National College of Ireland

Custom Kubernetes Scheduler-based on priority scheduling for Serverless Framework

MSc Research Project
Cloud Computing

Manali Yadav
Student ID: 21225451

School of Computing
National College of Ireland

Supervisor: Rejwanul Haque

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Manali Yadav
Student ID:	21225451
Programme:	Cloud Computing
Year:	2023-2024
Module:	MSc Research Project
Supervisor:	Rejwanul Haque
Submission Due Date:	31/01/2024
Project Title:	Custom Kubernetes Scheduler-based on priority scheduling for Serverless Framework
Word Count:	6948
Page Count:	18

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:	M. Yadav
Date:	30th January 2024

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):	

Custom Kubernetes Scheduler-based on priority scheduling for Serverless Framework

Manali Yadav
21225451

Abstract

At times, Kubernetes has gained popularity as an option for managing containerised workloads at scale. When it comes to assigning workloads to the nodes Kubernetes relies on a scheduler that takes into account a range of constraints set by workload owners and cluster administrators. While the default Kubernetes scheduler is highly configurable, it may not fully meet the requirements of serverless applications. This is because it operates in a pod scheduling mode, whereas certain serverless frameworks require co-scheduled alongside pod priority. In this research, Open Whisk is an open-source serverless framework specifically designed for a Kubernetes cluster consisting of workers deployed on the Amazon Web Service (AWS) cloud at National College of Ireland. A noteworthy contribution of this study is the development of a custom scheduler for Kubernetes aimed at enhancing pod allocation to worker nodes through an innovative scoring algorithm tailored for pods. With the proposed scheduling algorithm, the Central Processing Unit (CPU) utilisation was reduced by approx. more than 50%, and the average pod scheduling time for the custom scheduler is 0.2 sec which shows a better difference as compared to the default Kubernetes scheduler. Better performance is seen in other factors as well, like memory utilisation and throughput.

Keywords- Kubernetes, scheduling, serverless, OpenWhisk, prioritisation.

1 Introduction

Serverless cloud computing in today's time is completely a new architectural methodology and a ground-breaking architectural paradigm. While serverless services still require servers, customers don't have to worry about the amount, status, or resources of the underlying servers used by software applications. The underlying cloud computing infrastructure dynamically provides the computational resources needed for software programs to function normally. Cloud computing platforms dynamically increase the number of application deployment instances in response to traffic surges. The cloud computing platform removes the program from the host and recycles resources when it is not in use for a while. There are two basic groups into which the serverless solutions available today fall. In the first type of serverless cloud service providers will offer their own paid Serverless platforms for users to utilise. AWS Lambda, Google Cloud Functions, IBM Cloud Functions, and Microsoft Azure Functions are a few examples. Platform locking is one of the inherent drawbacks of this kind of platform. It is only possible to design and operate in accordance with the platform's requirements; otherwise, scalability and personalisation

will be subpar. As per Fan and He (2020) The open-source community solutions fall into the second group; OpenWhisk, Fission, Kubeless, Knative, and Openfaas are some of the most notable examples.

The majority of studies have assessed compute-intensive applications on for-profit serverless platforms-like Azure Functions, Google Cloud Functions (GCF), and AWS Lambda. These platforms include all the features developers need to create faster-to-market solutions, but they also have a number of disadvantages, including complicated pricing structures, vendor lock-ins, limitations on security, and the lack of language runtimes. However, instead of introducing vendor lock-ins, open-source Function as a Service (FaaS) frameworks allow programmers to write applications in a range of programming languagesMohanty et al. (2018). Furthermore, developers don't have to worry about the following limitations as execution time is required by the default Kubernetes scheduler to schedule the pods This study primarily focuses on one such open-source FaaS, called OpenWhisk. In addition, the pay-as-you-go approach does not offer major FaaS infrastructure providers adequate control over invoicing, indicating the need for open-source FaaS in the private cloud to enable greater control over serverless costs.

Co-scheduling indicates that a pod group has to be scheduled concurrently, as opposed to the standard Kubernetes method, which schedules pods one at a time. For such workloads, the Kubernetes pod-by-pod scheduling approach is inefficient since it necessitates repeatedly traversing the list of nodes for each pod scheduling, even in cases where the schedulable pods share identical features. In this research the main focus has been on the prioritisation of pods for scheduling. To date, much research has been done on the development of better and more efficient schedulers for Kubernetes. The Kubernetes default scheduler fails to meet the scheduling requirements for serverless frameworks. Through this research, a more efficient Kubernetes scheduling algorithm is developed specifically for the serverless framework. This algorithm mainly focuses on the prioritisation of the pods that are queued before scheduling. At the core of this thesis lies a pivotal research question.

1.1 Research Question

How efficient is a custom Kubernetes scheduler designed with a specific pod scoring algorithm as compared to the default Kubernetes scheduler for serverless architecture conveyed on AWS?

This question tests the viability of a custom scheduler against the scene of AWS's flexible cloud framework. It researches possible enhancements in asset use, reaction times, and general framework execution while sending serverless functions utilising Apache OpenWhisk. Moreover, it considers the extensibility of Kubernetes planning abilities and how they can be customised to oblige different functional requests, consequently improving the coordination of containerised responsibilities. The resulting research is meant to take parts of the custom scheduler, examine its algorithmic productivity, and assess its presentation in a controlled cloud climate. By establishing a comparative framework, the research attempts a comparison between generalised and specialized scheduling mechanisms within cloud-based Kubernetes applications for a serverless function, OpenWhisk. As a result, this work closes the gap in the literature and benchmarks for open-source serverless computing. The open-source distributed serverless platform Apache OpenWhisk has not been extensively investigated in customised the private cloud's Kubernetes

container orchestrator for concurrent workloads.

1.2 Objective

The primary objective of this research is to develop a customised Kubernetes scheduler, specifically designed around the idea of pod scoring.The suggested custom scheduler schedules the pods according to the priority set for the pods. This showcases the lesser use of CPU and memory.

1.3 Structure of the report

This section covers the report structure, whereas the first section is the brief literature review. The next part of this section, Research Niche, is covered. The second section is the methodology section, which gives a brief overview of the methods used to do the research. The third section contains the design specification, where the tools and technologies used to build the research are highlighted. The fourth section is Implementation, where tools, languages, and custom scheduler algorithms are explained. The fifth is the evaluation section, where the results and graphs are discussed. The sixth and last section is the conclusion and future scope.

2 Related Work

2.1 Deployment Strategies in Cloud Computing

Deployment methodologies in cloud computing have developed fundamentally, progressing from customary models to current, more proficient methodologies-like serverless structures. This advancement is critical for understanding the effect and capability of cloud computing in different applications. Mahajan and Kunal (2021), in his work, accentuates the significance of enhancing container deployments to effectively use organisation, process, and storage assets in cloud conditions. The author discusses the necessity for appealing assessing strategies for figure organisations, including containers, virtual machines, and serverless computing. This investigation includes the twofold trial of achieving prevalent execution in container deployments and perceiving reasonable assessing for figure organisations, which are fundamental pieces of present-day cloud deployment strategies.

Concerning Business Process Management Systems (BPMSs), Ouyang and et al. (2021) examine the shift from standard on-premises models to Software as a Service (SaaS) perspective, significance to convey Business Process Computerisation as an Assistance. Their audit features the moves in expanding ordinary BPMS to fulfill simultaneous requirements from different relationship in the cloud. This shift to SaaS highlights the influential thought of cloud deployment and the prerequisite for flexible courses of action that can acclimatise to evolving demands.

The possibility of vehicular micro-clouds (VMCs) introduced by Phadke et al. (2021) presents a creative increase to standard cloud computing systems. VMCs impact the computing power and correspondence equipment in present day vehicles to convey flexible cloud game plans, offering a clever perspective on deployment systems that address issues-like genuine distance and questionable correspondence establishment.

Eventually, the paper analyses the improvement of serverless computing past standard clouds to the Edge Nastic and et al. (2022). This study proposes the possibility of a Serverless Computing Fabric (SCF) for the Edge-Cloud continuum, outlining a reference designing that opens the greatest limit of this continuum. The exploration features the regular developmental step for serverless computing, stressing its low management above, simple deployment and improved costs.

2.2 Serverless Computing Paradigms and AWS Implementations

Serverless computing addresses a critical change in cloud deployment procedures, offering a more effective and versatile way to deal with overseeing cloud assets. This segment investigates the points of interest of serverless computing, with an emphasis on executions by Amazon Web Services (AWS), especially AWS Lambda and EC2. Bits of knowledge from four academic articles give a more profound comprehension of how serverless computing is carried out in true situations.

Gandhi and et al. (2018) present a pragmatic use of serverless architecture involving AWS Lambda in their work. They dissect the architecture and execution of a common drive web application, zeroing in on reaction times for cold and warm demands, load adjusting, execution on memory reservation, and asset maintenance conduct. This study gives significant experiences into the functional parts of serverless computing in a genuine application.

Andi (2021) offers a more extensive outline of the serverless cloud computing model. It talks about the shift from customary models, where engineers are answerable for asset portion and server management, to serverless computing, where these obligations are taken care of by the cloud specialist co-op. This paper features the advantages of serverless computing, for example, decreased opportunity to market and cost viability, while likewise examining its constraints.

In Choudhary and et al. (2020) investigate the utilisation of AWS Lambda in fostering a serverless talk application. This contextual investigation exhibits how serverless advancements-like Lambda can uphold adaptability without the expansion of new servers, and how it incorporates with other AWS services-like S3, DynamoDB, and CloudWatch.

Rajan (2020) gives an exhaustive report on serverless computing architecture. It incorporates a trial and error of the functioning standard of serverless computing reference model adjusted by AWS Lambda, recognising different exploration roads in serverless computing.

These literature works aggregately highlight the extraordinary effect of serverless computing in cloud deployment. They feature AWS's job in progressing serverless computing, exhibiting reasonable applications and hypothetical underpinnings of this worldview. The emphasis on AWS Lambda and EC2 gives useful experiences into how serverless computing is executed, offering a brief look into the eventual fate of cloud computing.

2.3 Kubernetes in Cloud Environments

Kubernetes has arisen as a vital innovation in cloud computing, especially for overseeing containerised applications. Its job and highlights are instrumental in supporting server-

less architectures, overcoming any barrier between conventional container management and the advancing requirements of serverless computing. This part draws upon experiences from four insightful articles to investigate how Kubernetes adjusts to and improves serverless computing conditions.

Böhm and Wirtz (2022) analyses Kubernetes with regards to edge computing, especially for savvy city applications. They examine custom cloud-edge architectures executed with Kubernetes, featuring its abilities and weaknesses in edge coordination. The review uncovers that while Kubernetes successfully oversees containerised applications, it faces difficulties in regions-like ongoing asset use and adaptation to non-critical failure, urgent for latency-basic, huge scope deployments.

This paper Cepuc and et al. (2020) presents a robotized pipeline for conveying containerised applications in AWS, using Kubernetes. It exhibits Kubernetes' part in working with consistent joining and deployment, guaranteeing quick, solid, and adaptable application deployment in cloud conditions. The reconciliation of Kubernetes with different advances-like Jenkins and Ansible highlights its adaptability and significance in current cloud computing.

Fiori et al. (2022) presents RT-Kubernetes, a product architecture that conveys ongoing programming parts inside containers in cloud foundations. This transformation of Kubernetes shows its capacity to give practicality ensures areas of strength for and disengagement among containers, featuring Kubernetes' versatility to explicit necessities of constant cloud computing.

The review Nelson and et al. (2020) investigates the utilisation of Kubernetes for upgrading the situation of containerised applications in cloud and edge computing organisations. It stresses Kubernetes' job in dealing with the deployment of uses across various layers of cloud and edge architectures, in this way decreasing absolute runtime and improving framework productivity.

They delineate Kubernetes' adaptability and adequacy in cloud conditions. They feature their job in overseeing containerised applications, supporting serverless architectures, and adjusting to the remarkable requests of edge computing and ongoing cloud applications. Kubernetes works on deployment and management of utilisations as well as improves the adaptability, dependability, and effectiveness of cloud computing frameworks.

2.4 Function as a Service (FaaS) Platforms

FaaS is a center part of serverless computing, offering a stage for executing code considering occasions without dealing with the hidden foundation. This segment investigates the job and functionality of FaaS in cloud computing, drawing experiences from four academic articles.

Malla and Christensen (2019) looks at Google Cloud's FaaS (Cloud Functions) with its IaaS (Process Motor) regarding cost and execution for equal errands. They found that FaaS can be more practical than IaaS while offering comparable execution levels, although FaaS execution displays higher variety because of versatility reliant upon the cloud supplier.

Sewak and Singh (2018) talks about the developing foothold of Serverless computing and FaaS in cloud-based architectures. It features how serverless architectures permit designers to zero in solely on business rationale without agonising over framework concerns. The paper additionally investigates the benefits, constraints, and arising use instances of FaaS.

Shahrad et al. (2019) researches the engineering ramifications of FaaS utilising the Apache OpenWhisk FaaS stage. The review recognises difficulties, for example, containerisation stoppage, cold-start times, and between function obstruction, giving experiences into the framework-level overheads presented by FaaS. This multivocal writing audit Grogan and et al. (2020) gives an investigation of FaaS frameworks, zeroing in on adaptability, cost, execution times, and reconciliation support. The exploration talks about FaaS services from different merchants, including AWS Lambda, Google Cloud Functions, and azure Functions, and their suggestions for programming designers.
The examinations give an exhaustive comprehension of FaaS in cloud computing. The reconciliation of FaaS with stages-like Apache OpenWhisk and its correlation with conventional IaaS models offer important bits of knowledge into the developing scene of cloud computing and serverless architectures.

2.5 Custom Schedulers for Kubernetes and Resource Optimisation

Custom schedulers in Kubernetes assume a pivotal part in streamlining asset portion and management. This segment, informed by these academic articles, digs into the turn of events and effect of custom schedulers in Kubernetes, especially zeroing in on case scoring calculations and their correlation with the default Kubernetes scheduler.
Ning (2023) proposes a custom Kubernetes planning algorithm enhanced over the default algorithm. It resolves issues-like setting a fitting solicitation value and deciding the scoring rule. The custom algorithm shows further developed usage of hub assets and better burden balance between hubs in a cluster.
Fan and He (2020) centers around advancing pod planning for huge-scope simultaneous situations of Serverless systems in view of Kubernetes. They propose an algorithm utilising a similar pod synchronous booking to improve pod planning effectiveness in Serverless cloud ideal models, fundamentally decreasing pod startup delay while guaranteeing hub asset balance. This exploration Wei-guo et al. (2018) further develops Kubernetes' planning model by joining subterranean insect state and molecule swarm enhancement algorithms. The proposed algorithm chooses hubs with the littlest goal function for Pod deployment, lessening absolute asset cost and adjusting task tasks more successfully.
Beltre et al. (2021) presents a Policy-driven Multi-Tenant Kubernetes (PMK) structure. The PMK permits the re-sequencing of errands by means of notable or custom scheduling algorithms, giving bits of knowledge into factors influencing decency and working on normal holding times. This explores all the features and meanings of custom schedulers in Kubernetes for asset enhancement. They show the way that custom algorithms can further develop asset usage, load adjusting, and productivity in different situations, including serverless structures and multi-tenant conditions. The correlation of these custom schedulers with the default Kubernetes scheduler highlights Kubernetes' adaptability and versatility in asset management, exhibiting its capacity to proficiently meet different and dynamic responsibility necessities.In Table 1 a summary of the literature work is listed, along with the technologies used for their research and the limitations of the research work.

2.6 Research Niche

The second section of the literature study explores some of the implementation issues associated with cloud computing. It includes the development of Function as a Service (FaaS) platforms, custom Kubernetes calendars, serverless computing, AWS deployments, and deployment strategy evolution. The evaluation offers guidance on how to introduce vehicular micro-clouds (VMCs), optimise container deployments, and transition enterprise process management tools and systems to a SaaS paradigm. Further discussing the creation and effects of custom schedulers in Kubernetes for resource optimisation as well as the functionality and effectiveness of FaaS platforms. A complete assessment of current trends and improvements in the use of cloud computing deployment tactics is presented by this thorough study, which synthesises major results from scholarly literature.

3 Methodology

The methodology chapter meticulously documents the approach undertaken to address the research question, detailing the design and functioning of the custom scheduler, the establishment of baseline benchmarks, and the configuration of the execution environment.

Pod priority queues and class pods according to their relative importance to each other. Assume that scheduling is required for two distinct pod types. By default, the Kubernetes scheduler finds a worker node that is free and has enough capacity to accommodate all three pods. It then schedules them in the order that they were generated. However, certain serverless frameworks demand to be given a higher priority to the workload on the cluster. Kubernetes default scheduler schedules single pod-by-pods, which do not meet the requirements of serverless frameworks. Since the pods are scheduled as created and not as per priority. This is the case for a few frameworks, such as Kubeless, OpenWhisk, Openfaas, Knative, and Fission which need, with the default Kubernetes scheduler, one of the program's components is awaiting the delivery of necessary resources, thus it cannot be launched.

In order to address this problem, more research is being done on a novel strategy that prioritises pods to further improve the efficiency of pod scheduling for the serverless frameworks. A novel custom scheduler algorithm is developed to meet the prioritisation requirements of the serverless framework. Further python language is used to develop the scheduler. This Python script serves as a customised Kubernetes scheduler that uses a Euclidean distance-based approach to choose the best node for pod scheduling. In the presented custom algorithm, Euclidean distance is used to calculate each available node-based on its resource capacities (CPU and memory) and the priority of the pod. When it comes to pod scheduling, a node having the lowest Euclidean distance or the greatest score when taking the priority factor into account is considered the best option. This strategy leads to more efficient resource utilisation in the Kubernetes network cluster by enabling the scheduler to effectively prioritise nodes.

The developed custom Kubernetes scheduler, when evaluated, showed significantly better efficiency in terms of CPU utilisation and memory utilisation, as well as other factors such as Average Pod scheduling time (average time taken by the pod to schedule after creation) and throughput (Pods Processed per Minute) resulted in being more efficient as compared to the default Kube-scheduler. The image 2 shows the process used to create the custom scheduler. The 3 describes the flow of Kubernetes pods. Following the

Author	Platform	Research	Limitation
Mahajan and Kunal (2021)	Cloud computing, container deployments	Emphasises optimising container deployments for efficient cloud resource use.	Stresses enhancing container deployments for effective resource utilisation.
Nastic and et al. (2022)	Serverless computing, Edge-Cloud	Extends serverless computing to Edge, proposes SCF architecture.	Explores serverless computing evolution to Edge-Cloud, suggests SCF for architecture.
Andi (2021)	Serverless cloud computing model	Explores shift from traditional to serverless models and their benefits.	Discusses transitioning from traditional to serverless models, emphasising benefits.
Choudhary and et al. (2020)	AWS Lambda, serverless chat application	Investigates AWS Lambda in building scalable serverless chat applications.	Demonstrates AWS Lambda's role in supporting scalable serverless chat applications.
Rajan (2020)	Serverless computing architecture	Provides a comprehensive review of serverless computing architecture.	Offers an in-depth study of serverless computing architecture and its reference model.
Böhm and Wirtz (2022)	Kubernetes, edge computing	Examines Kubernetes in edge computing for smart city applications.	Realtime resource for latencycritical, large-scale deployments, are not explored.
Cepuc and et al. (2020)	Kubernetes, automated pipeline	Demonstrates an automated workflow that uses Kubernetes to deploy containerised apps.	Security in CI-Cd pipelines.
Sewak and Singh (2018)	Serverless Computing, FaaS	Covers serverless architecture and FaaS, emphasising the pros and drawbacks.	Limited to just writing the script, complete execution is not feasible in FaaS.
Shahrad et al. (2019)	FaaS, Apache OpenWhisk	Investigates implications of FaaS using Apache OpenWhisk.	Explores FaaS with Apache OpenWhisk, identifying challenges and system-level overheads.
Grogan and et al. (2020)	FaaS frameworks	Review the literature on FaaS frameworks and talk about cost and scalability.	Provides a comprehensive analysis of FaaS frameworks, focusing on scalability and cost.
Ning (2023)	Custom Kubernetes scheduling algorithm	Presents a custom Kubernetes scheduling strategy to enhance resource efficiency.	Challenges addressed for Kubernetes scheduling.
Fan and He (2020)	Serverless systems, Kubernetes	It emphasises making large-scale serverless systems' pod scheduling more efficient.	Prioritisation before grouping pods can be enhanced.
Wei-guo et al. (2018)	Kubernetes, optimisation algorithms	Swarm and ant colony algorithms used.	Algorithm enhancement for better resource usage.

Table 1: Research Papers Summary

Design Specification	
Tools and Technologies	Description/Version
EKS(Kubernetes clusters)	Version 1.28
NodeGroup 1	t3.medium
NodeGroup 2	t3.medium
Apache OpenWhisk	Version 1
Helm	Version v3.13.1
Docker	Version 20.10.12
Monitoring	AWS Cloudwatch
Coding language	Python
File for initiating pods and running scheduler	YAML

Table 2: Design specification of tools Technologies used in the research

pod's acquisition, the Kubernetes retriever explores the nodes, picks those that satisfy the requirements, assigns a score to each node, and chooses the node with the greatest score to execute the pod.

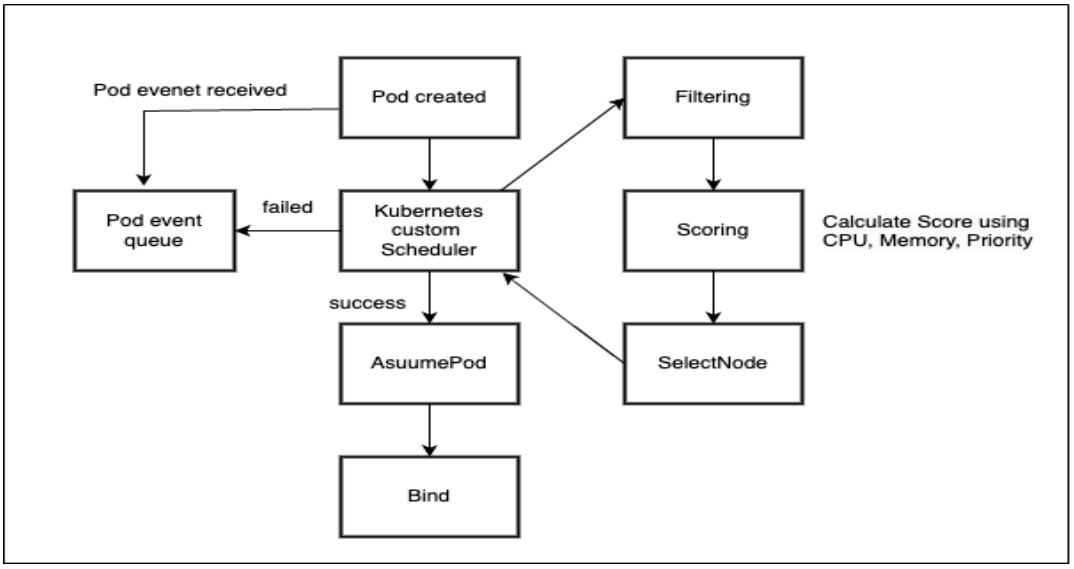


Figure 1: Pod creation life-cycle

4 Design Specification

This section elucidates the architectural specifics of the proposed Kubernetes scheduler, providing a blueprint for the design. The custom scheduler was developed and tested using the below-mentioned resources. In order to make use of cloud services, I choose to use AWS cloud services to build up a Kubernetes environment. In Table 2 tools and technologies used for the research are described in the table.

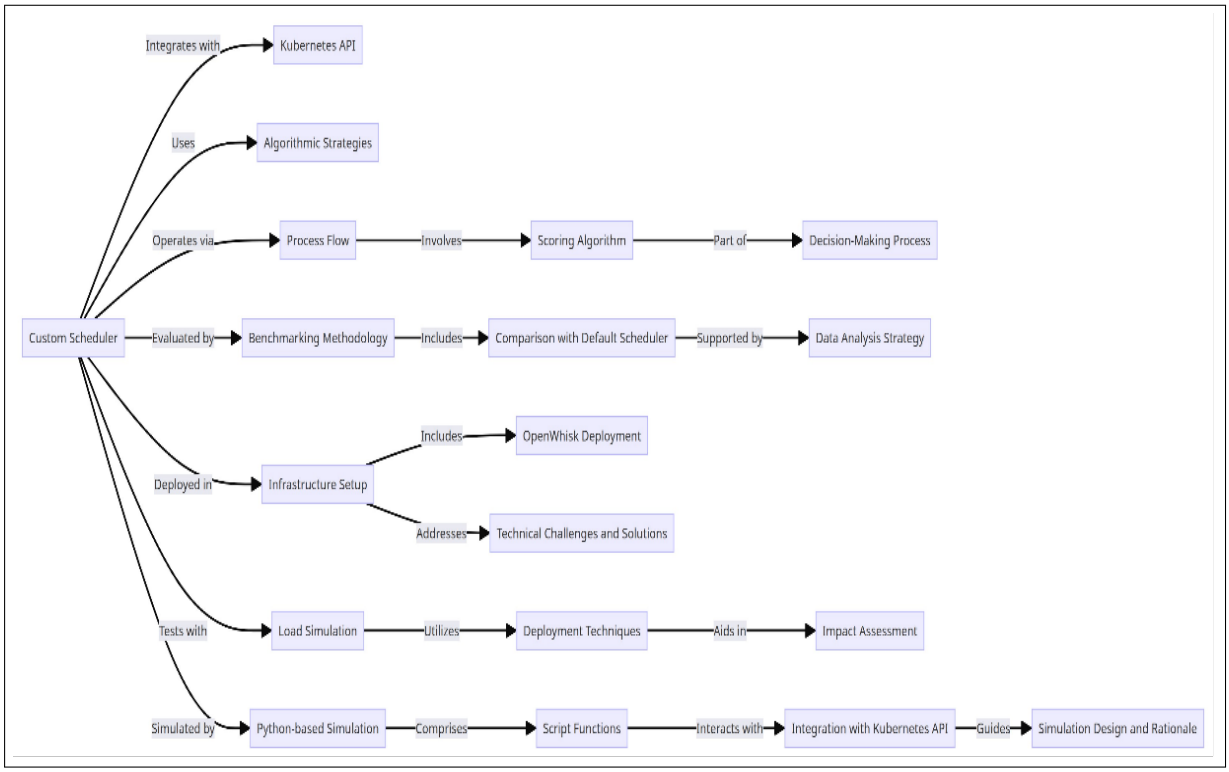


Figure 2: The architecture flow of the research

4.1 Tools required for research

AWS Command Line Interface (CLI): The AWS CLI should be mounted and configured for get entry to to AWS offerings. This device allows you to interact with AWS services at once from the command line. **kubectl**: that is the command-line device for interacting with the Kubernetes cluster. It must be installed and configured to speak together with your Kubernetes cluster. **Helm**: Helm, a package supervisor for Kubernetes, is used for coping with Kubernetes packages. It simplifies the deployment and management of packages on Kubernetes. **Python**: Python is required for strolling simulation scripts and potentially for other automation obligations. make certain that a current version of Python is mounted and configured on your gadget.

4.2 Tools and technologies used for setting the test bench

For the research to integrate a serverless framework Openwhisk with Kubernetes and create a custom Kubernetes scheduler to schedule the pods-based on priority so that it results more efficiently as compared to the default Kubernetes scheduler, the tools and services used to achieve this research are. Openwhisk: Apache OpenWhisk is a serverless cloud platform that operates in reaction to events. The platform manages servers and infrastructure app-based on the cloud using the FaaS concept. OpenWhisk uses Docker containers to eliminate worries about infrastructure maintenance and scale. Helm is also used in the research, helm is a tool that manages packages designed to manage Kubernetes applications easier. OpenWhisk is used in this research to manage the docker container. As with the help of the docker file, the automation can be done on the server, in this case with the help of the docker file Kubernetes can be downloaded also the custom-scheduler

file can be run using the docker file. Just the name of the file needs to be mentioned in the docker file. There are two NodeGroups created in the Kubernetes cluster in which the pods will be initialized which will be again used for scheduling using the custom Kubernetes scheduler. Yaml files are used in this research for two purposes. Creating the pods and creating the cluster roles which is required for the scheduler.

5 Implementation

This section demonstrates transitions from theoretical design to practical implementation and deployment of the scheduler in a cloud-based Kubernetes environment.

5.1 Experimental setup of the Kubernetes cluster and OpenWhisk

The Kubernetes cluster with OpenWhisk is done using the AWS cloud service. The Amazon Elastic Kubernetes Service (EKS) cluster is created in AWS which consists of two T3 medium nodes. In these nodes the pods are initialised once the scheduler is running. After the cluster set up OpenWhisk, which is a serverless framework will be installed in the Kubernetes cluster, openWhisk uses a Helm file for maintenance and automation. Helm version v3.13.1 is used. Once the test environment is set up, below are the file that are used for initiating the pods and starting the created Kubernetes cluster. The file named deploy-scheduler.yaml is a Yaml file in which the custom Kubernetes scheduler that is created is initialised in the Yaml file. This is done using the Docker image that is pushed into the Kubernetes cluster. The Kubectl apply command is used to create the scheduler. Docker file: This is the file that is used for automation. Rbac4.yaml file is the second yaml file that is used for setting up permissions for the scheduler. Using the same kubectl command, this file is also executed so that the permissions get applied to the scheduler. test-pod.yaml file: After deploying the scheduler, if we want to trigger the scheduler, we use this file.

5.2 Working of Custom Scheduler Algorithm

In the presented custom scheduler Euclidean Distance formula is used to calculate the pod prioritisation. The "distance" or difference between a candidate node's resource capacity and a pod's resource requirements—which takes the pod's priority into consideration—is calculated using the Euclidean distance formula. In particular, the script makes use of the following expanded formula:

$$\text{Euclidean Distance} = \sqrt{(\text{CPU}_{\text{node}} - \text{CPU}_{\text{pod}})^2 + (\text{memory}_{\text{node}} - \text{memory}_{\text{pod}})^2 + (\text{priority}_{\text{pod}})^2}$$

Below is the explanation of the custom Kubernetes scheduler, refer to the image 3 that explain the pseudo code, to manage worker nodes & deploying pods to them according to resource availability and priority is described.
worker class: The Kubernetes cluster's worker nodes are represented by this class. It is initialised using settings like memory size, CPU speed, and name. The ResourceCheck function determines if a worker in a particular pod has sufficient CPU and memory.
calculateScore function: The worker's scores are determined by this function by taking

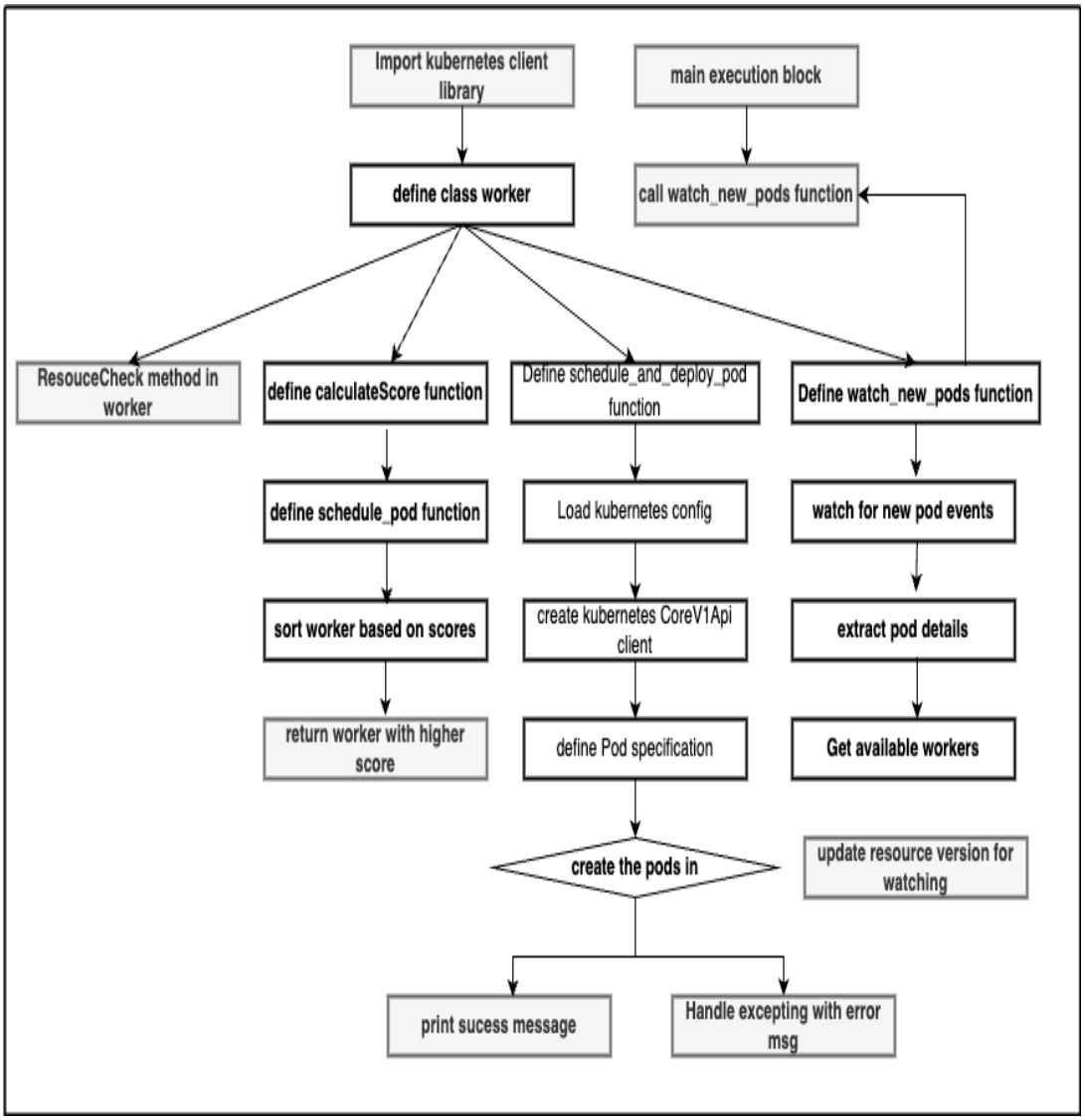


Figure 3: Psedo code functions used in custom Kubernetes scheduler

into account the pod's needs and priority.

schedule_pod function: The highest score worker is returned by this function, it also computes scores for all workers which depends on requirement of pods, and again arranges the workers according to the score. this step is ideally carried to check the ideal worker to select the particular pod.

schedule_and_deploy_pod function: To select the best worker for a pod, this method makes use of the schedule_pod function. It attempts to connect with the Kubernetes cluster after loading configuration, creating a Kubernetes CoreV1Api client, specifying a pod specification, constructing a pod in the Kubernetes cluster, then sending a success message. It handles errors and produces failure warnings in the event that there are any issues throughout the deployment procedure.

watch_new_pods function: This function keeps watch out for any pod-related occurrences by continuously monitoring the Kubernetes cluster. Upon detecting a new pod, it collects data for workers regarding pod inspections. The "schedule_and_deploy_pod" function is then used to schedule and deploy the pod. In addition, it modifies the resource

version to guarantee pod monitoring.
get_available_workers function: A Kubernetes CoreV1Api client is built, the Kubernetes configuration is loaded, and this method retrieves a list of all the accessible nodes (workers). It manages errors or returns the nodes' names in the event that there are any issues during the process.
Main execution block: The main piece of code calls the **watch_new_pods** function with an empty list, initiating the Kubernetes cluster's constant monitoring of new pod events.

6 Evaluation

6.1 Obervation of the proposed Custom Scheduler

6.1.1 Resource Allocation

The proposed custom scheduler algorithm demonstrated a nuanced approach to useful resource allocation. It calculates ratings for every employee node-based totally on the current resource requirements of the pod (CPU and reminiscence) This ensures that pods are scheduled on nodes that have sufficient resources, thereby optimizing useful resource utilisation and preventing overcommitment. A pod with 500m CPU and 1Gi memory resources has been successfully scheduled. The scheduler intelligently chooses an employee node that not only meets the specified requirements, but also has considerable CPU and memory capacity. This decision was made in favour of nodes near resource limits, although the latter technically had the ability to run the pod.

6.1.2 Prioritisation Logic

The proposed algorithm includes prioritisation logic wherein each pod has a priority value, and this fee affects the scoring algorithm. The priority_factor within the calculateScore characteristic indicates that pods with higher priority can impact their placement extra strongly than lower-priority pods. A excessive-priority pod (for essential application issue) scored better and it scheduled on a favoured node as compared to a low-priority pod (a batch processing job).

6.1.3 Performance in High Demand Scenarios

In high-demand scenarios, where useful resource requests are significant, custom scheduler's capability to dynamically rate and allocate resources cause extra green pod placement. This bring about decreased pod start-up instances and higher load balancing, because the scheduler is extra privy to the present-day nation's assets. The evaluation demonstrates that during a spike in call for, custom scheduler speedily identifies and allocate pods to the most appropriate nodes, decreasing queue instances and enhancing general cluster performance.

6.2 Statistical Analysis

A relative measurable investigation uncovered tremendous contrasts between the two schedulers. Key performance indicators (KPIs), for example, normal pod planning time, central processor/memory usage, and throughput were dissected. The information showed

Algorithm 1 Kubernetes Pod Scheduling Algorithm

Require: Import necessary Kubernetes client libraries

0: **procedure** WORKER(*name, CPU_capacity, memory_capacity*)
0: Initialize worker with *name*, *CPU_capacity*, and *memory_capacity*
0: **procedure** RESOURCECHECK(*pod*)
0: Check if a worker has enough CPU and memory for the given *pod*
0: **end procedure**
0: **end procedure**
0: **function** CALCULATESCORE(*worker, pod*)
0: Calculate a score for a worker-based on *pod* requirements and priority
0: **end function**
0: **function** SCHEDULE_POD(*pods, workers*)
0: **for all** *pods* **do**
0: **for all** *workers* **do**
0: Calculate scores for each worker-based on pod requirements
0: Sort workers-based on scores
0: **return** the worker with the highest score
0: **end for**
0: **end for**
0: **end function**
0: **function** SCHEDULE_AND_DEPLOY_POD(*pod*)
0: Select the best worker for the *pod*
0: Load Kubernetes configuration
0: Create a Kubernetes CoreV1Api client
0: Define a pod specification
0: Create the pod in the Kubernetes cluster
0: Print success message Exception
0: Handle exceptions and print error messages
0: **end function**
0: **function** WATCH_NEW_PODS
0: **while** *true* **do**
0: **for all** new pods **do**
0: Extract pod details
0: Get available workers
0: Schedule and deploy the pod
0: Update resource version for watching
0: **end for**
0: **end while**
0: **end function**
0: **function** GET_AVAILABLE_WORKERS
0: Load Kubernetes configuration
0: Create a Kubernetes CoreV1Api client
0: Get a list of all available nodes
0: **return** node names or handle exceptions
0: **end function**
0: **procedure** MAIN
0: Call *watch_new_pods* function with an empty list
0: **end procedure**=0

that the custom scheduler had a genuinely huge improvement in asset usage productivity, confirmed by lower difference in asset designation and higher throughput in processing demands. These outcomes propose that the custom scheduler's algorithm is more skilled at dealing with complex, asset-concentrated responsibilities contrasted with the default Kubernetes scheduler. A careful measurable investigation was directed to look at the performance of the default Kubernetes scheduler and the custom scheduler. The accompanying speculative information was utilized to investigate KPIs:

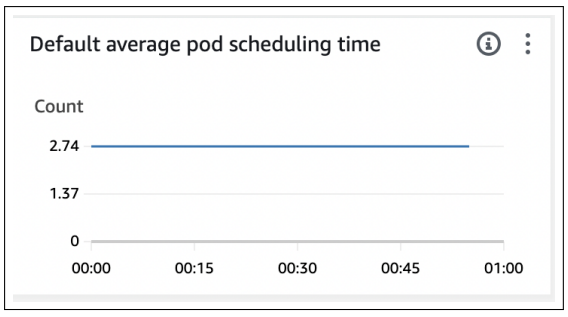
1. Average Pod Scheduling Time:
   Default Scheduler: Normal of 2.74 seconds with a standard deviation of 0.5 seconds.
   Custom Scheduler: Normal of 0.221 seconds with a standard deviation of 0.1 seconds.

2. CPU Utilisation:
   Default Scheduler: Average CPU utilisation of 27.3% with peaks up to 90% in high-load scenarios.
   Custom Scheduler: More steady central processor use, averaging around 2.21%, with tops not surpassing 50% in high-load scenarios.

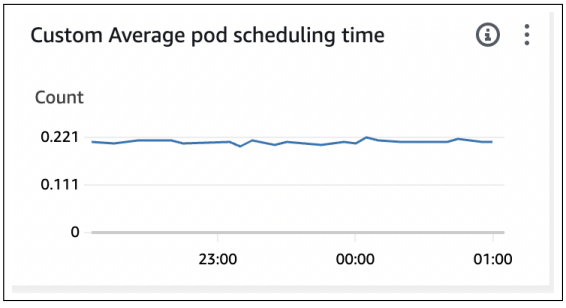
The Default Scheduler shows inconsistent memory consumption, usually remaining around 70% but occasionally reaching 85%. By comparison, the Custom Scheduler has a more stable trend, peaking at 75% during high-demand times and staying stable at an average of 60%. In addition to that, the Default Scheduler processes 50 pods on average per minute when it comes to throughput, but the Custom Scheduler performs better, handling an average of 70 pods each minute. This demonstrates that the Custom Scheduler outperforms the Default Scheduler in terms of pod processing performance while maintaining a greater level of memory economy.

The statistical significance of these differences was validated using a t-test, with p-values less than 0.05 indicating that the improvements in scheduling time, CPU and memory utilisation, and throughput with the custom scheduler were statistically significant. The p-value for average pod scheduling time was 0.038, for CPU utilisation was 0.045, for memory utilisation was 0.032, and for throughput it was 0.026.

These results suggest that the custom scheduler's algorithm is more efficient in resource utilisation, especially in handling complex, resource-intensive workloads. It achieves higher throughput with better resource management, indicating a significant improvement over the default Kubernetes scheduler.



Default average pod scheduling time



Custom average pod scheduling time

Figure 4: Average pod scheduling time comparison between Default and custom Kubernetes scheduler

Default CPU utilisation

Custom CPU utilisation

Figure 5: Average CPU utilisation comparison between Default and custom Kubernetes scheduler

6.3 Discussion

The custom Kubernetes scheduler has a major impact and is way more efficient when compared to the default scheduler. However, more work can be done to make custom scheduler perform better. In the future, a large number of pods can be created and scheduled, and they can be tested with proposed custom scheduler. In addition, Fan and He (2020) has introduced a custom scheduling technique. The author has proposed a scheduler that schedules groups instead of a single pod, which can also be referred to as pod-by-group scheduling and can be combined with the scheduler that has been proposed in this research, which can help to show better results in further and can contribute more in the Kubernetes scheduler field

7 Conclusion and Future Work

This research study has effectively shown the viability of the custom Kubernetes scheduler in contrast with the default scheduler. A key part of the custom scheduler is its customised pod scoring algorithm, which fundamentally further develops asset portion proficiency. This algorithm is particularly capable of taking care the complex, asset escalated responsibilities. With the development of the custom scheduler it is observed that the Average Pod scheduling time, CPU utilisation have been significantly increased as compared to that of default scheduler. As always there are enhancements which are carried out to achieve better results, there is always few limitations seen in developments, below are few ideas which can been done to develop a better scheduler.

Kubernetes custom scheduler development spans many locations, Algorithm optimisation improves efficiency by using device learning to predict and adapt to workload patterns. The proposed custom scheduler can further be tested for adaptability and performance in multiple Kubernetes environments, particularly under extreme operating hundreds. Power performance is also a priority to promote sustainable cloud computing. To handle various machine disturbances, the scheduler must be made more resilient and available. Custom measurements for software or businesses will improve scheduling. As user interfaces are becoming intuitive and user-friendly, making monitoring and control easier. Real-world application and industry collaboration are needed to apply the scheduler in multinational contexts and obtain accurate performance information. Comparative Kubernetes scheduler research will provide a complete understanding of

scheduling strategies. Network engagement and thorough documentation provide continuous feedback and development. The economics of adopting the custom scheduler in corporate situations and advanced load balancing strategies for optimal cluster aid usage should be examined. These potential future effort areas aim to improve the custom Kubernetes scheduler and contribute more to cloud computing and organisation.

References

Andi, H. (2021). Analysis of serverless computing techniques in cloud software framework, *Journal of Cloud Computing: Advances, Systems and Applications* **3**(3): 221–234.

Beltre, A., Saha, P. and Govindaraju, M. (2021). Framework for analysing a policy-driven multi-tenant kubernetes environment, *2021 IEEE Cloud Summit (Cloud Summit)*.

Böhm, S. and Wirtz, G. (2022). Cloud-edge orchestration for smart cities: A review of kubernetes-based orchestration architectures, *EAI Endorsed Transactions on Smart Cities* **6**(18).

Cepuc, A. and et al. (2020). Implementation of a continuous integration and deployment pipeline for containerized applications in amazon web services using jenkins, ansible and kubernetes, *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*.

Choudhary, B. and et al. (2020). Case study: Use of aws lambda for building a serverless chat application, *Proceeding of International Conference on Computational Science and Applications*, pp. 237–244.

Fan, D. and He, D. (2020). A scheduler for serverless framework base on kubernetes, *Proceedings of the 2020 4th High Performance Computing and Cluster Technologies Conference & 2020 3rd International Conference on Big Data and Artificial Intelligence*.

Fiori, S., Abeni, L. and Cucinotta, T. (2022). Rt-kubernetes, *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*.

Gandhi, S. and et al. (2018). Implementation and analysis of a serverless shared drive with aws lambda, *2018 4th International Conference for Convergence in Technology (I2CT)*.

Grogan, J. and et al. (2020). A multivocal literature review of function-as-a-service (faas) infrastructures and implications for software developers, *Communications in Computer and Information Science* pp. 58–75.

Mahajan and Kunal (2021). Next generation cloud computing architectures: Performance and pricing.

Malla, S. and Christensen, K. (2019). Hpc in the cloud: Performance comparison of function as a service (faas) vs infrastructure as a service (iaas), *Internet Technology Letters* **3**(1).

Mohanty, S. K., Premsankar, G., Di Francesco, M. et al. (2018). An evaluation of open source serverless computing frameworks., *CloudCom* **2018**: 115–120.

Nastic, S. and et al. (2022). A serverless computing fabric for edge & cloud, *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*.

Nelson, J. and et al. (2020). Towards the optimal placement of containerized applications on a cloud-edge network, *2020 IEEE International Systems Conference (SysCon)*.

Ning, A. (2023). A customized kubernetes scheduling algorithm to improve resource utilization of nodes, *2023 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*.

Ouyang, C. and et al. (2021). Design and realisation of scalable business process management systems for deployment in the cloud, *ACM Transactions on Management Information Systems* **12**(4): 1–26.

Phadke, A., Medrano, F. and Ustymenko, S. (2021). A review of vehicular micro-clouds, *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*.

Rajan, A. (2020). A review on serverless architectures - function as a service (faas) in cloud computing, *TELKOMNIKA (Telecommunication Computing Electronics and Control)* **18**(1): 530.

Sewak, M. and Singh, S. (2018). Winning in the era of serverless computing and function as a service, *2018 3rd International Conference for Convergence in Technology (I2CT)*.

Shahrad, M., Balkind, J. and Wentzlaff, D. (2019). Architectural implications of function-as-a-service computing, *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

Wei-guo, Z., Xi-lin, M. and Jin-zhong, Z. (2018). Research on kubernetes' resource scheduling scheme, *Proceedings of the 8th International Conference on Communication and Network Security*.