National College of Ireland

Latency-Aware Scheduling for Kubernetes: A Custom Approach for Cloud Environments

MSc Research Project
Cloud Computing

Himanshi Painuly
Student ID: x22143220

School of Computing
National College of Ireland

Supervisor: Prof Jitendra Kumar Sharma

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Himanshi Painuly
Student ID:	x22143220
Programme:	Cloud Computing
Year:	2023/24
Module:	MSc Research Project
Supervisor:	Prof Jitendra Kumar Sharma
Submission Due Date:	12/08/2024
Project Title:	Latency-Aware Scheduling for Kubernetes: A Custom Approach for Cloud Environments
Word Count:	5995
Page Count:	22

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:	
Date:	12th August 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):	

Latency-Aware Scheduling for Kubernetes: A Custom Approach for Cloud Environments

Himanshi Painuly
x22143220

Abstract

Cloud computing has made the microservices architecture a popular choice as it scales and offers flexibility, but maintaining these interconnected components efficiently is difficult. Current scheduling strategies in system such as Kubernetes lead to poor performance since they do not properly handle microservice dependencies leading to higher resource consumption and latency. In this research the focus is on these inefficiencies, which are crucial for organisations that are aiming to optimize network latency and reduce operational costs. Conventional Kubernetes schedulers often fails to look the dynamic nature of network performance, focusing primarily on static resource constraints like CPU and memory. Testing the Latency-Aware Scheduler(LAS) in this research introduces a custom latency-aware scheduler(LAS) that integrates real-time network latency measurements into the scheduling process when it is compared to the default Kubernetes scheduler. This custom scheduler calculates latency and available pods using custom node affinity rules, with the goal of minimising network delay by placing dependant pods on the same node. By modifying node affinity rules, incorporating a unique node scoring function, and converting network latency measurements into scheduling decisions it prioritises low network latency for maximum performance and user experience in latency-sensitive applications. Real-world testing in various cloud environments will reveal scalability and flexibility. This is essential for large-scale deployments.

Keywords: Kuberentes, Scheduling policies, Network Latency, AWS, Latency Aware Scheduler

1 Introduction

Kubernetes has become the leading platform for orchestrating containerised applications, providing powerful capabilities for managing deployments, scaling, and resource allocation. Nevertheless, with the rising adoption of microservices architectures and the deployment of applications across various cloud environments, Kubernetes encounters a number of obstacles. An important concern is the conventional scheduling approach, which primarily emphasises static resource constraints like CPU and memory. This approach frequently overlooks the dynamic and intricate nature of network performance, resulting in possible inefficiencies and issues with latency in service communication.

The basic Kubernetes scheduler's limitations become obvious in scenarios where network latency has significant effects on application performance. For latency-sensitive applications that require speedy and efficient communication across microservices, the typical scheduling algorithms fall short. They do not account for the real-time latency

between nodes and pods, which can result in inefficient pod placement, higher communication delays, and a general loss in application responsiveness and dependability. This gap emphasises the need for a more advanced scheduling system that incorporates network performance factors into its decision-making process. To address these difficulties, this study presents a customised latency-aware scheduler that optimises pod placement using real-time latency measurements. The custom scheduler uses network performance statistics to ensure that pods are scheduled in a way that reduces latency and improves communication efficiency across services. The scheduler enhances application performance and resource utilisation by exploiting real-time latency measurements, with a focus on the inefficiencies of traditional Kubernetes scheduling.

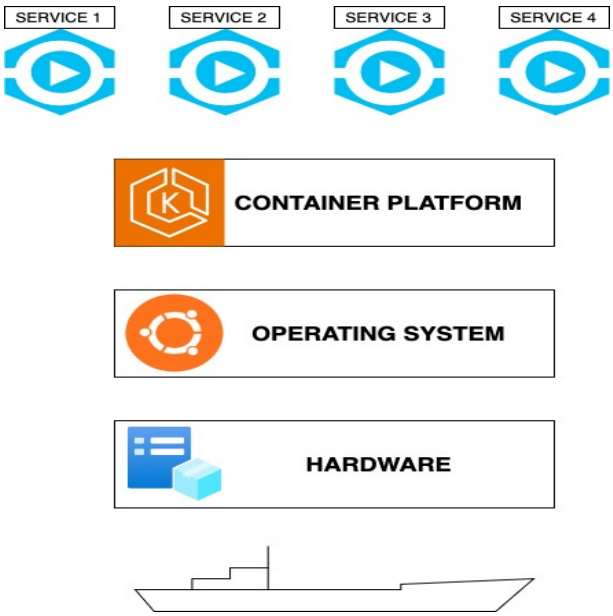

Figure 1: Evolution of Containers

The solution is deployed on AWS EC2 instances and utilises Prometheus for real-time monitoring and metrics collection. Prometheus, an open-source monitoring and alerting toolkit, is essential for collecting latency data and offering valuable insights into network performance. Through the integration of this data with the custom scheduler, the deployment on EC2 allows for a scalable and responsive scheduling system that can adjust to changing network conditions. This approach not only enhances the performance of applications that require low latency but also improves the efficiency and reliability of cloud-native deployments.

1.1 Background

With the rise of cloud-native architectures and the growing reliance on microservices, the task of managing network performance and latency becomes increasingly intricate. Although Kubernetes is highly proficient in deployment and resource allocation, it tends to prioritise static resource constraints such as CPU and memory, often neglecting the impact of network-latency between nodes and pods. This constraint results in less than optimal pod placements and increased communication delays, which have a negative impact on the performance of applications that are sensitive to latency.

In order to address these challenges, a new approach called latency-aware scheduling has been developed. This approach takes into account real-time network performance data when making scheduling decisions. With the inclusion of latency measurements, this method guarantees the best possible placement of pods, resulting in minimised communication delays and improved application responsiveness. This research presents a specialised scheduler that prioritises latency awareness by utilising real-time latency data collected through Prometheus on AWS EC2 instances. Through the optimisation of pod placement according to real-time network conditions, this solution enhances network performance and application efficiency, providing a more dependable and agile approach to cloud-native deployments.

1.2 Motivation

The development and deployment of Latency-aware custom scheduler in Kubernetes is a significant advancement in cloud computing and container orchestration, particularly for serverless deployments. The main goal of this study is to find ways to deal with the growing problems caused by network delay in modern cloud-native apps, especially those that use Kubernetes for management. As microservices designs become more common, how well services can communicate with each other has a direct effect on how well applications run and how users feel about them. Traditional Kubernetes schedulers are good at handling static resources like CPU and memory, but they don't take into account how network delay changes over time. This can cause big performance problems and waste.

Real-time data processing systems, financial services, and engaging web applications are all latency-sensitive and need the best network performance to make sure they have fast response times and are reliable. The normal scheduler focusses on allocating resources instead of network performance. As a result, these apps often have longer communication gaps and less-than-ideal pod placements, which makes them less responsive and effective overall. This gap shows how important it is to have a more advanced schedule system that uses real-time network delay data to make decisions. This optimisation reduces network latency and increases resource utilisation, enabling Kubernetes to be adapted to an organization's specific requirements. As a result, this method improves the scalability, resilience, and efficiency of serverless applications, making them more cost-effective and responsive to changing workloads.

1.3 Research Question

How can the pod scheduling policies, such as custom node affinity rules, be designed in Kubernetes to outperform the default pod-by-pod scheduler in reducing pod start-up time using the Latency-aware scheduler?

1.4 Objectives of Research

The objectives of this research is to develop a latency-aware scheduler(LAS) for Kubernetes which will optimizes the pod placement based on real-time network latency data, improving the performance and efficiency of latency-sensitive cloud-native applications.

- **Enhance Network Utilization**: Develop and implement dynamic affinity-rules in Kubernetes to place microservices in the best way possible based on how the

network is performing in real time. This will focus on reducing the latency and improve communication efficiency between the interdependent services.

- **Improve Quality of Service QoS**: Increase resource efficiency by intelligently allocating tasks based on VM load, thereby optimizing overall system performance and resource usage.
- **Develop a Custom Latency-Aware Scheduler**: Design and implement a custom Kubernetes scheduler that will integrate real-time network latency measurements to optimize pod-placement. This scheduler is designed to enhance the performance by by considering network latency along with other resource constraints such as CPU and memory.
- **Integrate Real-Time Monitoring**: Deploy and integrating Prometheus to monitor real-time network performance within the Kubernetes environment. This integration will allow the collection and analysis of latency-data to inform scheduling decisions, providing a dynamic approach to managing the pod placements. .
- **Evaluate Performance Improvements**: Evaluate the effects of the custom latency-aware scheduler on application performance through a comparison with the default Kubernetes scheduler. Metrics of interest involve decreases in network latency, enhancements in communication efficiency, and overall application responsiveness.
- **Deploy on AWS EC2**: Implementing and testing the custom scheduler on AWS EC2 instances is crucial to ensure optimal scalability and robustness in a cloud environment. This deployment will offer valuable insights into the performance of the scheduler in real-world scenarios and its ability to effectively manage applications with low latency requirements.

1.5 Outline

Finally, you can now close Section 1 by outlining the structure of the report, for instance: The remainder of the report is organised as follows. Section 2 offers important theory and evaluates works closely linked to the suggested strategy, laying the groundwork for understanding the next parts. Section 3 describes the suggested solution, which includes the development of unique pod scheduling strategies and the incorporation of bespoke node affinity criteria into a latency-aware scheduler. In Section 4, outlines the strategies used to solve the highlighted issue, such as developing latency-sensitive scheduling algorithms and test processes to evaluate the suggested strategy. goes over the implementation specifics, including process methods and the integration of the proposed solution into the Kubernetes environment.Section 5, displays the assessment findings, which demonstrate the proposed scheduler's efficacy in decreasing pod start-up times and compare it to the default schedulerSection 6. Section 7.presents findings and suggested future research paths, emphasising areas for further development and optimisation.

2 Related Work

Fazio et al. (2016)The researcher critically proposed techniques for modelling and querying microservices and datacenter resources, automating their selection and mapping de-

pending on a variety of performance requirements. He examined on frameworks for composing application-agnostic microservices to increase knowledge reuse and simplify complex interactions. The researcher also emphasised the need of monitoring microservices and datacenter resources together, as well as investigating predictive models for dynamic reconfiguration. The limitations remains due to the lack of standard microservice communication protocols, complex integration, ineffective tools for configuration recommendations, poor support for detailed dependencies in isolating microservice interference.

Shen et al. (2023)The research presents KaiS, a learning-based scheduling system built for edge-cloud networks to increase the long-term request processing throughput. KaiS uses graph neural networks to integrate system state information and various policy networks to control orchestration complexity, resulting in a 15.9% gain in throughput over prior systems. Though, managing resources across edge-cloud networks and integrating with Kubernetes (k8s) creates challenges because existing solutions frequently struggle with dynamic and diverse settings and are incompatible with k8s. KaiS tackles these difficulties by automating scheduling strategies and prioritising long-term throughput, demonstrating the potential of learning-based techniques to improve scheduling in dynamic environments.

Kataru et al. (2023)The paper evaluates four AWS deployment techniques like Elastic Beanstalk, EC2, CI/CD, and Docker & Kubernetes. It aims to identify the most efficient method for deploying applications with minimal latency, stressing the importance of technical nuances in AWS deployment strategies. The study emphasizes the need for organizations to understand and choose the right deployment technique to optimize user experience, focusing on cost efficiency and latency reduction. The challenges in the research includes a narrow scope of performance metrics and insufficient consideration of network configurations, data transfer costs, and regional variations also include factor it overlooks factors like fault tolerance, disaster recovery, and compliance requirements, which are crucial for comprehensive cloud application deployment. This research gap highlights the need for further exploration into optimizing microservice dependencies and enhancing the network latency to improve resource utilization and reduce network bandwidth use in Kubernetes environments.

2.1 Kubernetes with respect to Microservices in Cloud Computing

Miyazawa (2023)The objective of the study is to develop a communication latency-aware container scheduler for edge cloud environments that optimizes the deployment and reallocation of application containers based on communication latency between end devices and compute nodes. This scheduler addresses the challenges of heterogeneous network connectivity by routing application access requests to the best-suited computing node, thereby improving performance. However, the approach has limitations in effectively managing the diverse and dynamic network conditions found in edge environments, making it difficult to consistently make optimal, real-time decisions, which could have an impact on the system's overall performance.

Ding et al. (2023)The researcher proposed an integer nonlinear microservice placement model for Kubernetes with the goal of minimising costs while taking into account dy-

namic resource competition, microservice availability, and shared dependent libraries. To solve the model, a better evolutionary algorithm was used, resulting in faster throughput at the same or lower costs, hence optimising Kubernetes placement method. The model struggles to react to short-term and sudden request loads and potentially wasting resources or failing to meet performance requirements.

Medel et al. (2016)A Kubernetes performance model that uses a reference net-based technique to analyse and manage system resources. This model, which is based on real data from a Kubernetes deployment, helps with capacity planning and application design, notably the architecture of applications in terms of pods and containers. The study used a benchmark-based approach to better understand Kubernetes behaviour, emphasising the benefits of containers over virtual machines (VMs) due to their lower overhead. Yet, the model's reliance on VM cost as a restriction underscores the need for additional development to meet dynamic and real-time application needs in Kubernetes deployments.

Selvakumar1 et al. (2023)The researcher created a machine learning-based approach to dynamically reduce latency in microservices applications by assessing server capacity and queue characteristics, then prioritising service instances based on the shortest possible queue waiting time. This technique, which uses Netflix open-source software components, was evaluated with both interactive and non-interactive workloads and shown considerable gains in load balancing and latency reduction for extended chains of microservices. Although, the complexity of integrating machine learning components may provide implementation challenges and the solution's emphasis on lengthy service chains could make it ineffective for shorter chains or other performance concerns, limiting its broader usefulness.

2.2 Custom Scheduler in Kubernetes

Centofanti et al. (2023)The researcher proposed a unique Kubernetes scheduling strategy that prioritises user-perceived latency measured at the application layer above network-layer data. This approach uses an iterative discovery process to dynamically choose the best node for pod placement, which includes deploying ad-hoc service replicas to locate the node with the lowest latency for end users. The experimental findings indicated that this solution beats the default Kubernetes scheduler in terms of delay reduction. Nevertheless, the efficiency of this approach is dependent on latency measurement accuracy, which may be hampered by convergence time and user mobility issues. To solve these restrictions, future work will focus on optimising scheduling algorithms.

Ning (2023)The researcher designed a customised Kubernetes scheduling technique to improve node resource utilisation inside a cluster. This technique improves on the default Kubernetes scheduler by assigning suitable Request values and generating an optimised score mechanism to make better scheduling decisions. Performance assessments showed that the unique technique improves node resource utilisation and load balancing throughout the cluster. Though, the algorithm's emphasis on optimising certain parameters may restrict its effectiveness in cases that need consideration of additional aspects such as network latency. Moreover, its scalability and generalisability to various large-scale deployments have yet to be completely addressed.

Lai et al. (2023)To address the issue of node heterogeneity in Kubernetes-managed edge computing systems, the researcher created the Delay-Aware Container Scheduling (DACS) method. DACS assigns pods efficiently by taking into account both the leftover resources of worker nodes and any delays induced by pod assignments. Performance testing on a Kubernetes cluster constructed with VMware revealed that DACS dramatically lowers processing and network delays, beating other schedulers such as kube-scheduler, ElasticFog, TSRA, NAS, and KCSS. Yet the algorithm's success is dependent on precise delay measurements, and its scalability and adaptability in large-scale, dynamic edge environments requires more investigation.

2.3 Need for the Advanced Scheduler

Gog et al. (2016) the researcher designed Firmament, a centralised scheduler that can scale to over ten thousand computers with sub-second placement latency by combining several min-cost max-flow (MCMF) algorithms, addressing the issue sequentially, and applying problem-specific optimisations. Experiments with a Google workload trace from a 12,500-machine cluster revealed that Firmament improves placement latency by 20 times over the preceding Quincy scheduler and equals the latency of distributed schedulers for short jobs. In addition, Firmament outperformed four popular schedulers in terms of placement quality, improving batch job response time by six times. Despite its advantages, Firmament's reliance on continual rescheduling via MCMF optimisation may result in computational cost in highly dynamic contexts. Moreover, being a centralised system, it may encounter scalability and reliability difficulties, particularly under conditions of unpredictable workload spikes or hardware failures.

Merkouche et al. (2022)The researcher developed TERA-Scheduler, an advanced scheduling technique for Kubernetes that takes into account microservice dependencies to minimise the scheduling-duration and inter-pod traffic, resulting in a 39% improvement in response times over the default Kubernetes scheduler. Experiments shown that TERA-Scheduler improves latency-aware systems by detecting dependencies, obtaining resource needs, and implementing effective scheduling algorithms. Nevertheless, effective scheduling needs earlier identification of microservice dependencies, which adds complexity. Moreover, while the results are encouraging, additional research is needed to combine numerous resource measurements with QoS requirements and investigate implementation in private cloud settings for further optimisation.

2.4 Research Gap

Several research gaps have been discovered in existing microservice design and Kubernetes scheduling methodologies. One major gap is the absence of standardised communication protocols for microservices, which makes integration difficult and affects system efficiency. The lack of these standards prevents smooth communication across microservices, resulting in operational inefficiencies. Another key difficulty is the complexity of integration and setup procedures, especially in dynamic contexts such as edge-cloud networks, where resource allocation and network circumstances are constantly changing. This complexity typically leads to inefficient microservice setup and performance. Furthermore, present systems lack adequate real-time monitoring and dynamic reconfiguration capabilities, and current tools fail to deliver the critical real-time performance indicators required for op-

timum resource management. Pontarolli et al. (2020)Another important difficulty is the improper management of microservice dependencies, which are difficult to manage and may result in sub-optimal performance, particularly when they influence network latency and application responsiveness. Finally, typical Kubernetes scheduling approaches prioritise static resource measurements such as CPU and RAM while ignoring dynamic considerations such as network latency, resulting in wasteful pod placements and reduce the overall system performance.

Introducing a latency-aware scheduler is critical for filling important research gaps in microservice design and Kubernetes scheduling. It optimises microservice placement based on real-time network latency, hence increasing communication efficiency and system performance. This dynamic technique enables constant adaptation to changing network circumstances, improving resource utilisation and dealing with microservice dependencies more efficiently. The scheduler increases program performance and user experience by lowering network latency, particularly in real-time settings. Overall, a latency-aware scheduler is critical for developing resource management tactics in dynamic and complicated environments, providing significant improvements over standard static scheduling approaches.

3 Methodology

This methodology provides a comprehensive overview of the architecture and components of Kubernetes. It also introduces a specialised scheduler that is specifically designed to enhance pod placement decisions by taking latency into account. An overview of the Kubernetes master and worker nodes is provided, emphasising their roles and essential elements like the API Server, Controller Manager, Kubelet, and other components. The proposed custom scheduler incorporates real-time network latency data, which is collected through Prometheus, into the scheduling process. This method improves the placement of pods to prioritise applications with low latency requirements, resulting in better performance and resource usage in a Kubernetes cluster, especially in a cloud environment such as AWS EC2.

3.1 Components of Kubernetes and Proposed Design

3.1.1 Master Node

- **Function:** The master node is responsible for controlling the Kubernetes cluster, handling tasks such as managing the cluster's state, scheduling workloads, and overseeing administrative duties.

- **Components:**
  API Server: Serves as the interface for the Kubernetes API, managing all administrative commands and interactions with the cluster.
  Controller Manager: Manages controllers responsible for performing routine tasks like replication, node management, and deployment operations.
  Scheduler: Allocates pods to nodes according to specified scheduling policies and constraints.
  etcd: It is a reliable and highly-available key-value store that effectively manages the data and state of a cluster.

3.1.2 Worker Nodes

- **Function:** Worker nodes are responsible for executing applications and services. Every worker node is responsible for hosting the containers and managing their execution.

- **Components:**
  Kubelet: A crucial component that monitors container status on the node and maintains communication with the master node.
  Kube-Proxy: Kube-Proxy handles network routing and load balancing for services on the node.
  Container Runtime: the software that handles the execution of containers, such as Docker or containerd.

3.1.3 Pods

- **Function:** These are the smallest deployable units in Kubernetes. They encapsulate one or more containers that can share storage and network resources.

- **Characteristics:** Pods allow for the seamless deployment and management of applications as a single unit.

3.1.4 Services

- **Function:** Manages a group of pods and offers a consistent endpoint for accessing them, making it easier to balance the workload and find the services.

- **Types**: ClusterIP for internal access, NodePort for external access, LoadBalancer for integration with cloud load balancers, and ExternalName for external services.

3.1.5 Namespaces

Facilitate the division of cluster resources among multiple users or teams, enhancing resource organisation and management efficiency.

3.1.6 Deployments

Manage the deployment and scaling of pods, ensuring that the applications are maintained in the desired state.

3.1.7 ConfigMaps and Secrets

It is used to manage configuration data and sensitive information for applications running in the cluster.

3.1.8 Monitoring Tools

The Prometheus application will be accessible to external users through the Prometheus Kubernetes Service. The Grafana application will be available through the Prometheus Kubernetes Service, allowing external users to access and view the Grafana dashboards.

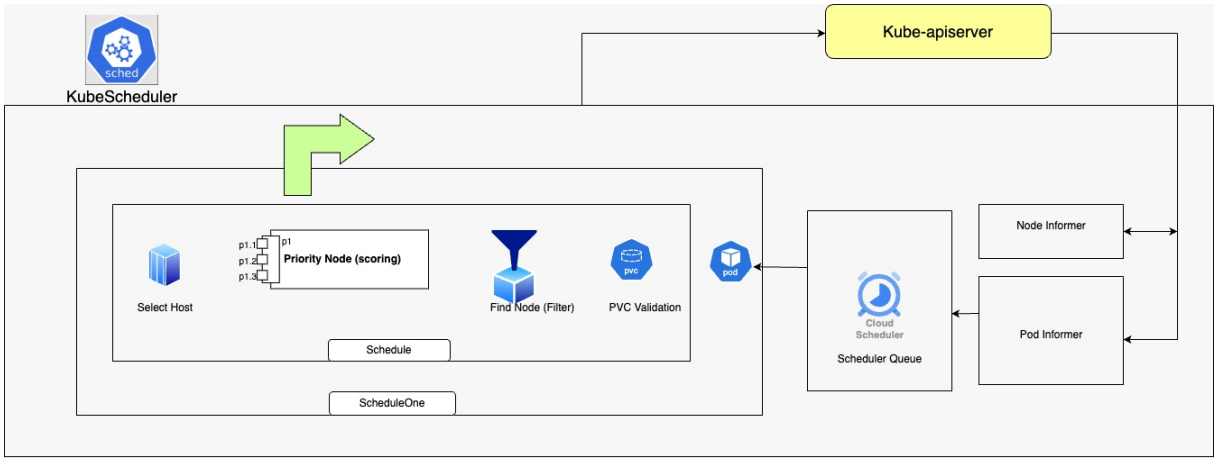

Figure 2: Kubernetes Custom Scheduler

3.2 Custom scheduler(Latency Aware)

The custom latency-aware scheduler is a major improvement compared to the traditional Kubernetes scheduling approach. It incorporates real-time network latency measurements into the scheduling process. Unlike the standard scheduler, which primarily considers static resource constraints like CPU and memory, this solution takes into account real-time latency data to enhance pod placement optimisation. This integration is accomplished through real-time monitoring with Prometheus, which consistently gathers and offers latency metrics. Through careful analysis of these metrics, the custom scheduler can make well-informed decisions regarding pod placements, guaranteeing that latency-sensitive applications are deployed on nodes that minimise communication delays. This approach tackles the performance limitations commonly found in traditional schedulers, resulting in improved efficiency and reduced latency caused by less-than-optimal pod placements.

Tang et al. (2023)The custom scheduler functions by collecting the latency data from Prometheus, which is subsequently utilised to evaluate the present network conditions between nodes and pods. The scheduler assesses potential placement options by considering latency information, with the goal of placing pods on nodes that provide the shortest communication latency. When the ideal placement conditions are not met, the scheduler has the ability to adapt its criteria and reassess the available nodes in order to achieve the most optimal pod placement. This approach not only improves the responsiveness and efficiency of latency-sensitive applications but also enhances overall resource utilisation within the Kubernetes cluster. Deploying this scheduler on AWS EC2 instances allows us to utilise scalable cloud infrastructure to efficiently handle real-world workloads and confirm the success of our latency-aware scheduling approach.

3.3 Workflow of the proposed custom scheduler(LAS)

In order to further optimise the scheduling process, particularly for latency-sensitive applications, several important components and enhancements have been incorporated, building on the workflow of the custom Kubernetes scheduler.

- **Latency Measurement Integration:** The scheduler includes a latency meter

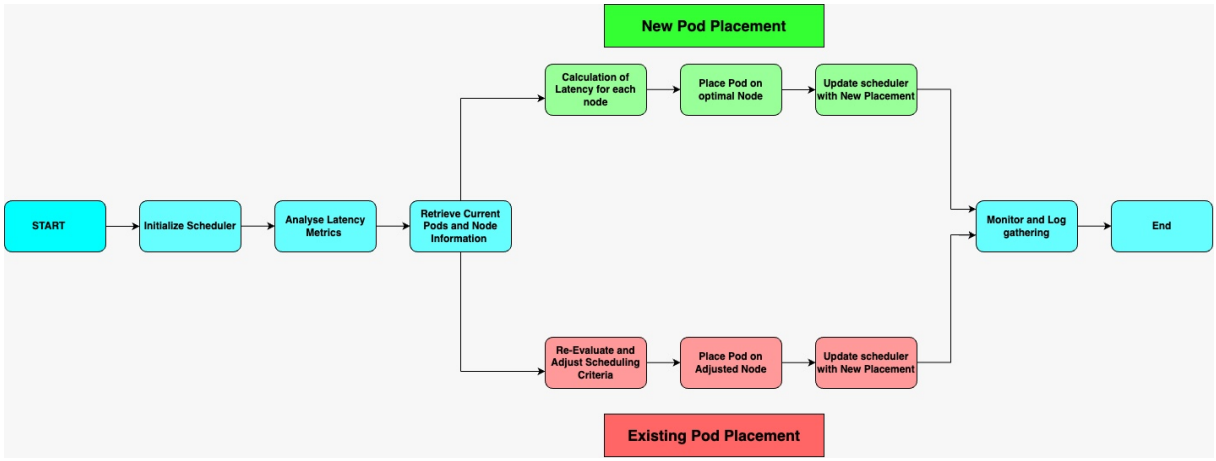

Figure 3: Workflow of the Proposed Custom Scheduler(LAS)

that dynamically assesses network performance between pods and nodes in the Kubernetes cluster. This latency meter is implemented as a Go-based application that acts as a proxy server, strategically placed within the cluster. It efficiently captures and tracks the time it takes for requests to travel between nodes, continuously recording these metrics to offer valuable insights into the current network conditions. Through the integration of latency measurements, the scheduler gains the ability to make more precise evaluations when selecting the optimal node for deploying pods, especially in environments where low latency is important.

- **Prometheus Monitoring:** In order to achieve real-time data collection and monitoring, Prometheus is deployed alongside the scheduler. The latency meter sends its captured data to Prometheus, which stores the metrics for ongoing analysis. This configuration enables the custom scheduler to retrieve up-to-date and pertinent latency data for making scheduling decisions. Having access to real-time latency information from Prometheus is essential for the scheduler to choose nodes that minimise network delays, resulting in enhanced performance for deployed applications.

- **Routing Manager:** A pod called a routing manager is deployed within the Kubernetes cluster to handle the network requests and manage traffic routing. This pod comes with specific annotations and environment variables that determine its behaviour, such as directing requests to a default service. The routing manager is crucial in efficiently managing latency measurements and routing operations. With its ability to manage traffic flow and route requests effectively, the routing manager plays a crucial role in providing the scheduler with valuable insights for optimising pod placements. .

- **Scheduler Logic:** The decision-making process of the scheduler is closely connected to the latency data that Prometheus collects and stores. When a new pod is scheduled, the scheduler retrieves the latest latency metrics from Prometheus, assesses the network conditions between potential nodes, and chooses the node with the lowest latency. If the most optimal node is not available, the scheduler will dynamically reassess and adapt its placement criteria to ensure that the next best option is chosen. This logic guarantees that applications, especially those that are

sensitive to network latency, will experience decreased communication delays and improved performance.

- **Deployment and Testing:** The entire system which includes the scheduler, latency measurement services, and routing managers, is deployed well and tested on AWS EC2 instances to validate its performance in a scalable cloud environment. Configuration files in Kubernetes are utilised for deploying these components, while scripts automate the process to guarantee accurate setup. This deployment phase is critical for testing the scheduler's effectiveness, with performance metrics such as latency and overall application responsiveness being closely monitored to measure improvements. With extensive deployment and testing, the custom scheduler is optimised to provide top-notch performance in real-world cloud environments.

4 Design Specification

The custom Kubernetes scheduler was developed and tested using the AWS cloud services to take advantage of the capabilities and flexibility of Elastic Compute Cloud (EC2) in building a cluster that is both scalable and dependable. The Kubernetes cluster was set up using Kubeadm, a tool that streamlines the process of creating multi-node clusters. The cluster configuration consisted of a master node hosted on an EC2 t3.large instance. The master node provided 2 CPUs, 16GB of storage for the master node, and 8GB for the worker nodes. This configuration ensured that the master node had the required computational power and storage capacity to efficiently oversee the entire cluster. Worker nodes, which handle application workloads, were deployed on t2.medium instances.

The custom scheduler was developed using Bash scripting and YAML configuration for its flexibility, simplicity, and minimal runtime overhead. Bash is a scripting language that is commonly used in the deployment of microservices. It is a great option for incorporating custom scheduling logic into Kubernetes. Node Exporter was installed on both the master and worker nodes to monitor the cluster's health and performance. This tool exports important metrics to Prometheus, a powerful monitoring system installed on the master node. Additionally Prometheus was deployed to offer a user-friendly and interactive platform for visualising the data that was collected. The installation and management of Prometheus were made more efficient by utilising Helm, a package manager for Kubernetes, which automated the deployment process. This extensive monitoring system ensured constant monitoring of the cluster's performance, allowing for prompt responses to any problems and maximising the overall efficiency of the Kubernetes environment.

4.1 Proposed Kubernetes Scheduler-Architecture

Kubernetes depends on its built-in scheduler, when it comes to assigning Pods to nodes within a cluster, . The scheduler takes into account various factors such as resource availability, node and pod relationships, and tolerations for taints. Nevertheless, if there is any specific requirements, you can use a custom scheduler. Users have the ability to define custom rules and logic for placing Pods, which can be extremely valuable for managing specialised resource needs, complex scheduling requirements, or specific dependencies. An example of a custom scheduler could be a Bash script that collects node and Pod data, applies user-defined criteria, and then makes scheduling choices based on that information. With these customisations, users can experience increased flexibility,

Tools and Technologies	Details/Edition
Cluster Creation Platform	AWS EC2
OS version	UBUNTU SERVER 22.04 LTS
Application Container	USER DEFINED MICROSERVICES
Containerization Orch Software	KUBERNETES 1.28.4
Software for Containerization	DOCKER 24.0.7
Monitoring Tools	PROMETHUS,NODE EXPORTER AND GRAFANA
Number of CPUs for Worker and Master	2 FOR EACH
Storage	16GB FOR MASTER AND 8GB FOR WORKERS
Coding Language	GO LANG
File comminutor for Pods and Nodes	YAML

Figure 4: Tools and Technologies

optimised resource usage, enhanced system reliability and fault tolerance, resulting in improved workload management and cluster performance.

4.1.1 Step 1: Formation of Kubernetes Cluster on AWS Cloud

Setting up a Kubernetes cluster on AWS EC2 instances is part of the initial stage. In this configuration setup, an EC2 instance is utilised for hosting the Kubernetes master node, while extra EC2 instances serve as worker nodes. The network architecture consists of a master node and several worker nodes, usually two or more, which together create the entire Kubernetes cluster. YAML files are commonly used to define and deploy the required Kubernetes pods and services. These YAML files provide a comprehensive overview of the configuration for different components, such as deployments, services, and configurations.

4.1.2 Step 2: Installation of Docker, Kubernetes,and Kubeadm

The second stage involves installing Docker, Kubernetes, and Kubeadm on all nodes using a shell script. This script provides the instructions and settings required to correctly configure the container runtime and Kubernetes components. The installation method involves running the script on both the master and worker nodes. It is essential to ensure that all nodes have Docker and Kubernetes installed and configured appropriately for the cluster and custom scheduler to run smoothly.

4.1.3 Step 3: Integration of Observability Tools

The observability tools such like Prometheus and Node Exporter are deployed to ensure that the custom scheduler is used efficiently. Prometheus is used to gather and display critical data including node availability, resource utilisation, and network bandwidth consumption. Each node runs Node Exporter, which collects system-level metrics. Grafana and Prometheus are connected to visualise the gathered data. This configuration provides the custom scheduler with real-time measurements needed to make educated scheduling choices based on latency and resource utilisation.

4.1.4 Step 4: Implementation and Execution of the Custom Scheduler

During this step, a custom latency-aware scheduler written in Go is deployed inside the Kubernetes environment. This scheduler is designed to improve pod placement by adding real-time latency and resource data into the decision-making process. It uses a mix of filtering and binding algorithms to assess node appropriateness in terms of latency and resource availability. The scheduler handles the difficulty of interdependent microservices by placing relevant services on the same node to reduce communication latency. This is accomplished by analysing latency data obtained from the Latency Meter and Prometheus metrics, which informs judgements on optimum node selection. The deployment process is optimised by automation using Bash scripts that handle YAML settings, service deployments, and scheduler execution. This method assures that the scheduler decreases network latency and increases overall application performance by making live data-driven placement decisions.

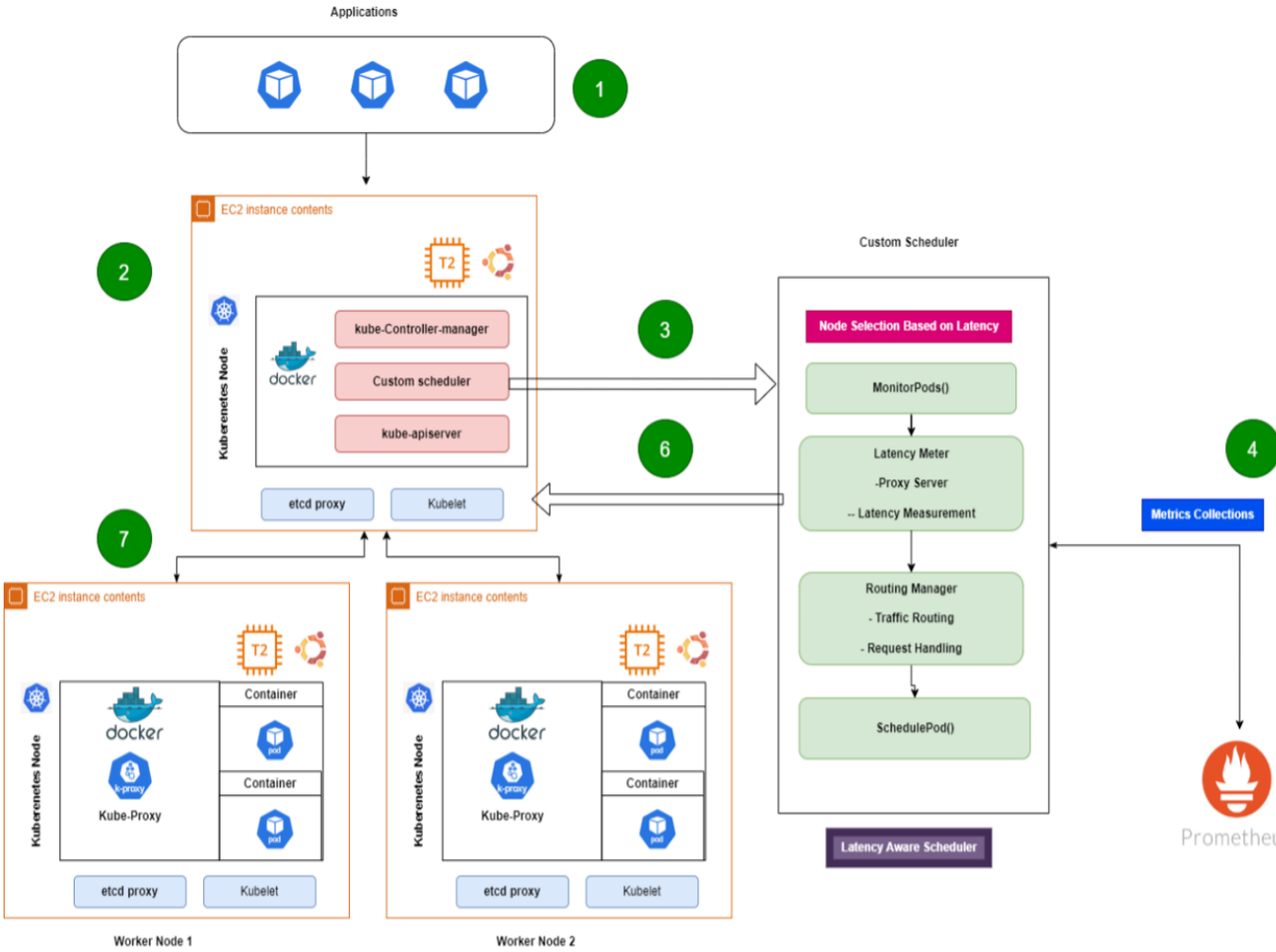

Figure 5: Architecture diagram of the Custom Scheduler

5 Implementation

The custom scheduler(LAS) optimises pod placement by integrating various components that consider real-time latency measurements. The key components consist of a latency meter, Prometheus monitoring, and a routing manager, all working together to deliver a complete scheduling solution.

- **Latency Measurement Integrating:** As part of their role, a database administrator utilises a latency meter, which is implemented in Go, to accurately measure the network performance between pods and nodes. This meter functions as a proxy server within the Kubernetes cluster, capturing the duration of requests as they travel between nodes. It computes latency by utilising a specific formula.

$$\text{Latency} = \text{End Time} - \text{Start Time} \tag{1}$$

  where Start Time is defined as when a request is sent and End Time is when the response is received. This dynamic measurement reflects the current network conditions, providing real-time latency data is crucial for making the scheduling decisions.

- **Prometheus:** It continually gathers and maintains latency measures, while the latency meter sends data for real-time analysis. In this query indicates the average duration of HTTP requests over the last 5 minutes, offering insights into network performance. Prometheus collects metrics using queries like:

$$\text{Latency Metric} = \text{avg\_over\_time}(\text{http\_request\_duration\_seconds}[5\text{m}]) \tag{2}$$

- **Routing Manager:** It is deployed as a Kubernetes pod, efficiently manages traffic routing and ensures precise latency data collection. The scheduler handles the data, choosing nodes based on their low latency and high resource availability. It uses a scoring system to determine the best nodes

- **Scheduler Logic:** The custom scheduler handles latency data gathered by the latency meter and Prometheus. When scheduling a new pod, the scheduler checks Prometheus for the most recent latency data and assesses network conditions between possible nodes. It finds the node with the lowest latency using the following method:

$$\text{Node Score} = \frac{1}{\text{Average Latency}} + \text{Resource Availability Score.} \tag{3}$$

  The scheduler provides higher scores to nodes with lower latency and more resource availability, ensuring that dependent microservices are clustered together to reduce communication delays. If the best node is unavailable, the scheduler reconsiders additional choices to ensure optimum placement.

- **Deployment and Testing:** The custom scheduler and its components are deployed on AWS EC2 instances, allowing for testing and validation in a scalable cloud environment. The deployment requires the application of Kubernetes configuration files for the scheduler, latency measurement services, and routing managers.

Automation scripts handle the setup process, ensuring that all components are properly configured and functioning. The performance of the scheduler is evaluated by measuring latency and overall application performance, confirming its effectiveness in optimising pod placement. This thorough approach guarantees that the custom scheduler efficiently minimises communication delays and improves the performance of latency-sensitive applications within the Kubernetes cluster.

5.1 Formation of a Kubernetes cluster

This section provides a detailed guide on how to set up a Kubernetes cluster on AWS using EC2 instances. The cluster consists of a master node and two slave nodes. The master node is configured on a T2.Large instance with Ubuntu 22.04 LTS to meet its increased computational requirements. It is responsible for running the custom scheduler, Control Panel, and API server. The two slave nodes are T2.Medium instances, each equipped with two CPUs. Commands are executed on each instance to form the cluster. Containerd is the container runtime in use, responsible for handling the entire container lifecycle and serving as a crucial link between the operating system and Kubernetes. Important components of Kubernetes are kubelet, which ensures containers run within pods, kubeadm, which assists with cluster configuration, and kubectl, the command-line tool for managing the cluster. The Container Network Interface (CNI) offers networking capabilities for containers. Docker is a popular platform for building, packaging, and running containers. Kubernetes is compatible with various container runtimes. With the help of these tools and technologies, you can effortlessly deploy and manage a Kubernetes cluster on AWS. This ensures that containers within the cluster are orchestrated and operated smoothly.

5.2 Working of the Proposed scheduling algorithm

The custom scheduler, written in Go, seamlessly integrates with the Kubernetes control plane through API calls during its startup phase. It sets up informers to keep an eye on the cluster, specifically monitoring the status of nodes and unscheduled pods. When a new pod needs scheduling, the scheduler enters the "ScheduleOne" phase, during which it chooses the most suitable node for deploying the pod. At the same time, a custom Bash script is run to verify dependencies among microservices. This script utilises methods such as "identify available nodes()" and "check any latency on the process running on node()" to generate a comprehensive list of nodes and analyse any existing latency issues.

The scheduler evaluates the nodes, taking into account resource availability and the results of latency checks, in order to select the optimal node for the pod. After a decision is reached, the scheduler assigns the pod to the chosen node and sends out an event to inform the system of its selection. This scheduler has a caching mechanism that enhances network efficiency by facilitating data exchange between microservices, resulting in reduced bandwidth usage within the Kubernetes framework. This design optimises pod placement to minimise latency and network overhead, resulting in improved system performance.

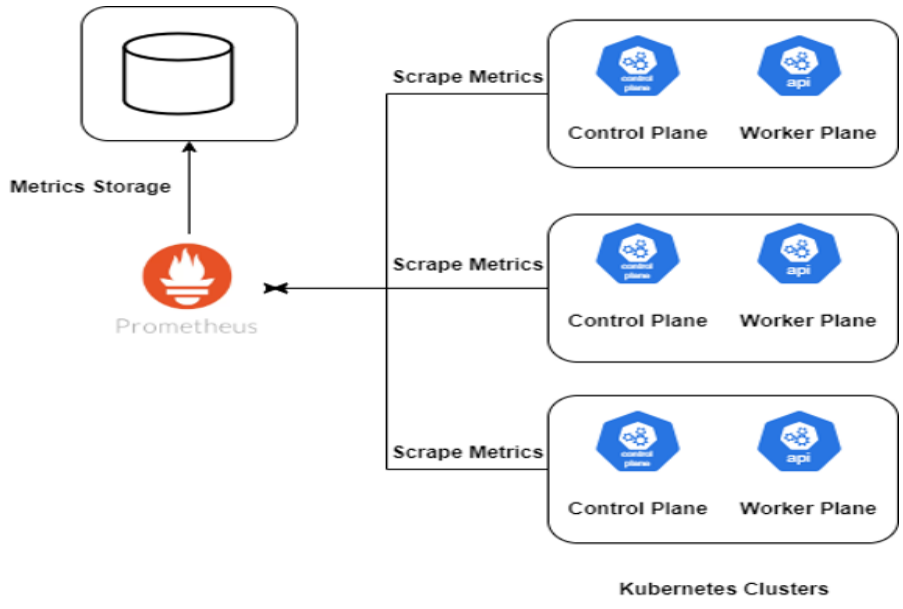

Figure 6: Proposed scheduling Algorithm working

5.3 Prometheus Monitoring Tool

Monitoring plays a vital role in Kubernetes, and Prometheus with Node Exporter greatly improves this capability by capturing comprehensive node-level metrics. Node Exporter offers comprehensive insights into system performance, including resource utilisation, network behaviour, and overall system well-being. Integrating these metrics into Prometheus provides a comprehensive understanding of node-specific performance. This information can be utilised to enhance the custom scheduler by integrating real-time metrics for more accurate node configurations. Having the ability to visualise these metrics in Grafana dashboards provides a complete perspective of the cluster and its individual nodes. This allows for well-informed decisions regarding workload distribution, resource optimisation, and cluster management. Utilising Prometheus, Grafana, and Node Exporter together forms a robust observability stack for Kubernetes. Following are the some components of Monitoring tools :

- **Data Collection Agents:** Collect system and application metrics(e.g., Node Exporter)
- **Quering and Alerting :** The querying and alerting engine is responsible for retrieving data and triggering alerts when certain conditions are met.
- **storage backend :** The storage backend securely stores large volumes of collected data.

5.4 Key Differences from the Default Kubernetes Scheduler

The custom latency-aware scheduler has notable differences compared to the default Kubernetes scheduler. It utilises real-time network metrics to optimise pod placement and minimise delays for latency-sensitive applications. This is in contrast to the default scheduler, which only prioritises resource availability. The custom scheduler seamlessly integrates with Prometheus to provide real-time latency data, empowering better

decision-making. In contrast, the default scheduler relies on fixed resource information. It utilises sophisticated logic to allocate pods based on latency and dependencies, striving for improved co-location of microservices, while the default scheduler takes a more generalised approach. In addition, it dynamically adapts placements according to current latency conditions and utilises Prometheus for comprehensive monitoring, providing better observability compared to the limited insights of the default scheduler.

6 Evaluation

To evaluate the effectiveness of our custom latency-aware scheduler, we conducted simulations of network delays using the 'tc' (traffic control) tool. Initially, we examined the existing network setup by using the command 'tc qdisc show' for the network interface 'ens5'. This provided us with a snapshot of the network's configuration. Next, a 400-millisecond delay was introduced to one of the nodes by utilising the command 'sudo tc qdisc add dev ens5 root netem delay 400ms'. This caused a delay in the network connection on that node. We confirmed the delay was applied correctly by running 'tc qdisc show' again.

Observing the custom scheduler's handling of pod placement in the Kubernetes cluster, we closely monitored the effects of the delay. Given the 400 ms delay on node 2, the scheduler made the decision to allocate pods on node 1 instead, as it had lower latency. This demonstrated the scheduler's ability to make intelligent choices by considering network performance. It would select nodes with superior network conditions to reduce any delays in pod communication. In general, the test results indicate that the scheduler successfully takes into consideration network delays and optimises the placement of pods to enhance performance.

```
ubuntu@k8s-master:~/latency_aware_scheduler/yaml-samples$ kubectl get pods -n kube-system | grep latency-aware-scheduler
latency-aware-scheduler             1/1     Running   2 (3h27m ago)   13h
ubuntu@k8s-master:~/latency_aware_scheduler/yaml-samples$ nano default-scheduler-deployment.yaml
ubuntu@k8s-master:~/latency_aware_scheduler/yaml-samples$ nano custom-scheduler-deployment.yaml
ubuntu@k8s-master:~/latency_aware_scheduler/yaml-samples$ kubectl apply -f default-scheduler-deployment.yaml
deployment.apps/default-scheduler-deployment created
ubuntu@k8s-master:~/latency_aware_scheduler/yaml-samples$ kubectl apply -f custom-scheduler-deployment.yaml
deployment.apps/custom-scheduler-deployment created
ubuntu@k8s-master:~/latency_aware_scheduler/yaml-samples$ kubectl get pods -l app=myapp-default -o custom-columns=NAME:.metadata.name,SCHEDULER:.spec
.schedulerName
kubectl get pods -l app=myapp-custom -o custom-columns=NAME:.metadata.name,SCHEDULER:.spec.schedulerName
NAME                                    SCHEDULER
default-scheduler-deployment-547496cf6f-jk6qk   default-scheduler
default-scheduler-deployment-547496cf6f-pjs6k   default-scheduler
NAME                                   SCHEDULER
custom-scheduler-deployment-665bf8ff9f-bzpr6   latency-aware-scheduler
custom-scheduler-deployment-665bf8ff9f-t88gl   latency-aware-scheduler
ubuntu@k8s-master:~/latency_aware_scheduler/yaml-samples$
```

6.1 Testing the Custom Scheduler

- **Installing the Latency Meter:** Use the command kubectl apply -f latency-meter.yaml to deploy the latency meter app. This application is designed to monitor and record network delays, and it will automatically transmit the collected data to Prometheus.

- **Configure the Routing Manage:** Utilise the 'kubectl apply -f routing-manager.yaml'

command to deploy the routing manager and effectively manage traffic within the cluster.

- **Deploy a Test App:** Deploy a sample app, such as Nginx, with ‘kubectl apply -f nginx-deployment.yaml‘ to test how the scheduler works.

6.2 Simulate Network Issues

- **Verify Current Network Configuration:** Utilise the ‘tc qdisc show‘ command to view the existing network setup on your nodes.
- **Introduce Network Delay:** To simulate a network delay on one node, execute the command ‘sudo tc qdisc add dev ens5 root netem delay 400ms‘.
- **Confirm the Delay:** - Execute ‘tc qdisc show‘ once more to ensure that the delay was implemented correctly.

```
qdisc netem 8001: dev ens5 root refcnt 3 limit 1000 delay 400ms
qdisc noqueue 0: dev docker0 root refcnt 2
qdisc noqueue 0: dev flannel.1 root refcnt 2
qdisc noqueue 0: dev cni0 root refcnt 2
qdisc noqueue 0: dev veth45843bb4 root refcnt 2
qdisc noqueue 0: dev vethd659415e root refcnt 2
ubuntu@k8s-master:~/latency-aware-scheduler$ kubectl get pods
NAME                                READY   STATUS    RESTARTS       AGE
app-deployment-659d65868f-qtdwk     2/2     Running   22 (47m ago)   4d23h
app-deployment-659d65868f-xb4x2     2/2     Running   22 (47m ago)   4d23h
app-deployment-78d9857f87-hjq9t     0/2     Pending   0              2d1h
nginx-deployment-55db865859-7n27m   2/2     Running   24 (47m ago)   5d5h
nginx-deployment-55db865859-w177k   2/2     Running   24 (47m ago)   5d2h
test-app-d9cdc56cb-2rppm            0/1     Pending   0              2d1h
test-app-d9cdc56cb-45q68            0/1     Pending   0              2d1h
test-app-d9cdc56cb-5d2k8            0/1     Pending   0              2d1h
test-app-d9cdc56cb-cnwhc            0/1     Pending   0              2d1h
test-app-d9cdc56cb-dlmw5            0/1     Pending   0              2d1h
test-app-d9cdc56cb-mrpbr            0/1     Pending   0              2d1h
test-app-d9cdc56cb-ppkj8            0/1     Pending   0              2d1h
test-app-d9cdc56cb-q2xnk            0/1     Pending   0              2d1h
test-app-d9cdc56cb-rxncq            0/1     Pending   0              2d1h
test-app-d9cdc56cb-x9xhb            0/1     Pending   0              2d1h
test-pod                            1/1     Running   47 (47m ago)   5d4h
ubuntu@k8s-master:~/latency-aware-scheduler$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS       AGE    IP            NODE       NOMINATED NODE   READINESS GATES
```

6.3 Test Pod Placement

- **Deploy Pods with the Custom Scheduler:** Deploy a few test pods and observe the placement decisions made by the custom scheduler.
- **Monitor Placement Decisions**: Verify whether the scheduler is assigning pods to nodes with lower network delay.

6.4 Evaluate How the Scheduler Performs

- **Verify Pod Placement:** Utilise ‘kubectl get pods -o wide‘ to ensure that pods are assigned to nodes with minimal latency.
- **Reviewing Latency Data:** Utilise Prometheus and Grafana to analyse the collected latency data and assess the alignment between the scheduler’s decisions and the simulated network conditions.
- **Review Logs:** Verify the logs of the custom scheduler and latency meter to ensure proper functionality and accurate data usage.

```
NAME          STATUS   ROLES           AGE     VERSION
k8-node1      Ready    <none>          3d16h   v1.28.12
k8-node2      Ready    <none>          3d16h   v1.28.12
k8s-master    Ready    control-plane   3d16h   v1.28.12
```

6.5 Results

For the evaluation of the custom latency-aware scheduler, a Kubernetes cluster was deployed on AWS EC2 instances. Realistic conditions were simulated by introducing network latency to test its performance. This specialised scheduler, created to enhance pod placement by taking network latency into account, was seamlessly integrated with Prometheus for live latency monitoring and a customised latency meter to dynamically assess network performance. For the purpose of emulating network conditions, a delay of 400 milliseconds was implemented on one of the nodes using the 'tc' (traffic control) tool. This delay was implemented to simulate real-world network latency and evaluate the performance of the custom scheduler in handling such situations. After that, the custom scheduler was used to allocate pods within the cluster. The scheduler utilised latency data gathered from Prometheus and the latency meter to make well-informed placement choices. According to the figure and calculations, it clearly states that the LAS is much capable when compared to default scheduler with consideration to network-latency and resource utilisation. It is clearly evident LAS works better in the application pod-placement.

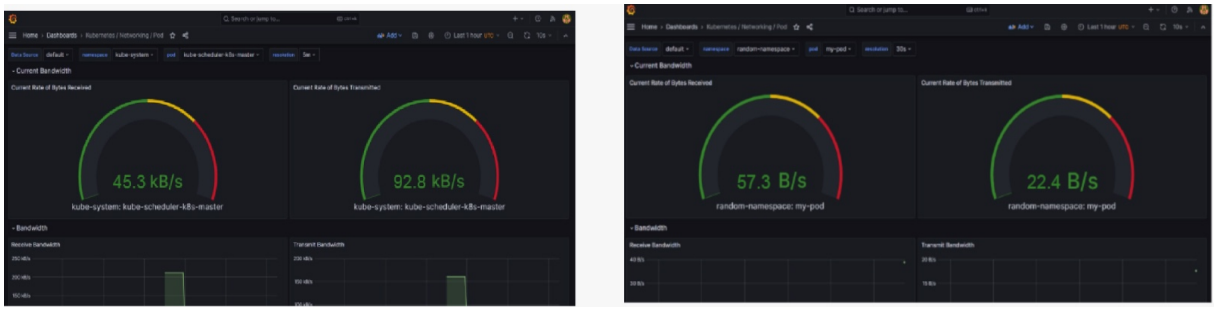

Figure 7: Promotheus Results

The testing showed that the custom scheduler efficiently used the latency data to prioritise nodes with lower network delays. As an example, when node 2 encountered a delay of 400 milliseconds, the scheduler always chose to assign pods to node 1, which had a lower latency. This behaviour showcased the scheduler's ability to minimise network delays and decrease communication overhead between microservices, aligning with its design to improve performance by optimising pod placement according to current network conditions.The Prometheus metrics provided precise and practical latency data, while the Grafana visualisations confirmed that the scheduler made decisions based on up-to-date network conditions. The graphs clearly demonstrated distinct variations in latency among nodes, confirming the efficiency of the scheduler. The scheduler's ability to enhance application performance and responsiveness was demonstrated by the consistent prioritisation of nodes with lower latency in simulated conditions.

On the other hand, the assessment also pointed out a few constraints. The simulated latency may not accurately reflect the real-time fluctuations in network performance

encountered in live environments. In addition, the integration of Prometheus and the custom latency meter can complicate matters and potentially lead to performance issues if not handled properly.

Metric	Diagram 1	Diagram 2
Namespace	kube-system	random-namespace
Pod	kube-scheduler-k8s-master	my-pod
Current Rate of Bytes Received	45.3 kB/s	57.3 B/s
Current Rate of Bytes Transmitted	92.8 kB/s	22.4 B/s

7 Conclusion and Future Work

The efficiency of a custom scheduler (LAS)that takes into account real-time network latency measurements to optimise pod placement within Kubernetes clusters. By incorporating a Go-based latency meter, utilising Prometheus for monitoring, and implementing a custom scheduling algorithm, the scheduler is able to dynamically evaluate network conditions. This results in a notable enhancement in performance as it effectively reduces latency between dependent microservices. This approach differs from the default Kubernetes scheduler, which focusses on static resource metrics and may not consider network performance, resulting in possible inefficiencies. The versatility of the custom scheduler in adjusting to fluctuating network conditions highlights its potential for practical implementation in a wide range of cloud environments. This, in turn, provides improved efficiency for applications that are sensitive to latency.

In order to further enhance the scheduler's capabilities, future research could focus on testing its performance in dynamic network environments, assessing its scalability in complex topologies and multi-cloud scenarios, and improving the scheduling algorithms to incorporate metrics like throughput and packet loss. Incorporating the scheduler with other Kubernetes components, such as horizontal pod autoscaling, can result in more extensive optimisations. Collecting feedback from actual users and performing usability testing will be crucial in improving the interface and configuration options of the scheduler. These efforts can greatly improve the scheduler's capabilities, leading to more efficient Kubernetes deployments and better application performance in cloud-native environments.

References

Centofanti, C., Tiberti, W., Marotta, A., Graziosi, F. and Cassioli, D. (2023). Latency-aware kubernetes scheduling for microservices orchestration at the edge, *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, pp. 426–431.

Ding, Z., Wang, S. and Jiang, C. (2023). Kubernetes-oriented microservice placement with dynamic resource allocation, *IEEE Transactions on Cloud Computing* **11**(2): 1777–1793.

Fazio, M., Celesti, A., Ranjan, R., Liu, C., Chen, L. and Villari, M. (2016). Open issues in scheduling microservices in the cloud, *IEEE Cloud Computing* **3**(5): 81–88.

Gog, I., Schwarzkopf, M., Gleave, A., Watson, R. N. M. and Hand, S. (2016). Firmament: Fast, centralized cluster scheduling at scale, *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, Savannah, GA, pp. 99–115.
URL: *https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog*

Kataru, S. S., Gude, R., Shaik, S., Kota, L. V. N. S. S. S., S, S. and M, B. R. (2023). Cost optimizing cloud based docker application deployment with cloudfront and global accelerator in aws cloud, *2023 International Conference on Sustainable Communication Networks and Application (ICSCNA)*, pp. 200–208.

Lai, W.-K., Wang, Y.-C. and Wei, S.-C. (2023). Delay-aware container scheduling in kubernetes, *IEEE Internet of Things Journal* **10**(13): 11813–11824.

Medel, V., Rana, O., Bañares, J. and Arronategui, U. (2016). Modelling performance resource management in kubernetes, *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pp. 257–262.

Merkouche, S., Haroun, T., Bouanaka, C. and Smaali, M. (2022). Tera-scheduler for a dependency-based orchestration of microservices, *2022 International Conference on Advanced Aspects of Software Engineering (ICAASE)*, pp. 1–8.

Miyazawa, H. (2023). A latency-aware container scheduling in edge cloud computing environment, *2023 Congress in Computer Science, Computer Engineering, Applied Computing (CSCE)*, pp. 1728–1731.

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)*, pp. 588–591.

Pontarolli, R. P., Bigheti, J. A., Fernandes, M. M., Domingues, F. O., Risso, S. L. and Godoy, E. P. (2020). Microservice orchestration for process control in industry 4.0, *2020 IEEE International Workshop on Metrology for Industry 4.0 IoT*, pp. 245–249.

Selvakumar1, G, J. and L. S., A. S. (2023). Csse - tech science press.
URL: *https://www.techscience.com/csse/v46n1/51304*

Shen, S., Han, Y., Wang, X., Wang, S. and Leung, V. C. M. (2023). Collaborative learning-based scheduling for kubernetes-oriented edge-cloud network, *IEEE/ACM Transactions on Networking* **31**(6): 2950–2964.

Tang, J., Jalalzai, M. M., Feng, C., Xiong, Z. and Zhang, Y. (2023). Latency-aware task scheduling in software-defined edge and cloud computing with erasure-coded storage systems, *IEEE Transactions on Cloud Computing* **11**(2): 1575–1590.