National College of Ireland

Kubernetes Proactive Resources Scheduling using Multi-armed bandit Algorithm

MSc Research Project
MSc Cloud Computing

Nikhil Mondhe
Student ID: x21174105

School of Computing
National College of Ireland

Supervisor: Rejwanul Haque

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Nikhil Mondhe
Student ID:	x21174105
Programme:	MSc Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Rejwanul Haque
Submission Due Date:	14/08/2023
Project Title:	Kubernetes Proactive Resources Scheduling using Multi-armed bandit Algorithm
Word Count:	5217
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Nikhil Mondhe
Date:	13th August 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Kubernetes Proactive Resources Scheduling using Multi-armed bandit Algorithm

Nikhil Mondhe
x21174105

Abstract

Effective resource allocation in Kubernetes clusters is critical for improving application performance while utilising cluster resources efficiently. Traditional resource scheduling approaches frequently rely on fixed policies, which results in suboptimal resource utilisation and, on occasion, performance bottlenecks. To address these issues, this study employs the Multi-armed Bandit Algorithm in a novel approach for proactive resource scheduling in Kubernetes environments. To dynamically allocate resources to different application workloads, the proposed approach employs Thompson Sampling algorithm, a popular technique in the field of multi-armed bandits. The algorithm seeks to strike a balance between exploring potentially better resource configurations and exploitation of known high-performing configurations by treating each application as a "arm" and allocating resources based on historical performance data. The objective of this study is to compare the effectiveness of the Thompson Sampling-based proactive scheduling approach to the default Kubernetes scheduler. The comparison is based on minimising CPU and memory usage across multiple workloads. The findings of this study have the potential to make a significant contribution to the field of container orchestration and resource management by providing insights into the effectiveness of advanced algorithms in solving resource allocation challenges. The experimental results demonstrated that proactive resource scheduling strategies can improve the overall scalability, performance, and efficiency of Kubernetes clusters.

1 Introduction

Cloud computing has changed how businesses and people manage, store, and run applications and data. It involves delivering on-demand computing resources like computing power, storage, databases, and different services over the internet. Users can rent resources from cloud service providers instead of owning physical infrastructure, which gives them scalability, cost-effectiveness, and flexibility. Different service models, like Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), are used in cloud computing to meet different needs. Because it is flexible, organisations can quickly add or take away resources as needed. This helps them get the most out of their resources and save money. Cloud computing also lets users access apps and data from anywhere with an internet connection. This makes it possible for workers to be mobile and flexible. Also, the cloud's dependability, availability, and security, along with its advanced data management and analytics features, make it a must-have technology for modern businesses that want to thrive in the digital age.

In recent years, cloud computing has made a lot of progress to meet the growing needs of complex and varied workloads. Containerization technology is one of the most well-known and widely accepted new trends. Containers are a modern way for developers to package applications and all of their configurations and dependencies. They offer a consistent environment that is light and easy to move around throughout the software development lifecycle, which includes development, testing, and production. Docker, the most popular platform for containerization, has helped make this technology more popular by making it easy to create, deploy, and manage containers.

Kubernetes, an open-source platform for managing containers, is another important change in this area. Kubernetes automates the deployment, scaling, and management of containerized applications. This makes it easier to handle complex workloads from multiple applications. With Kubernetes, developers can take advantage of advanced features like load balancing, self-healing, and declarative configuration, which make it easier to manage large-scale container deployments.

Serverless computing is a new trend that is also worth mentioning. This paradigm lets developers focus only on writing code and not worry about the infrastructure underneath. Cloud providers take care of the infrastructure and automatically scale up or down applications based on demand. Serverless computing simplifies operations and lets developers speed up deployment cycles and make better use of resources.

Using a microservices architecture has also changed how applications are built and put into use. With microservices, an application is broken up into smaller, independent services that are each in charge of a certain set of functions. APIs let these services talk to each other, which makes them more modular and flexible. This method makes it easier to develop, deploy, and maintain multi-application workloads because each service can be updated and scaled on its own. This improves the overall agility and scalability of the system.

1.1 Motivation

Containerized applications have revolutionised software deployment by enabling efficient computing resource utilisation and rapid scalability. Kubernetes, a leading container orchestration platform, has played a critical role in streamlining containerized workload deployment and management. Effective resource management in Kubernetes clusters,

on the other hand, remains a challenge, with suboptimal resource allocation affecting application performance, resource utilisation, and overall cluster efficiency. Traditional resource scheduling approaches frequently rely on predefined static allocation policies that lack adaptability to dynamic changes in application demands. These approaches can result in over-provisioning, which wastes resources, or under-provisioning, which causes performance bottlenecks and lower application responsiveness. The need for intelligent and adaptive resource scheduling mechanisms becomes more apparent as containerized environments become more diverse and dynamic.

The motivation for this research is to address these resource allocation challenges by utilising innovative machine learning techniques, specifically the Multi-armed Bandit Algorithm. This research aims to revolutionise the way Kubernetes clusters manage resources by taking a proactive approach to resource allocation, where decisions are guided by historical performance data and ongoing exploration.

1.2 Research Question

How can the Thompson sampling algorithm be used to solve the multi-armed bandit problem in Kubernetes to optimise application performance while minimising resource usage and CPU usage?

1.3 Paper Structure

This report is broken up in seven sections for simplicity. This section 2 present a review of related work in the fields of Container Orchestration, Kubernetes Resource Management, and Kubernetes Pod Allocation, as well as a comparison of the algorithms that have been proposed for these areas. In Section 3, we will discuss the research methodology as well as the outline of the proposed system. The design specification and overall system architecture are discussed in Section 4. In Section 5, we will discuss how to put the suggested strategy into action. In sections 6 and 7, the results and conclusions of the experiment are presented.

2 Related Work

2.1 Containers Orchestration

Containerisation has the advantage of distributed an application into manage themselves into modules that can be handle from anywhere remotely, as well as splitting the dependency connected with this packages. The use of containers results in more agile software development cycles. Despite considering that there are several other possible implementations, Docker is most commonly used container tool in cloud computing. Architectures are built on groups of containerized service instances that collaborate. The containers have to be able to tolerate errors, be distributed across multiple locations, and have high availability. Khan (2017) details the capabilities that container orchestration tools really ought to possess. In their study, they presented a plan for recognising important processes for methods to implement containers and orchestration frameworks. This plan was outlined as a design. Within their methodology, they have also described orchestration platforms that come highly recommended and are freely available. When compared to other orchestration tools, Khan (2017) and Jawarneh et al. (2019) found that Kubernetes

performed better than Apache Mesos, Docker Swarm, and Mesosphere for the deployment of critical application. These studies attempted to fill a knowledge gap regarding the instrument that provides IT administrators with the greatest degree of success when performing orchestration tasks. According to the findings of the researchers, Kubernetes is the orchestration tool that is the most effective when it comes to the deployment of more complicated applications, whereas other tools are more effective when it comes to the deployment of applications that are less complicated. On the other hand, applications that call for a significant amount of computing power might benefit from the utilisation of orchestration tools. With the help of these tools, it will be possible to have a faster response time and an improved quality of service. For the purposes of this investigation, we will orchestrate containers using Kubernetes due to the fact that its performance is superior when applied to applications that are more complex.

2.2 Kubernetes Resource Management

As a powerful container orchestration platform, Kubernetes is critical for automating the deployment, scaling, and management of containerized applications. Effective resource management in Kubernetes is critical for improving performance, ensuring efficient resource utilisation, and dealing with fluctuating workloads. Kubernetes includes a number of resource management techniques and algorithms, each with its own set of advantages and disadvantages. Static resource allocation is a traditional approach to resource management in which resource requests and limits are pre-configured for each Kubernetes Pod. These resource requests specify the minimum CPU and memory requirements for a Pod, whereas resource limits specify the maximum resource consumption. These predefined values are used to schedule pods. While static allocation allows for resource isolation, it lacks flexibility in dealing with varying workloads because the allocated resources remain constant regardless of actual usage. This approach may result in underutilization of resources during periods of low demand and poor application performance during peak times. Furthermore, overly conservative limits may result in unnecessary scaling or underutilization of cluster resources.

Dynamic resource allocation Arora and Ksentini (2021)is used to address the limitations of static resource allocation. The Horizontal Pod Autoscaler (HPA) in Kubernetes is a prime example of dynamic allocation. The HPA adjusts the number of Pod replicas automatically based on metrics such as CPU utilisation or custom metrics. Dynamic allocation, as opposed to static allocation, allows for greater adaptability to changing workloads Chang et al. (2017). However, it is still a reactive strategy that responds to shifting resource demands. Because of this reactive nature, slight delays in scaling up or down can occur, potentially affecting application performance during rapid workload fluctuations. Furthermore, the accuracy of scaling decisions is heavily dependent on the metrics used, making it critical to choose appropriate metrics to ensure efficient resource allocation.

Another technique used in Kubernetes is reactive scaling based on resource utilisation. The Cluster Autoscaler Wang et al. (2020) is an example of this approach, which automatically adjusts the size of the cluster to meet resource demands. While reactive scaling can alleviate some of the problems associated with dynamic allocation, it introduces some time lag as it waits for resource utilisation to exceed predefined thresholds. Because the scaling action is not initiated until the threshold is exceeded, this time lag may cause performance issues during sudden workload spikes. Furthermore, incorrect

scaling threshold configuration can result in overly aggressive or insufficient scaling, affecting resource efficiency.

More proactive scheduling mechanisms are required as the demand for more efficient resource management grows. Predictive scaling is one such approach, which uses historical workload patterns and predictive algorithms to forecast future resource demands. Kubernetes can proactively scale resources before actual spikes occur by analysing historical metrics Casalicchio and Perciballi (2017) and trends, reducing response time and improving resource efficiency.

Furthermore, machine learning and AI-based scheduling algorithms can improve Kubernetes resource management. These advanced algorithms can process massive amounts of data, identifying complex patterns and making more intelligent resource allocation and scaling decisions. This level of sophistication enables the system to anticipate and adapt to changes in workload, optimising resource utilisation and improving user experience.

2.3 Allocation of Pods using Kubernetes

Due to its critical role in efficient resource utilisation and container orchestration, pod allocation in Kubernetes has been the subject of significant research and development.

Resource-aware scheduling has received a lot of attention. Various algorithms, such as bin-packing and genetic algorithms, have been proposed in studies to optimise pod placement on nodes based on resource requirements. introduced a bin-packing algorithm that reduces resource waste by efficiently packing pods onto nodes, resulting in better resource utilisation. Similarly, Carrión (2022) proposed genetic algorithms that dynamically adjust pod assignments based on CPU and memory utilisation, achieving better load balancing and scalability.

Auto-scaling strategies are critical in Kubernetes for dealing with dynamic workloads. Buchaca et al. (2020) presented a predictive auto-scaling approach that forecasts future resource demands using machine learning. By accurately forecasting workload fluctuations, this strategy preemptively scales up or down the number of pods, improving performance and cost efficiency. In contrast, Imdoukh et al. (2020) propose a reactive auto-scaling policy that adjusts the pod count based on real-time resource usage, with the goal of maintaining low latency and high throughput during traffic spikes.

Load balancing algorithms have been investigated in order to efficiently distribute incoming traffic across pods. Sinha and Sinha (2020) proposed a weighted round-robin approach for effectively balancing traffic load that takes into account the load on each node as well as the affinity of pods. In addition, Proposed a distributed load balancing algorithm based on the smallest-variance-first principle, which effectively reduces response time while improving overall system performance.

Pod affinity and anti-affinity have also been investigated in order to improve application performance and resilience. Affinity-based pod allocation strategy that places pods close together to reduce communication latency and improve data locality. On the other hand, investigate anti-affinity rules to ensure high availability and fault tolerance by avoiding co-location of critical pods on the same node.

While significant progress has been made in pod allocation techniques, challenges in dealing with resource heterogeneity, scalability, and fault tolerance remain. On my research investigate more advanced machine learning-based approaches and adapt to emerging containerization trends to further optimise pod allocation in this dynamic and distributed environment as Kubernetes adoption grows.

2.4 Algorithm Comparison

Reference	Algorithm	Approach	Advantages	Limitations
Kuleshov and Precup (2014)	Epsilon-greedy	Explore with probability ε Exploit the best-performing arm with probability $1-\varepsilon$	Simplicity, Tunability	May suffer from sub-optimal performance if ε is set too high or too low. Can take time to converge to the best arm(s).
Kaufmann et al. (2012)	Upper Confidence Bound (UCB)	Calculate upper confidence bounds for each arm	Effective Exploration, No need for ε	Can handle stochastic rewards efficiently. Provides good regret bounds in theory. Can be sensitive to initial conditions and variance of reward distributions. May require more computation than epsilon-greedy.
Agrawal and Goyal (2012)	Thompson Sampling	Maintain probability distributions for each arm's reward	Probabilistic Approach, Bayesian Framework	Naturally handles uncertainty in rewards. Provides good theoretical guarantees in some scenarios. Sampling can be computationally expensive, especially for complex reward distributions. Requires prior knowledge for Bayesian updating.

Table 1: Multi-armed Bandit Algorithms

3 Methodology

Proactive resource scheduling in Kubernetes refers to the process of intelligently allocating resources (such as CPU and memory) to different workloads (containers or pods) running on a cluster. Traditional reactive scheduling assigns resources based on current demand, which can result in resource underutilization or overutilization. Proactive scheduling seeks to optimise resource allocation decisions in advance by taking workload characteristics, historical usage patterns, and performance objectives into account.

Making decisions about which nodes in the cluster each workload should be placed on and how much resources should be allocated to them is the problem. The challenge is to find an appropriate balance between maximising resource utilisation, minimising response

times, and ensuring workload fairness. This is especially important in scenarios involving varying workloads and dynamic resource demands, where traditional approaches may result in inefficient resource allocation.

3.1 Multi-armed Bandit Problem to Resource Allocation:

The Multi-armed Bandit problem provides a useful structure for addressing Kubernetes' proactive resource scheduling challenge. The Multi-armed Bandit problem is analogous to the resource allocation scenario in that you have multiple "arms" (cluster nodes) to choose from, and each arm provides a different reward (resource allocation) based on an unknown probability distribution (workload performance).

The goal of the Multi-armed Bandit problem is to determine the best strategy for allocating pulls (choosing nodes) in order to maximise the cumulative reward (overall cluster performance) over time. To achieve the best possible outcome, the dilemma is to balance exploration (trying out different arms to gather information about their rewards) and exploitation (choosing the best-performing arms based on current knowledge).

In this study, the scheduler extender uses the results of the multi-armed bandit algorithm to decide where to put pods on nodes. By doing this, it anticipates to improve the overall use of resources, reduce competition for resources, and boost the performance of the cluster. The Kubernetes Scheduler Extender's job is to manage and scale the pods based on what amount of work is being done in the containers. The pods will be set up based on what the server wants. The Kubernetes controller uses the tools that come with it. The following diagram shows how the Kubernetes system is put together. Below Figure 1 represent the standard parts of the cluster of Kubernetes.

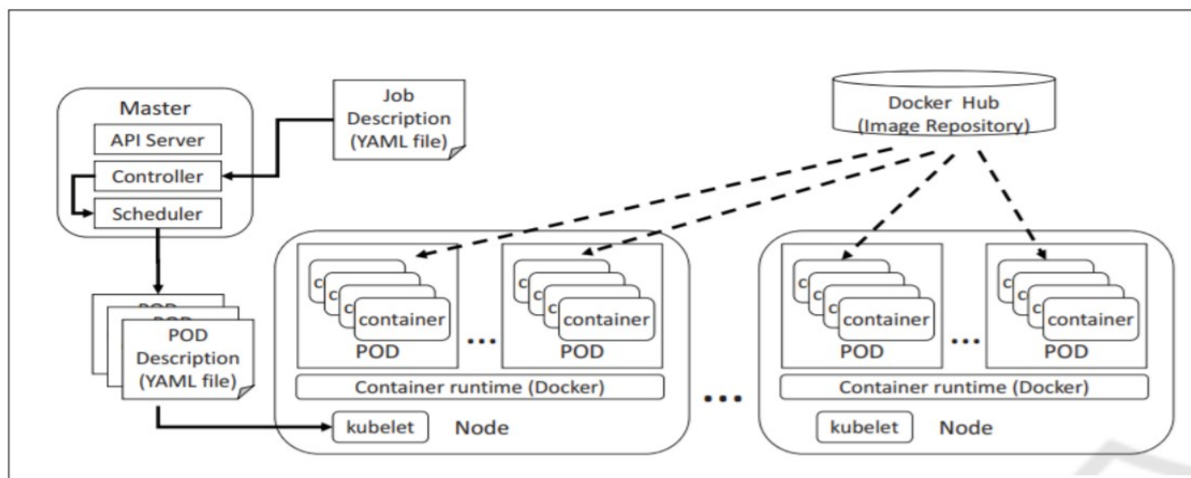

Figure 1: Kubernetes Cluster Component (Lin et al. (2019))

3.2 Research Methodology Flowchart

In this research, I am utilising Amazon Web Services, which offers a variety of services including EKS and EC2 instances. For all operations performed in this study, a t3.2xmedium instance was utilised. The flask framework and "numpy" library is utilised to implement the Thompson sampling algorithm in python. This algorithm, which I

implemented which is deployed as a docker image in a docker container using a docker commands. The Kubernetes cluster was created with "kubectl", and the service was containerized by unitizing the dockers. The services are deployed on the cluster, and custom controller is then used to generate the required number of pod. Kubernetes has tasked the load generator with generating the application's load.

And for gathering the metrics for the application, I'm utilising Prometheus, which collects the real-time data for this exception of the application and sends it to grafana, which displays the graphical representation of this metrics comparison using various metrics queries to identify the memory usage and CPU usage of the standard Kubernetes scheduling algorithm and the Thompson sampling algorithm that I show in this study.

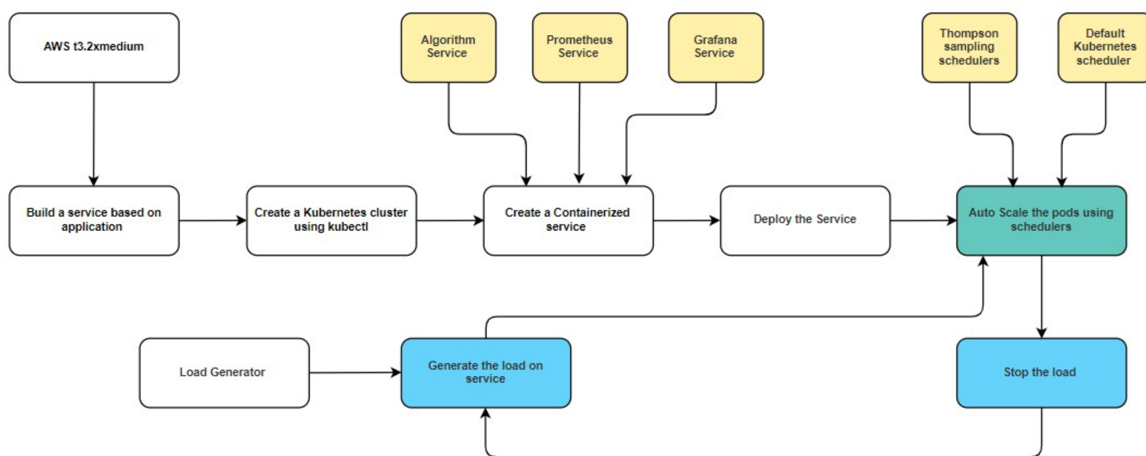

Figure 2: Process flow of reasearch

3.3 Technologies and Tools for Research

In this research was carried out on instance **EC2 t3.medium (Ubuntu)**. Kubernetes default scheduler is used for resource allocation. The algorithm in **PYTHON** programming language to implement and perform the computationally critical operations.

To conduct this research, the following tools and services were utilised:

- **Elastic compute service (EC2)** : It is a web service that provides resizable compute capacity in the cloud. This is service which is provide by Amazon Web Services. It serves as a foundation for setting up and running the Kubernetes cluster on which your research experiments are conducted.
- **Amazon EKS (Elastic Kubernetes Service)** : Amazon EKS is a fully managed Kubernetes service that makes it easier to deploy, manage, and scale containerized applications using Kubernetes. EKS abstracts the complexities of Kubernetes cluster management, enabling you to focus on your workloads and applications. Its leverage EKS to create and manage your Kubernetes clusters, which are essential for implementing and testing your proactive resource scheduling solution.

- **kubectl** : It allows to deploy applications, inspect cluster resources, and manage various aspects of your Kubernetes environment. will use kubectl to deploy your proactive resource scheduling components and manage the Kubernetes cluster during the research.
- **Docker** : Is a platform for developing, shipping, and running applications in containers. Containers provide an isolated environment for applications and their dependencies, ensuring consistency across different environments. In this research, containerize workloads (pods) to represent different arms in the Multi-armed Bandit problem, allowing you to simulate and evaluate the resource allocation strategy.
- **Prometheus** : It collects and stores time-series data, enabling monitoring of various Kubernetes cluster and application aspects. This study collects metrics associated with resource utilisation and workload performance, such as CPU utilisation and memory usage.
- **Grafana** : It uses for to visualise the Prometheus-collected metrics, facilitating the analysis of proactive resource scheduling strategy.

3.4 Analysis performed In this Research

In this study, total three experiments are taken consideration to ensure consistency in the evaluation of the Thompson sampling algorithm. After generating the workload, time frames of five minutes are used to evaluate the three experiments.

It will compare the algorithms Thompson sampling algorithm and kubernetes default resource scheduler in order to validate the CPU usage and memory utilization of resources. Observations are shown based on current and projected usage.

4 Design Specification

The design requirements for the project are broken down and described in this part. While section 4.1 outlines the necessary configuration of the system for the execution of this research, section 4.2 describes the architecture of the system.

4.1 Required System Specifications

In this study, the deployment and operation of containers are managed by the orchestration Elastic Kubernetes Services. Kubernetes is used to manage the containers inside the cluster. Docker container is used to combine application code with any given environment. The Kubernetes Cluster and Virtual Machine Configurations are shown in tables 2 and 3.

AWS Kubernetes Cluster		
Instance	AWS	t3.medium
Operating System	Ubuntu	Version = 20.06
Orchestration tool	Kubernetes	Version = 1.19
Container engine	Docker	Version = 18.01

Table 2: Cluster Configuration

Virtual Machine (VPC)	
vCPU	8
Memory	32GiB
Network Performance	upto 5 gbps
Cost	$0.3341/ hr

Table 3: Virtual machine Configuration

4.2 System Architecture

To use the Kubernetes cluster, the kubectl package must be installed. In this study, version 1.19 of kubeadm is utilised. To execute the command in cluster, version 1.21 of kubectl is installed. In the kubernetes cluster, there is one master node and two worker nodes. Containers are utilised by the worker node in order to organise pods. etcd manages clusters between worker and master nodes.

The Thomspon sampling Controller, Application programming interface (API) server, and the Scheduler are the three components that are located within the master node. The controller manager is responsible for managing the fundamental Kubernetes functions. The controller is responsible for exercising control over the pods managed by Kubernetes and bringing them through a shared state to the state that is desired. In the event that adjustments are required, the Kubernetes controller will submit the appropriate requests. Pods undergo configuration and verification with the application programming interface (API) server. The Kubernetes scheduler is where the policies are stored.

Docker, the proxy server system, and Kublet are the components that make up a worker node. Communication protocol kublet is used between master and worker nodes. To manage the state of the master node, Kublet is responsible for the administration of pods that comply with the pod specifications by the master node in the PodSpec. Representation of architecture is shown in Figure 8 as fallows.

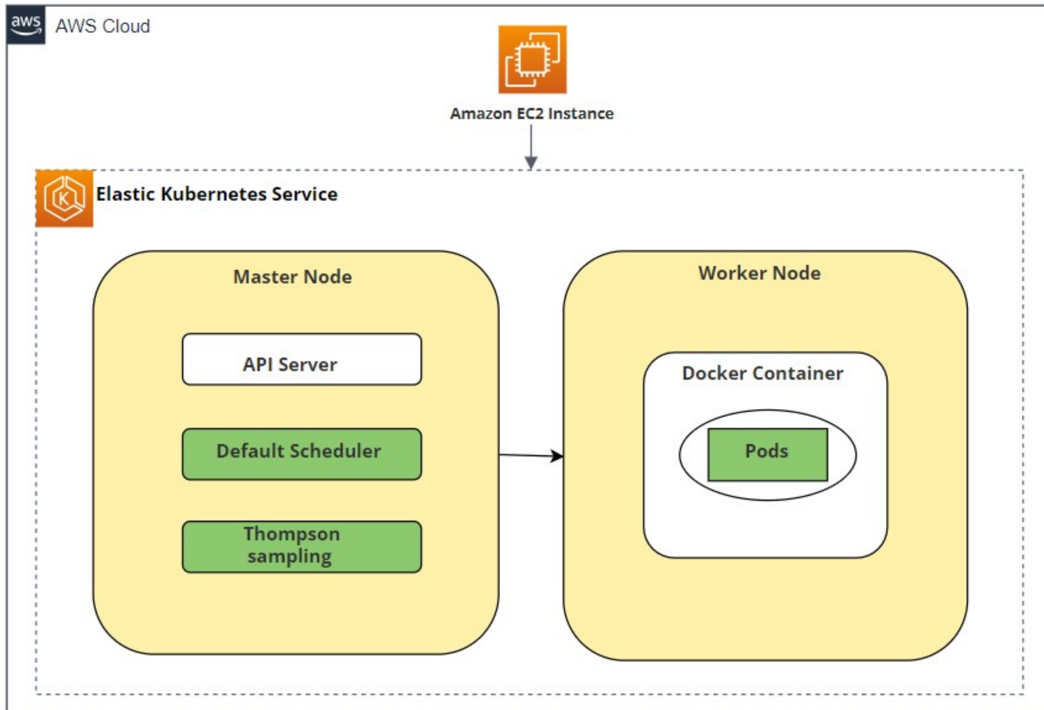

Figure 3: Representation of Architecture

5 Implementation

5.1 Implementation of K8s Algorithm

5.1.1 Process of Scheduling

The primary responsibility in the K8s Scheduler was to acknowledge the creation of a new Pod by the Application Programming Interface (API) Server, locate a suitable host for the Pod, and record the relevant information in Etcd. Figure 4 depicts the steps involved in the scheduling process for one of the Pods:

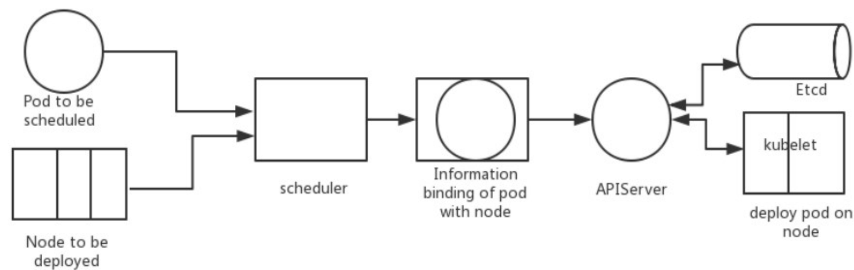

Figure 4: Flow Chart of Scheduling

Algorithm 1 Pod Scheduling Algorithm

Require: List of Pods to be scheduled
Ensure: Mapping of Pods to Nodes

```
1: for all Pods in List of Pods do
2:     candidateNodes ← allNodes
3:     candidateNodes ← filterNodesByResources(Pod, candidateNodes)
4:     scoredNodes ← scoreNodesByResources(Pod, candidateNodes)
5:     prioritizedNodes ← prioritizeNodes(scoredNodes)
6:     bestNode ← selectBestNode(prioritizedNodes)
7:     bindPodToNode(Pod, bestNode)
8: end for
9: return Mapping of Pods to Nodes
```

- ***Input and Output:*** The algorithm takes a list of pods that need to be scheduled as input and aims to produce a mapping of which pod should be placed on which node as output. This mapping is the result of the scheduling process.

- ***Initialization and Candidate Nodes:*** For each pod in the list of pods, the algorithm begins by initializing a set of candidate nodes as all available nodes in the cluster. This is the starting point for considering potential placement options for the pod.

- ***Filtering Based on Resource Constraints:*** The algorithm then applies a filtering process to narrow down the list of candidate nodes based on the resource requirements of the pod. Nodes that don't have sufficient resources (such as CPU and memory) to accommodate the pod's needs are filtered out from consideration.

- ***Scoring Nodes for Resource Availability:*** After filtering, the remaining candidate nodes are scored based on their resource availability. This score reflects how well each node can fulfill the resource requirements of the pod. Nodes with more available resources tend to receive higher scores.

- ***Prioritization of Nodes***: The scored nodes are further prioritized based on their scores. Nodes with higher scores, indicating better resource availability and suitability, are given higher priority. This step helps to determine the order in which nodes will be considered for pod placement.

- ***Selecting the Best Node:*** Among the prioritized nodes, the algorithm selects the node with the highest score. This node is considered the best candidate for placing the pod. The high score indicates that the selected node can provide the necessary resources for the pod's successful execution.

- ***Assigning Pod to Node:*** The algorithm then assigns the selected pod to the best node. This assignment is recorded, indicating that the chosen node will host the pod during execution.

- ***Mapping and Output:*** After processing all pods in the list, the algorithm produces a mapping that associates each pod with the node it has been assigned to. This mapping represents the final outcome of the scheduling process.

5.2 Implementation of Thompson sampling Algorithm

Algorithm 2 Thompson Sampling Algorithm

Require: Number of arms (bandit options)
Ensure: Selected arm

1: Initialize arrays for counts and rewards for each arm
2: **for all** time step t **do**
3:     **for all** arm i **do**
4:         Sample a value from the Beta distribution using arm's counts
5:     **end for**
6:     Select the arm with the highest sampled value
7:     Observe reward from the selected arm
8:     Update arm's counts and rewards based on the observed reward
9: **end for**
10: **return** the arm that was selected most often

The Thompson Sampling algorithm involves using probability distributions to make decisions. In this case, the Beta distribution is commonly used due to its ability to model uncertainty about success probabilities. Let's represent the algorithm mathematically:

Notation:

$$
\begin{aligned}
N &: \text{Total number of arms (bandit options)} \\
t &: \text{Time step} \\
T_i(t) &: \text{Number of trials (attempts) for arm } i \text{ up to time step } t \\
R_i(t) &: \text{Number of rewards (successes) for arm } i \text{ up to time step } t \\
\theta_i &: \text{True probability of success for arm } i \\
X_i(t) &: \text{Random variable representing reward outcome for arm } i \text{ at time step } t
\end{aligned}
$$

Algorithm:

1. **Initialization**: - Initialize $T_i(0) = R_i(0) = 0$ for all arms i.

2. **Sampling Phase**: - At each time step t, sample a value $p_i(t)$ from the Beta distribution $\text{Beta}(R_i(t)+1, T_i(t)-R_i(t)+1)$ for each arm i. - $p_i(t)$ represents the estimate of the success probability for arm i at time step t.

3. **Arm Selection**: - Select the arm a_t with the highest sampled value: $a_t = \arg\max_i p_i(t)$. - a_t is the arm chosen at time step t for exploration or exploitation.

4. **Observation and Update**: - Observe the reward $X_{a_t}(t)$ from the selected arm a_t at time step t. - Update the counts and rewards for the selected arm: - $T_{a_t}(t+1) = T_{a_t}(t) + 1$ - $R_{a_t}(t+1) = R_{a_t}(t) + X_{a_t}(t)$

5. **Repeat**: - Repeat steps 2-4 for a specified number of time steps.

6. **Recommendation**: - At the end of the process, select the arm i that has been chosen the most often: $i^* = \arg\max_i \sum_t \nVdash(a_t = i)$. - i^* is the recommended arm, which is the algorithm's decision based on the observed rewards.

Example Scenario: Consider a scenario with 3 arms (options). Let's assume the true success probabilities for these arms are $\theta_1 = 0.3$, $\theta_2 = 0.6$, and $\theta_3 = 0.8$. The algorithm aims to maximize the cumulative rewards over a number of time steps.

The algorithm seeks to investigate arms with uncertain reward probabilities while exploiting arms with higher success probabilities. Thompson Sampling provides a probabilistic approach to multi-armed bandit problems by modelling uncertainty with the Beta distribution and constantly updating estimates based on observed rewards.

6 Evaluation

In this part of the paper, the performance evaluation is carried out by carrying out a variety of experiments on proposed system and compare the results with the Kubernetes use by default. The proposed system makes use of the Elastic Kubernetes service in order to execute a Kubernetes cluster and to host the application for testing and evaluation purposes. The Default Kubernetes scheduler algorithm , which was used and developed by the same system specifications as the proposed system, has been used for comparison purposes. The experiments were carried out by producing load for a period of five minutes in each of the three distinct conditions (as outlined in table 4 below). The users are simulate and the number of receiving requests per second on the application by using the testing tool which is used to generate a load on application called locust. Prometheus and Grafana were utilised in order to monitor and record various metrics, including CPU utilisation, memory utilisation, and response time.

Experiment	**Total User**	**Per second request**
1	1000	100
2	3000	300
3	5000	500

Table 4: Simulations of experimentation

6.1 Experiment 1

For the first scenario, we have determined the total no. of users and the number of requests that are made per second based on scenario 1, as shown in Table 4. It is possible to see from the figure below that the 95th percentile of the overall response period (Maximum time to respond for 95% requests) and average response time are showing in *"green line"* for both of the systems (proposed default) are very close to one another. In contrast, the cluster that uses the standard Kubernetes system uses significantly more memory and CPU resources than the proposed system does (30% and 10.9% respectively).

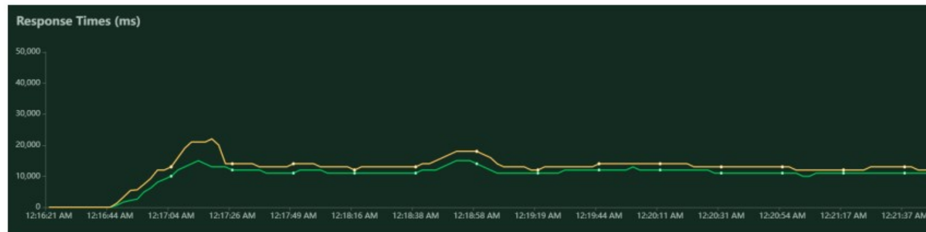

Figure 5: Default kubernetes System

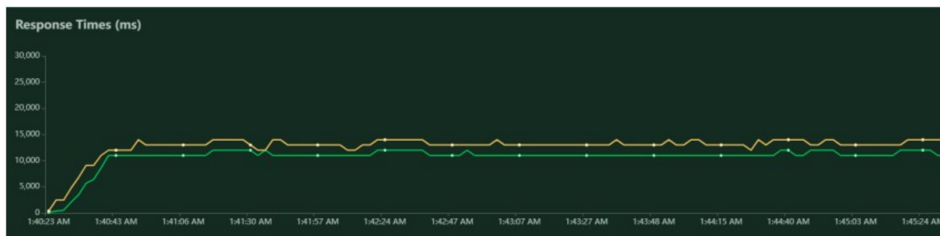

Figure 6: Proposed system

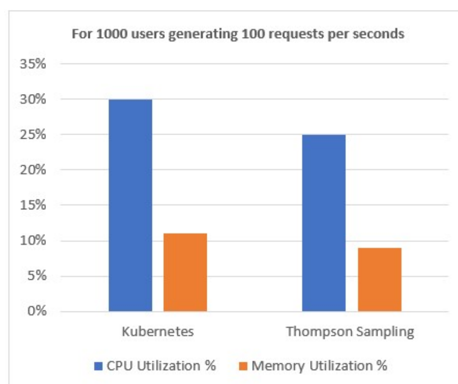

Figure 7: Resources consumption comparison

6.2 Experiment 2

The configuration that was stated in scenario 2 was applied to the second experiment, and the results are presented in Table 4. It is possible to notice, based on the information presented in Figure, that the time taken to respond for the first couple of seconds for both of the system is zero. This may suggest that web application was initially inactive when it was first launched. The response time for proposed model is 24464 milliseconds lower compared to that corresponding to default Kubernetes simulation is 34925 milliseconds after one minute of receiving load. As can be seen in the figure, the amount of time spent using the CPU and the amount of memory consumed by the default system is consistently higher than that of the proposed system.

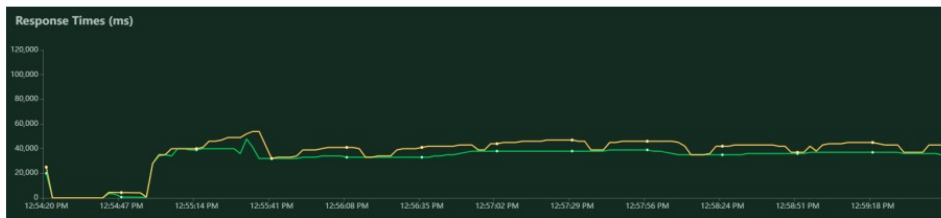

Figure 8: Default Systems

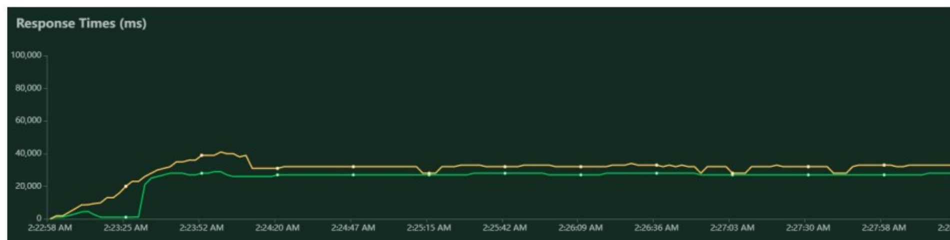

Figure 9: Proposed system

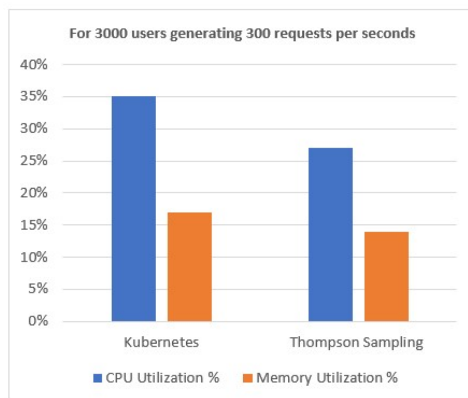

Figure 10: Resources consumption comparisons

6.3 Experiment 3

In the third scenario, we chose parameters based on scenario 3 of Table 4, which can be found here. It is clear to the beginning that an application that is hosted in default Kubernetes system struggles to achieve satisfactory responses when five thousand users with five hundred request per seconds are deployed. On the other hand, suggested setup that is based on Thompson sampling maintains stable. Additionally, CPU utilisation in standard Kubernetes cluster strikes above 40 percent and memory nearly twenty five percent, whereas CPU utilisation and consumption of memory in Thompson sampling base cluster stays on to remain less compared to those of the default model.

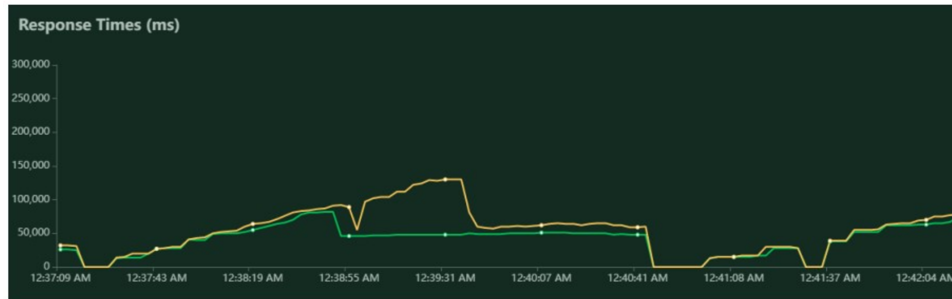

Figure 11: Default System

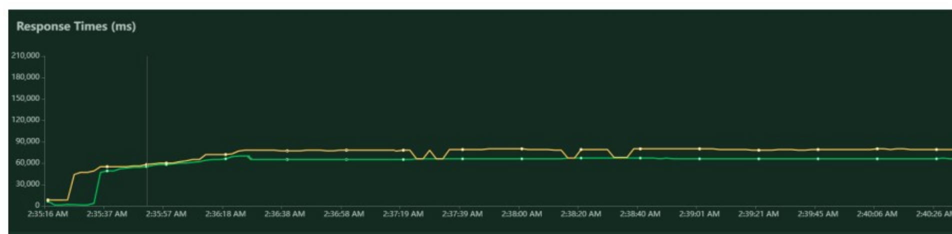

Figure 12: Proposed system

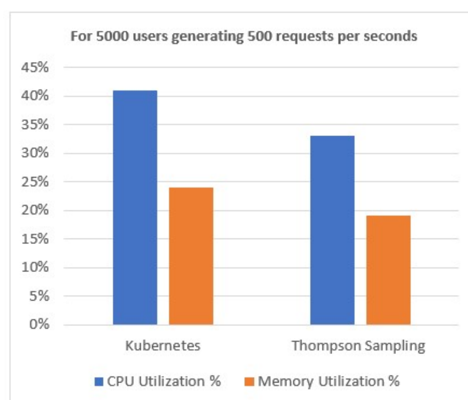

Figure 13: Resources consumption comparisons

6.4 Learning Outcomes

As illustrated in Figure 14, the CPU utilisation of the Default Kubernetes Scheduler is increasing in comparison to the proposed system from the various scenarios we discussed. 6.1 6.2 6.3

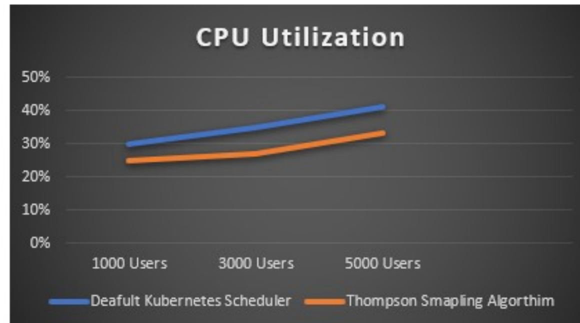

Figure 14: CPU Utilization

As shown in Figure 15, the Memory Usage of the Default Kubernetes Scheduler consumes more memory while the proposed system consumes less memory.

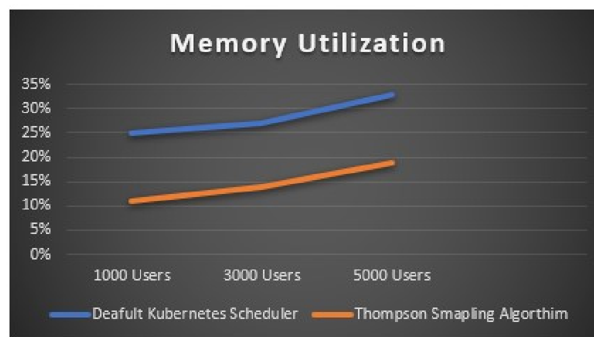

Figure 15: Memory Utilization

6.5 Discussion

Based on the experiments that were performed on the proposed system that is based on Thompson sampling algorithm and the default system that is based on kubernetes, it is possible to say that the time it takes to respond is almost equal for both approaches when there is less no. of users and less receiving request per second in a application (scenario 1). This was determined by comparing the results of the experiments. If we compare the proposed design to the default design, we can see that the proposed design makes less use of the available resources. However, when the total no of users and demands rises (scenario 2 and scenario 3), default Kubernetes system begins is getting difficulty to handle the resources. This results in failed requests, longer response times and increased memory and CPU consumption. In contrast, the Thompson sampling algorithm system that was proposed outperforms the default one by maintain consistency and reliability in

response rate while also consuming less resources than kubernetes default scheduler one does.

When compared to other models, the proposed design requires a lower amount of resources; consequently, any additional costs associated with buying or providing additional resources can be avoided.

7 Conclusion and Future Work

In this study, we set out to revolutionise Kubernetes resource management by employing a proactive approach based on the Multi-armed Bandit Algorithm, specifically the Thompson Sampling variant. We demonstrated the efficacy of this algorithm in dynamically allocating resources to application workloads through extensive experimentation and analysis, resulting in optimised CPU and memory utilisation. We demonstrated the Thompson Sampling-based proactive scheduler's potential to improve overall cluster efficiency and application performance by comparing it to the default Kubernetes scheduler. This study not only emphasised the benefits of intelligent resource allocation, but also the importance of adaptive scheduling strategies in today's dynamic containerized environments. The findings of this study pave the way for more sophisticated resource management mechanisms that can address the intricate demands of modern applications as container orchestration evolves.

While this study successfully establishes the benefits of proactive resource scheduling using the Thompson Sampling-based Multi-armed Bandit Algorithm, several avenues for future research are identified. To begin, broadening the scope to include other performance metrics such as network latency and storage usage could provide a more complete picture of the algorithm's impact. Furthermore, researching hybrid scheduling approaches that combine the strengths of both traditional and proactive algorithms may provide a more complex solution. Furthermore, the research could be expanded to deal with more complex scenarios like multi-cluster orchestration or heterogeneous environments. Furthermore, the deployment of real-world applications and evaluation of the algorithm's behaviour under varying workload conditions may provide insights into its practical applicability. Finally, advances in machine learning and Kubernetes may lead to the development of more sophisticated algorithms, necessitating ongoing research in this domain to ensure the efficient management of containerized applications in dynamic computing landscapes.

References

Agrawal, S. and Goyal, N. (2012). Analysis of thompson sampling for the multi-armed bandit problem, *Conference on learning theory*, JMLR Workshop and Conference Proceedings, pp. 39–1.

Arora, S. and Ksentini, A. (2021). Dynamic resource allocation and placement of cloud native network services, *ICC 2021 - IEEE International Conference on Communications*, pp. 1–6.

Buchaca, D., Berral, J. L., Wang, C. and Youssef, A. (2020). Proactive container autoscaling for cloud native machine learning services, *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 475–479.

Carrión, C. (2022). Kubernetes scheduling: Taxonomy, ongoing issues and challenges, *ACM Computing Surveys* **55**(7): 1–37.

Casalicchio, E. and Perciballi, V. (2017). Auto-scaling of containers: The impact of relative and absolute metrics, *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 207–214.

Chang, C.-C., Yang, S.-R., Yeh, E.-H., Lin, P. and Jeng, J.-Y. (2017). A kubernetes-based monitoring platform for dynamic cloud resource provisioning, *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6.

Imdoukh, M., Ahmad, I. and Alfailakawi, M. G. (2020). Machine learning-based autoscaling for containerized applications, *Neural Computing and Applications* **32**: 9745–9760.

Jawarneh, I. M. A., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R. and Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison, *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pp. 1–6.

Kaufmann, E., Cappé, O. and Garivier, A. (2012). On bayesian upper confidence bounds for bandit problems, *Artificial intelligence and statistics*, PMLR, pp. 592–600.

Khan, A. (2017). Key characteristics of a container orchestration platform to enable a modern application, *IEEE Cloud Computing* **4**(5): 42–48.

Kuleshov, V. and Precup, D. (2014). Algorithms for multi-armed bandit problems, *arXiv preprint arXiv:1402.6028* .

Lin, C.-Y., Yeh, T.-A. and Chou, J. (2019). Dragon: A dynamic scheduling and scaling controller for managing distributed deep learning jobs in kubernetes cluster., *CLOSER*, pp. 569–577.

Sinha, G. and Sinha, D. (2020). Enhanced weighted round robin algorithm to balance the load for effective utilization of resource in cloud environment, *EAI Endorsed Transactions on Cloud Systems* **6**(18).

Wang, M., Zhang, D. and Wu, B. (2020). A cluster autoscaler based on multiple node types in kubernetes, *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Vol. 1, pp. 575–579.