can-7d0c016430c6 (accessed Jul. 13, 2020).

[32] "Chaos engineering – O'Reilly." https://www.oreilly.com/content/chaos-engineering/ (accessed Jul. 13, 2020).

[33] "How chaos engineering will guarantee the resilience of your services – eldermoraes.com." https://eldermoraes.com/how-chaos-engineering-will-guarantee-the-resilience-of-your-services/ (accessed Jul. 13, 2020).

[34] H. F. E. B, S. Gerasimou, and A. Sen, DeepFault : Fault Localization, vol. 1, no. 2. Springer International Publishing, 2019.

[35] A. R. Sampaio, J. Rubin, I. Beschastnikh, and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," Journal of Internet Services and Applications, vol. 10, no. 1, 2019, doi: 10.1186/s13174-019-0104-0.

[36] "What is Istio?" https://www.redhat.com/en/topics/microservices/what-is-istio (accessed Jul. 15, 2020).

[37] "Introduction to YAML: Creating a Kubernetes deployment | Mirantis." https://www.mirantis.com/blog/introduction-to-yaml-creating-a-kubernetes-deployment/ (accessed Jul. 15, 2020).

[38] "Generalised logistic function – Wikipedia." https://en.wikipedia.org/wiki/Generalised_logistic_function (accessed Jul. 14, 2020).