Maximizing Power Efficiency in K8s using Custom Scheduler

MSc Research Project
Cloud Computing

Vishnu Praneeth Divi
Student ID: x23293080

School of Computing
National College of Ireland

Supervisor: Shreyas Setlur Arun

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Vishnu Praneeth Divi
Student ID:	x23293080
Programme:	Cloud Computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Shreyas Setlur Arun
Submission Due Date:	11/08/2025
Project Title:	Maximizing Power Efficiency in K8s using Custom Scheduler
Word Count:	5796
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:	Divi Vishnu Praneeth
Date:	11th August 2025

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):	

Maximizing Power Efficiency in K8s using Custom Scheduler

Vishnu Praneeth Divi
x23293080

Abstract

This research addresses the challenge of energy efficiency in Kubernetes environments where container orchestration decisions significantly impact data center power consumption. optimizing energy consumption has become challenge for environmental sustainability and operational cost reduction. The default Kubernetes scheduler primarily focuses on CPU and memory requests rather than actual resource usage patterns, leading to ineffecincy energy utilization across clusters. This study introduces an approach to Kubernetes scheduling that incorporates real-time power metrics and parameter-optimized models to make energy-aware placement decisions.

1 Introduction

In the rapidly evolving landscape of cloud computing, containerization has emerged as the dominant paradigm for application deployment and management. Kubernetes, as the de facto standard for container orchestration, manages workloads across clusters of nodes, making scheduling decisions that significantly impact resource utilization and energy consumption. However, as data centers currently account for approximately currently representing 2% of the world's energy consumption, and recent forecasts expect it to grow even further to at least 8% by 2030 Vijouyeh et al. (2020). Energy consumption has risen due to several key factors: the surge in data collection, the increasing demand for computationally-intensive tasks driven by Artificial Intelligence (AI), and the end of Moore's law's exponential Shalf (2020) there is an need to address energy efficiency in containerized environments .

The default Kubernetes scheduler operates through a two-phase process: filtering and scoring. During filtering, the scheduler identifies nodes that satisfy the basic requirements of a pod, such as CPU and memory requests. In the scoring phase, it ranks these feasible nodes according to various criteria, ultimately selecting the highest-scoring node for pod placement. However, this approach has several significant limitations when it comes to energy efficiency:

1. **Request-Based Resource Allocation:** The scheduler primarily considers CPU and memory requests specified in pod definitions, rather than actual resource usage patterns. This can lead to suboptimal resource utilization when actual consumption differs significantly from requested resources.

2. **No Energy Awareness:** The default scheduler has no built-in mechanisms to consider the energy consumption characteristics of nodes or the power impact of scheduling decisions.

1.1 Research Question

This research focuses on the following question: *"How can we optimize Kubernetes scheduling for energy efficiency while maintaining performance requirements?"*

1.2 Objective

The main goal of this research is to build a custom scheduler plugin based in k8s that will use the actual power metrics of the node,container in the cluster to schedule a pod resource in an optimized way possible

1. Integrate real-time energy metrics collection using Kepler
2. Utilize LoadWatcher for comprehensive resource monitoring
3. Evaluate the energy efficiency improvements compared to the default Kubernetes scheduler
4. Analyze the trade-offs between energy efficiency and performance
5. Utilize the advantage of scalability and faster deployment offered by the cloud services, the k8s cluster is formed using bare metal Ibm cloud virtual machine

1.3 Paper Structure

Section	**Description**
1	Introduction to containerization and Kubernetes. This section also outlines the key research objectives.
2	Discusses the default Kubernetes scheduler, various custom schedulers, and introduces the origin of the proposed scheduler.
3	Covers the research methodology adopted in the study.
5	Describes the design and development process of the custom Kubernetes scheduler.
6	Provides evaluation metrics comparing the proposed custom scheduler with the default scheduler. Discusses how optimized resource consumption can lead to improved power efficiency over time.
7	Summarizes the results, presents comparisons, and offers suggestions for future improvements to the custom Kubernetes scheduler.

2 Related Work

Kubernetes is a open-source orchestration platform tool that transforms how containerized applications are managed in modern computing world. By automating crucial operational aspects including deployment, scaling, and maintenance, Kubernetes enables organizations to focus on application development rather than infrastructure management. Central to its architecture is the scheduling component, which performs the essential function of workload distribution by identifying appropriate nodes within the cluster for task execution, subsequently enabling the kubelet to run the containerized workloads. The distinctive strength of Kubernetes lies in its architectural adaptability, offering users significant flexibility in resource allocation strategies. While the platform provides default scheduling algorithms with predefined evaluation criteria, it simultaneously supports extensive customization options. According to the CNCF Annual Survey shows a 66% increase in Kubernetes production usage among non-cloud businesses in 2023, compared to 2022. This rise reflects growing acceptance of Kubernetes as a critical technology in the evolving tech ecosystem Foundation (2023). Organizations can tailor the scheduling process to their specific operational requirements through multiple approaches: modifying configuration parameters of the default scheduler, developing entirely new custom schedulers, or implementing specialized plugins that extend the core scheduling functionality. This section explores various research efforts and technical approaches aimed at enhancing Kubernetes scheduling capabilities. Through analysis of diverse implementation strategies.

2.1 Kubernetes Survey

Traditional kuberenetes schedulers has several limitations. For instance, it schedules Pods based on the requested resources (like CPU and memory) rather than the actual resource usage or power consumption patterns of nodes. This can lead to suboptimal performance in scenarios where machines remain underutilized, resulting in inefficient resource usage, higher energy consumption, and poor handling of network bandwidth, disk I/O, or increased latency Senjab et al. (2023)

2.2 Custom Scheduling

The evolution of Kubernetes scheduler customization is been an active area of development since its early years 2015-2017 where practitioners exploring ways to enhance its capabilities beyond default configurations. These modifications typically arise to solve the problems that the standard scheduling mechanisms may not sufficiently address certain specialized workload requirements

One significant approach involves incorporating application-specific knowledge into scheduling decisions. Some implementations have demonstrated how client applications can actively communicate their requirements to the scheduler, enabling more nuanced resource allocation strategies. This becomes particularly valuable when considering that container-based isolation is different fundamentally from traditional virtual machine isolation models. By establishing direct communication channels between applications and the scheduler, systems can achieve better resource utilization while minimizing contention scenarios that might otherwise degrade performance Medel et al. (2017).

Network-aware scheduling represents another important advancement in this domain.

Innovative solutions is developed to solve network bandwidth as a primary scheduled resource, compared to traditional compute resources like CPU and memory. Traditional Kubernetes schedulers often neglect network bandwidth problems which can lead to performance problems for applications with intensive data transfer requirements. Modern adaptations address this gap by extending the standard resource specification model to include bandwidth parameters, while maintaining compatibility with Kubernetes 'established Quality of Service (QoS) classification system - including Guaranteed, Burstable, and Best-effort service tiers. This holistic approach ensures that network resources receive the same level of scheduling consideration as other critical system resources Xu et al. (2018).

2.3 Theoretical formulations and heuristic methods

These approaches are mostly used to solve resource allocation mainly focusing on power consumption

Paper	**Methodology**	**Limitations**
Escrig et al. (2024)	Involves a Deep Reinforcement Learning (DRL) agent utilizing Proximal Policy Optimization (PPO), trained with custom Neural Networks, to recommend optimal node allocations for Kubernetes pods.	A significant limitation is the reduced energy-saving effectiveness of the DRL method under cluster saturation [Drop to about 2.08% compared to 16% in less saturated conditions].
Li et al. (2025)	Introduces the Energy-Aware Elastic Scaling (EAES) algorithm for Kubernetes microservices, aiming to reduce energy consumption ,ensure SLA compliance and combines workload prediction with feedback control, notably by releasing excess idle container	include balancing energy efficiency, response time, and service availability.
Dakic et al. (2025)	proposes a custom ML-driven Kubernetes scheduler that uses a neural network to forecast web application scheduling times	introduces increased infrastructure complexity and requires storing large volumes of historical data for ML training.
Piontek et al. (2023)	CO2 -aware workload scheduling algorithm implemented in K8s to reduce carbon emissions	Focuses exclusively on application-layer power measurement, potentially overlooking lower-level power consumption.

2.4 Hardware and software-based approaches

These approach enables precise power measurement and monitoring at different levels of technology stack

Paper	Methodology	Limitations
Schmitt et al. (2019)	Introduces a monitoring and modeling approach to estimate power consumption for cloud applications	Focuses exclusively on application-layer power measurement, potentially overlooking lower-level power consumption
James and Schien (2019)	implements a low-carbon Kubernetes scheduler to dynamically place workloads It leverages real-time carbon intensity data from APIs and local air temperature to select compute nodes	scheduler itself relies on external real-time carbon intensity APIs which are not exhaustively available globally
Townend et al. (2019)	implemented a holistic software scheduler for Kubernetes that integrates detailed hardware and software behavior models to optimize container placement. experimentally validated in a real data center	assumed deterministic and known workloads, while hardware modeling proved complex and time-consuming to fully explore
CNCF kepler conference (2022)	uses software counters to measure power consumption through eBPF-based data collection and aggregation	not possible on virtual machine as uses linear regression for power assumption

2.5 Ml based

The ml based approach has gained popularity for optimization of energy efficiency,these approaches aim to build a model for resource estimation under a specific workload, one of the shout out features are handling fluctuating demands often encountered in modern computing environments. These approaches can adjust their model parameters in real time whenever an change event occurs in contrast offline training may demand manual intervention to fine-tune the model to achieve accuracy. the main drawback of ml-based approaches is the high execution time and causes poor allocation and management during the learning period [Ma and Ding (2022); Shapi et al. (2021)]

2.6 Power Aware Scheduler

the scheduler is extended integrating energy-aware capabilities using Prometheus with the kepler for power metrics during the score phase of the kubernetes scheduling framework. The node with the highest score is selected for task execution *CNCF peaks* (2022).

2.7 Need for the proposed scheduler

Existing research in Kubernetes scheduling and energy efficiency has made significant progress, offering various approaches to optimize resource utilization and reduce environmental impact. However, several limitations and opportunities for further advancement remain. Many studies focus on specific aspects, such as initial scheduling or autoscaling, without providing a holistic solution that integrates real-time power consumption data with dynamic workload characteristics and hardware-specific optimizations. In summary, this work directly addresses this by incorporating precise, hardware-specific power consumption data, allowing for more accurate and effective energy optimization. This moves beyond theoretical models to practical, measurable energy savings specific to the underlying infrastructure contrast to Papers such as "Optimizing Energy Consumption of Kubernetes Clusters with Deep Reinforcement Learning" Ma and Ding (2022) demonstrate impressive energy reductions but acknowledge the challenges of DRL model deployment and training. My approach simplifies this complexity by providing an optimized solution that is readily deployable and requires minimal overhead, making advanced energy-efficient scheduling accessible without extensive ML expertise or resource-intensive training. Another common limitation observed in the reviewed literature, including works like "A Low Carbon Kubernetes Scheduler" James and Schien (2019) and "Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes," Townend et al. (2019) is the focus on carbon intensity or general data center efficiency without deeply integrating with the dynamic nature of Kubernetes workloads and the need for continuous adaptation. My work offers a solution that is inherently dynamic and adaptive, continuously optimizing scheduling decisions basedon real-time workload demands and energy metrics, ensuring sustained efficiency in ever-changing cloud environments. Moreover, while surveys like "A survey of Kubernetes scheduling algorithms" Senjab et al. (2023)identify gaps in research, they do not provide concrete solutions. My contribution fills these gaps by offering a practical, optimized scheduling solution that considers both performance and energy efficiency, In summary, my work builds upon the strengths of existing research by leveraging real-time, hardware-specific energy metrics and providing a highly optimized and easily deployable solution contributing significantly to the field of sustainable cloud computing.

3 Methodology

3.1 Research Approach

This research employs a design methodology to develop and evaluate a custom Kubernetes scheduler focusing on maximizing energy efficiency. This approach involves systematically designing, implementing and testing a novel scheduling solution that integrates real-time power metrics with parameter-optimized models to make energy-aware placement decisions. The methodology follows a structured process of problem identification,solution design, implementation and evaluation against established benchmarks. This research focuses on energy-efficient computing and practical implementation within Kubernetes environments. By leveraging existing frameworks like Kepler and PEAKS and integrating with tools such as Prometheus and Kepler. This research aims to create a practical solution that addresses the real-world challenge of energy consumption in containerized environments.

3.2 Components of the Proposed Custom Scheduler

The custom scheduler architecture consists of several interconnected components that work together to enable energy-efficient pod scheduling in Kubernetes. Each component serves as a specific function within the overall system.

3.2.1 Kubernetes Extension Points

The custom scheduler leverages the Kubernetes scheduling framework's extension points, particularly focusing on the scoring phase as illustrated in the figure 2. the scheduler integrates with the following extension points:

- **PreFilter** – Performs preliminary checks before the filtering phase.
- **Filter** – Identifies feasible nodes that meet the pod's requirements.
- **PreScore** – Prepares data needed for the scoring phase.
- **Score** – Assigns scores to nodes based on energy-efficiency metrics.
- **Normalize Score** – Normalizes scores across different scoring plugins.
- **Reserve** – Reserves resources on the selected node.

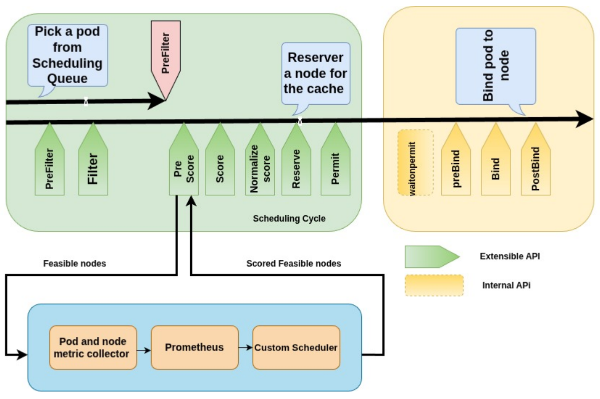


Figure 1: kubernetes extension points

The above extension points allow the custom scheduler to integrate easily with the existing Kubernetes architecture while adding energy-aware decision making capabilities

3.2.2 Power Metrics Collection Layer

A critical component of the custom scheduler is the power metrics collection layer, which gathers real-time energy consumption data from cluster nodes. This layer consists of

- **Kepler(kubernetes-based Efficient Power Level Exporter)**: Collects power consumption metrics at both node and container levels using hardware counters
- **LoadWatcher**: Monitors real-time resource utilization across the cluster
- **Prometheus**: Stores and aggregates time-series metrics data
- **Node Exporter**: Deploys on each worker node to expose hardware and os metrics

The metrics collection layer provides essential data about the energy profile of each node enabling the scheduler to make informed decisions about pod placement based on actual power consumption patterns rather than just CPU and memory requests

3.2.3 Power Model

The power model component translates collected metrics into actionable scoring inputs, it implements:

- **Parameter Optimization**: Trained parameters based on historical workload patterns
- **Resource Weighting**: Assigns different weights to various resource types(CPU, memory,I/O) based on their impact on energy consumption

3.2.4 Score Calculator

The score calculator component evaluates nodes based on their energy efficiency

- **Energy Efficiency Score**: Calculates a score for each node based on predicted
- **Multi-factor Evaluation**: Considers current node utilization, pod requirements and power model prediction
- **Score Normalization**: Ensures scores are comparable across different nodes

  The score calculator produces a final ranking of nodes that prioritizes energy efficiency while still meeting the performance requirements of the workload

3.2.5 API Server Integration

The custom scheduler integrates with the Kubernetes API server to:

- **Receive Pod Scheduling Requests**: gives scheduling decisions for newly created pods
- **Access Node Information**: Retrieves current state and metrics for all available nodes
- **Bind Decisions**: Communicates final scheduling decisions back to the API server

  This integration enables that the custom scheduler operates within the standard Kubernetes control flow while adding energy-aware decision capabilities.

3.3 Functioning of Custom Scheduler

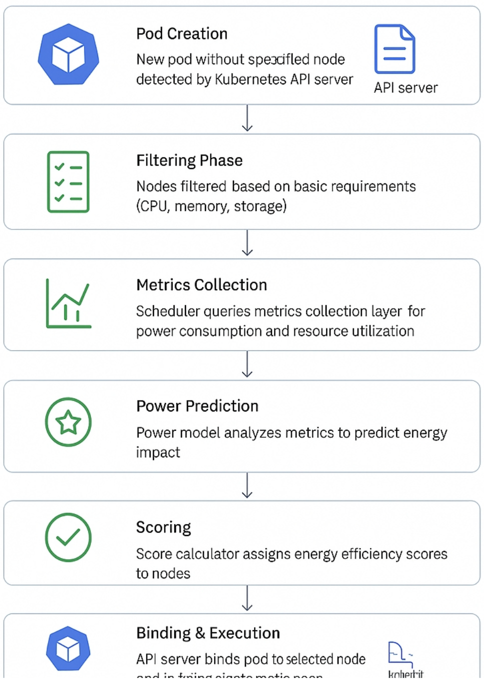
The custom scheduler operates through a series of steps that transform a pod scheduling request into an energy efficient placement decision

3.3.1 Pod Scheduling Workflow

3.4 Energy Decision Making

The core of the custom scheduler is energy aware decision making process which considers:

- **Current Node Power Consumption**: Prioritizes the nodes that are operating at energy efficient utilization levels
- **Cluster-wide Energy Optimization**: Aims to minimize the total energy consumption across the entire cluster



Pod Creation
New pod without specified node detected by Kubernetes API server
API server
Filtering Phase
Nodes filtered based on basic requirements (CPU, memory, storage)
Metrics Collection
Scheduler queries metrics collection layer for power consumption and resource utilization
Power Prediction
Power model analyzes metrics to predict energy impact
Scoring
Score calculator assigns energy efficiency scores to nodes
Binding & Execution
API server binds pod to selected node

Figure 2: Custom Scheduler Flow

4 Design Specification

The custom scheduler is designed to operate on baremetal servers and IBM Cloud environment as multi node cluster approach using kubeadm or kops industry standard kubernetes clusters. The design specification outlines how the scheduler integrates with the baremetal servers and IBM cloud architecture.

4.1 IBM Cloud Integration

As illustrated in the IBM Cloud architecture diagram 3, the custom scheduler operates within the control plane along with the Kubernetes default scheduler, key integrations include

- **VPC Network Integration**: Operates within IBM's Cloud virtual private cloud network
- **Bare Metal Node Support**: uses IBM cloud's bare metal offerings, where direct hardware metrics provide the most accurate power consumption data

  The scheduler is designed to be compatible with any cloud provider [aws,azure,ibm] with bare metals server to ensure security and compliance features ensuring that energy efficient scheduling doesn't compromise the security.

5 Implementation

This Section explains the technical implementation of custom power efficient Kubernetes scheduler by explaining the step-by-step process of building and integration of various components to achieve energy-aware pod placement.

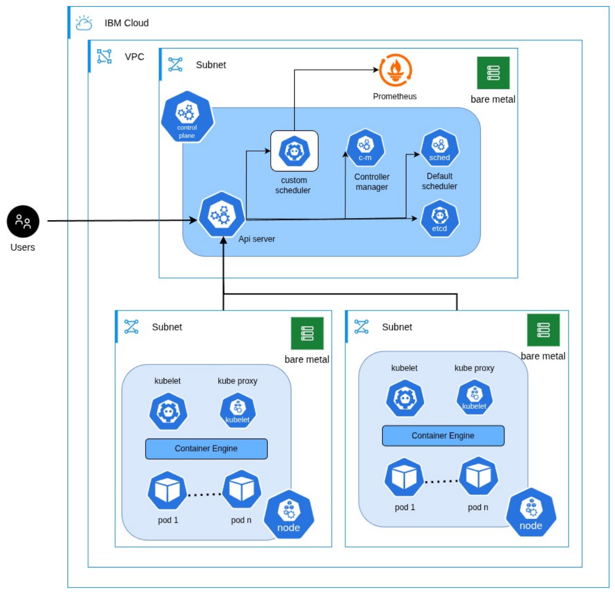


Figure 3: IBM cloud Architecture

5.1 Tools and Programming languages

The Custom Scheduler was developed using a set of tools and technologies aligned with the Kubernetes ecosystem to ensure compatibility, performance and ease of integration.

5.1.1 Programming language

Go (Golang) was choosen as primary programming language for the custom scheduler as it support

- **Native kubernetes Integration**
- **Performance**
- **Concurrency**

5.1.2 Development Tools and Frameworks

- **Kubernetes Client Libraries:** The Official Go client libraries for Kubernetes (client-go) used to interact with the Kubernetes API server which enables the custom scheduler to watch for new pods, retrieve node information and bind pods to selected nodes
- **Prometheus Client Libraries:** Go Client libraries for Prometheus were used to query and interact with the Prometheus metrics server allowing the scheduler to retrieve the real-time power consumption and resource utilization data

5.1.3 Testing Environment

Initial development and testing were performed on local Kubernetes clusters included:

- **Kubeadm:** For running a multi-node Kubernetes cluster locally for quick tests and demonstration

For more testing and performance evaluation, the scheduler was deployed on a multi node production grade Kubernetes cluster using kubeadm provisioned on IBM Cloud bare metal Servers as detailed in 3 chapter. This environment allowed for accurate power measurements and realistic workload testing

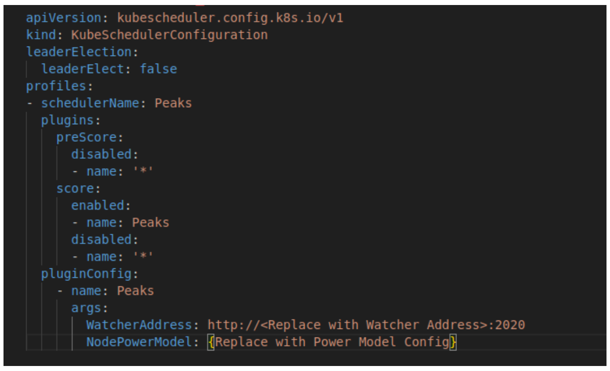
5.2 Core Scheduler Logic Implementation

The Custom Scheduler is implemented as a standalone process which can interacts with the Kubernetes API server which leverages the scheduler framework extension. The core logic follows the standard Kubernetes scheduling lifecycle and also with energy-aware decision making.

5.2.1 Kubernetes Scheduler Framework Integration

The implementation aligns to the Kubernetes scheduler framework which provides a pluggable architecture for custom scheduling logic. the steps are defined as follows:

1. **Registering the Custom Scheduler:** The Custom Scheduler is registered with kubertnes by defining a Scheduler object and configuring the kube scheduler component to recognize and use it. specifying the scheduler name in the pod's **spec.schedulerName** field



```yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
leaderElection:
  leaderElect: false
profiles:
- schedulerName: Peaks
  plugins:
    preScore:
      disabled:
      - name: '*'
    score:
      enabled:
      - name: Peaks
      disabled:
      - name: '*'
  pluginConfig:
    - name: Peaks
      args:
        WatcherAddress: http://<Replace with Watcher Address>:2020
        NodePowerModel: {Replace with Power Model Config}
```

Figure 4: Custom Scheduler Configuation

2. **Implementing Plugins:** The custom plugins that implement specific functionality defined by the scheduler frmework of kubernetes. These plugins are invoked at different stages of scheduling cycle:

   - **PreFilter Plugin:** Performs Initial checks on pods and nodes before filtering, ensure power metrics are available for a node
   - **Filter Plugin:** Filters out nodes that do not meet the Pod's basic requirements (e.g., resource requests, node selectors)
   - **PreScore Plugin:** Prepares data or state for subsequent scoring phase , where the scheduler fetch the latest power metrics from Prometheus
   - **Score Plugin:** This is critical plugin for power efficiency, it assigns the numerical score to each feasible node, with higher scores indicate better power efficiency for placing the current pods. The scoring logic uses the power model and real time metrics

- **Normalize Score Plugin:** Normalizes the score produced by different scoring plugins to ensure fair suggestion

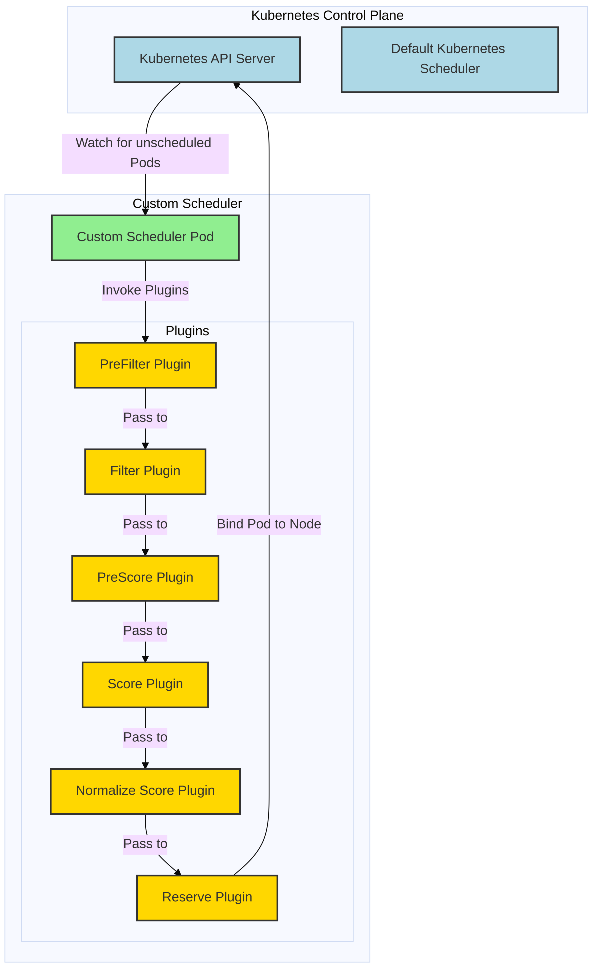


Figure 5: Custom Scheduler Framework Cycle

5.3 Metrics Integration

Accurate and real-time power and resource metrics are crucial to power-efficient custom scheduler decisions. The implementation integrates with Prometheus, kepler and loadwatcher to collect and expose these metrics.

5.3.1 Prometheus Setup

it servers as the central time-series database for all collected metrics. the steps involved:

- **Prometheus Server Deployment:** A Prometheus is deployed within the Kubernetes cluster configured to scrape metrics from various targets
- **Configuration For Scrapping:** configuration file should update to include scrape targets for:
  - **Node Exporter:** Deployed on each worker node to expose host-level metrics [CPU,memory,disk,I/O,network]

  - **Kepler Exporter:** Configured to Scrape power metrics exposed by Kepler
  - **kube-state-metrics:** To get metrics about the state of Kubernetes (pods,nodes,deployment

5.3.2 Kepler Integration

Kepler (Kubernetes-based Efficient Power Level Exporter) is important for obtaining fine-grained power consumption data

- **Deployment as DaemonSet:** Kepler was deployed as a DaemonSet in all worker nodes in kubernetes cluster. This ensures that kepler agent runs on every node collecting power metrics
- **Power Measurement Sources:** it is configured to utilize available hardware counters (e.g., Intel RAPL) for direct power measurements are supported
- **Metrics Exposure:** Kepler exposes power metrics in Prometheus format(e.g.,kepler_container_jou , kepler_node_joules_total) which are scraped by Prometheus server

5.4 LoadWatcher Integration

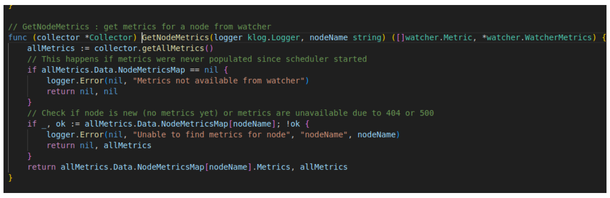
Loadwatcher provides real-time resource utilization metrics utilizing kepler power data

- **Deployment:** Loadwatcher agents are deployed on worker nodes to monitor CPU,memory and other resource usage at granular level.
- **Metrics Reporting:** it reports resource utilization data which can also be exposed by Prometheus and directly accessible by the custom scheduler by an API

5.4.1 Flow of the Scheduler

The Custom Scheduler retrieves metrics from Prometheus using its client libraries.

- **PromQL Queries:** The Scheduler Constructs PromQL (Prometheus Query Language) queries to fetch specific metrics
- **API calls:**Direct API calls to loadwatcher are made to retrieve the queried data which later then used by the power model and scoring logic



```go
// GetNodeMetrics : get metrics for a node from watcher
func (collector *Collector) GetNodeMetrics(logger klog.Logger, nodeName string) ([]watcher.Metric, *watcher.WatcherMetrics) {
	allMetrics := collector.getAllMetrics()
	// This happens if metrics were never populated since scheduler started
	if allMetrics.Data.NodeMetricsMap == nil {
		logger.Error(nil, "Metrics not available from watcher")
		return nil, nil
	}
	// Check if node is new (no metrics yet) or metrics are unavailable due to 404 or 500
	if _, ok := allMetrics.Data.NodeMetricsMap[nodeName]; !ok {
		logger.Error(nil, "Unable to find metrics for node", "nodeName", nodeName)
		return nil, allMetrics
	}
	return allMetrics.Data.NodeMetricsMap[nodeName].Metrics, allMetrics
}
```

Figure 6: Fetching metrics from loadwatcher

5.5 Power model and Scoring Implementation

The power efficient scheduler lies in its power model and the scoring logic that utilizes the power model to rank nodes

5.5.1 Power Model Implementation

The power model estimates the change in power consumption resulting from scheduling a new pod on a node. While the theoretical model is based on an exponential power function:

$$P(x) = K_0 + K_1 \cdot e^{K_2 \cdot x} \tag{1}$$

where:

- x is the CPU utilization (between 0 and 100),
- K_0 represents the constant baseline (idle) power consumption,
- K_1 is a scaling factor,
- K_2 controls the exponential growth of power with respect to utilization.

This model reflects the nonlinear relationship between power consumption and CPU load.

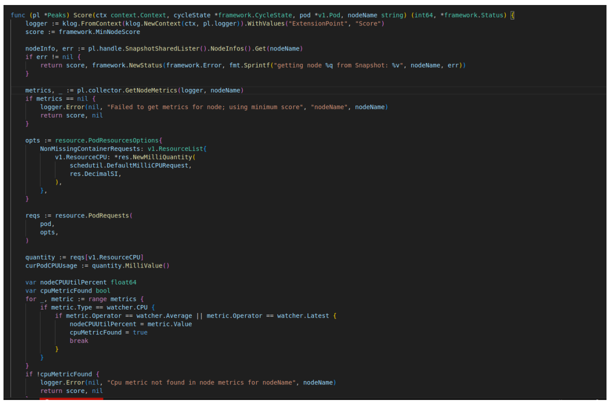
- **Scoring Implementation** In practice, the scheduler does not compute absolute power values for nodes. Instead, it uses the **change in power** caused by scheduling a pod, calculated as:

  $$\Delta P = K_1 \cdot \left(e^{K_2 \cdot U_{pred}} - e^{K_2 \cdot U_{curr}}\right) \tag{2}$$

  Where:

  - U_{curr} is the current CPU utilization of the node,
  - U_{pred} is the predicted utilization after scheduling the pod,
  - ΔP is the estimated increase in power consumption.

  This differential approach allows the scheduler to compare nodes based on their relative increase in power, without needing the constant K_0, which does not affect scheduling decisions. The node with the lowest predicted power jump (ΔP) is preferred, assuming all other scheduling criteria are met.



```go
func (pl *Peaks) Score(ctx context.Context, cycleState *framework.CycleState, pod *v1.Pod, nodeName string) (int64, *framework.Status) {
    logger := klog.FromContext(klog.NewContext(ctx, pl.logger)).WithValues("ExtensionPoint", "Score")
    score := framework.MinNodeScore

    nodeInfo, err := pl.handle.SnapshotSharedLister().NodeInfos().Get(nodeName)
    if err != nil {
        return score, framework.NewStatus(framework.Error, fmt.Sprintf("getting node %q from Snapshot: %v", nodeName, err))
    }

    metrics, _ := pl.collector.GetNodeMetrics(logger, nodeName)
    if metrics == nil {
        logger.Error(nil, "Failed to get metrics for node; using minimum score", "nodeName", nodeName)
        return score, nil
    }

    opts := resource.PodResourcesOptions{
        NonMissingContainerRequests: v1.ResourceList{
            v1.ResourceCPU: *res.NewMilliQuantity(
                schedutil.DefaultMilliCPURequest,
                res.DecimalSI,
            ),
        },
    }

    reqs := resource.PodRequests(
        pod,
        opts,
    )

    quantity := reqs[v1.ResourceCPU]
    curPodCPUUsage := quantity.MilliValue()

    var nodeCPUUtilPercent float64
    var cpuMetricFound bool
    for _, metric := range metrics {
        if metric.Type == watcher.CPU {
            if metric.Operator == watcher.Average || metric.Operator == watcher.Latest {
                nodeCPUUtilPercent = metric.Value
                cpuMetricFound = true
                break
            }
        }
    }
    if !cpuMetricFound {
        logger.Error(nil, "Cpu metric not found in node metrics for nodeName", nodeName)
        return score, nil
```

Figure 7: scoring implementation

- **Parameter Training** The parameters K_1 and K_2 were derived through offline regression using data collected from bare-metal servers. The process involved recording CPU utilization and power consumption across different load levels and fitting the exponential model to the data.
- **Dynamic Parameter Adjustment** While the initial model parameters are statically configured, the implementation supports periodic retraining. This allows to update the model coefficients as hardware characteristics or workload patterns change, improving accuracy and maintaining optimal power-aware scheduling over time.

6 Evaluation

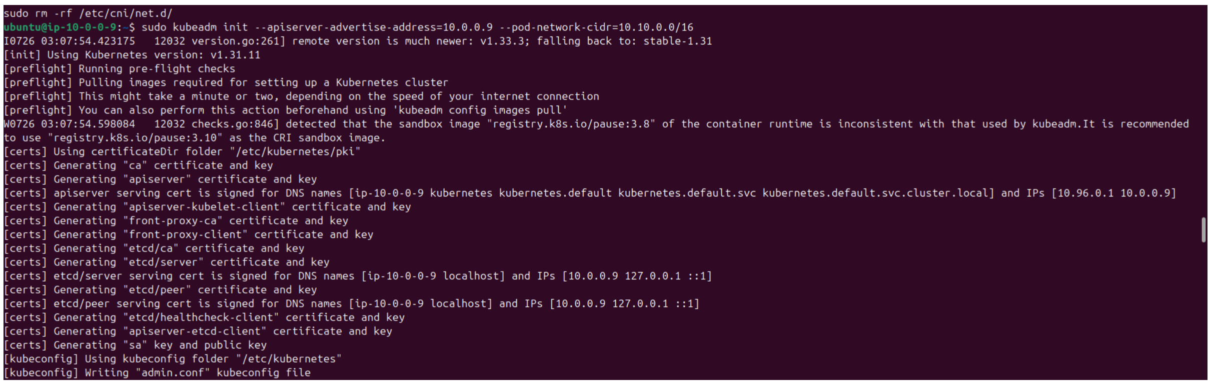
This section gives information about the experimental evaluation of the custom Kubernetes focusing on energy efficiency and resource utilization benefits compared to default Kubernetes scheduler

6.1 Experimental Setup

The setup provides the real-world Kubernetes workloads and measure the energy consumption and resource utilization of nodes under different scheduling strategies. The evaluation is performed on kubernetes cluster configured with heterogeneous bare metals servers using kubeadm and monitored with grafana and prometheus

6.1.1 cluster configuration

The kubernetes cluster creation is done using kubeadm for experimental deployment. The cluster consists of two heterogeneous bare-metal worker nodes to replicate like a production grade cluster with varying hardware capabilities



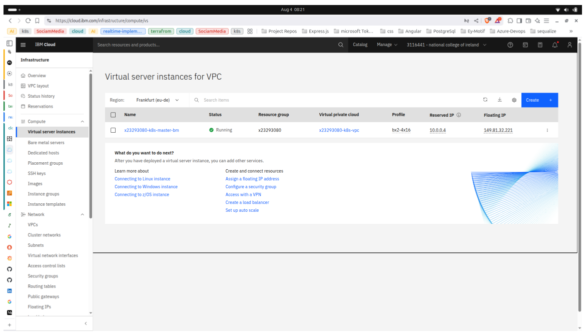
```
sudo rm -rf /etc/cni/net.d/
ubuntu@ip-10-0-0-9:~$ sudo kubeadm init --apiserver-advertise-address=10.0.0.9 --pod-network-cidr=10.10.0.0/16
I0726 03:07:54.423175   12032 version.go:261] remote version is much newer: v1.33.3; falling back to: stable-1.31
[init] Using Kubernetes version: v1.31.11
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action beforehand using 'kubeadm config images pull'
W0726 03:07:54.598084   12032 checks.go:846] detected that the sandbox image "registry.k8s.io/pause:3.8" of the container runtime is inconsistent with that used by kubeadm.It is recommended
to use "registry.k8s.io/pause:3.10" as the CRI sandbox image.
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [ip-10-0-0-9 kubernetes kubernetes.default kubernetes.default.svc kubernetes.default.svc.cluster.local] and IPs [10.96.0.1 10.0.0.9]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [ip-10-0-0-9 localhost] and IPs [10.0.0.9 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [ip-10-0-0-9 localhost] and IPs [10.0.0.9 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
```

Figure 8: Kubeadm Cluster Initialization

Figure 8 show the successful initialization of the Kubernetes cluster using Kubeadm. This command-line output confirms the setup of essential Kubernetes components, including certificates, API server, controller manager, and scheduler

- **Node1**: 16vcpu, 128GB RAM

- **Node2**: 16vcpu, 256GB RAM



Virtual server instances for VPC

Region: Frankfurt (eu-de)

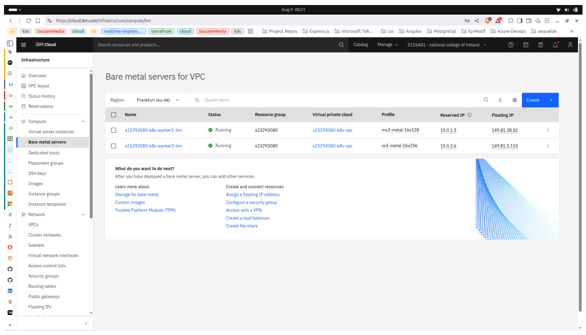
Name	Status	Resource group	Virtual private cloud	Profile	Reserved IP	Floating IP
x23293080-k8s-master-bm	Running	x23293080	x23293080-k8s-vpc	bx2-4x16	10.0.0.4	149.81.32.221

What do you want to do next?
After you have deployed a virtual server instance, you can add other services.

Learn more about	Create and connect resources
Connecting to Linux instance	Assign a floating IP address
Connecting to Windows instance	Configure a security group
Connecting to z/OS instance	Access with a VPN
	Create a load balancer
	Set up auto scale

Aug 4 08:21
https://cloud.ibm.com/infrastructure/compute/vs
IBM Cloud
Search resources and products...
Catalog
Manage
3116441 - national college of ireland
Create

(a) Master Node



Bare metal servers for VPC

Region: Frankfurt (eu-de)

Name	Status	Resource group	Virtual private cloud	Profile	Reserved IP	Floating IP
x23293080-k8s-worker1-bm	Running	x23293080	x23293080-k8s-vpc	mx3-metal-16x128	10.0.1.5	149.81.38.82
x23293080-k8s-worker2-bm	Running	x23293080	x23293080-k8s-vpc	vx3-metal-16x256	10.0.2.6	149.81.5.153

What do you want to do next?
After you have deployed a bare metal server, you can add other services.

Learn more about	Create and connect resources
Storage for bare metal	Assign a floating IP address
Custom images	Configure a security group
Trusted Platform Module (TPM)	Access with a VPN
	Create a load balancer
	Create file share

Aug 4 08:21
https://cloud.ibm.com/infrastructure/compute/bm
IBM Cloud
Search resources and products...
Catalog
Manage
3116441 - national college of ireland
Create

(b) Worker Nodes

Figure 9: Overview of the Master and Worker nodes on Cloud

This setup is important for evaluating the scheduler performance under real-time with varied workload conditions where optimal placement on different node types can significantly impact energy efficiency and resource utilization

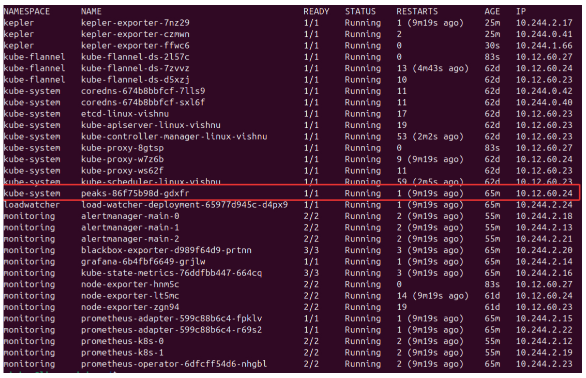
6.1.2 Workload Configuration

workloads are generated using **stress-ng** to create CPU-bound tasks allowing to control over resource demands . This experiment involved both continuous deployments and burstable jobs to test the schedulers under different load patterns. The YAMl configuration were designed for these workloads for comparative analysis

- **Default Scheduler Workloads:** For the default scheduler various job configurations are used to simulate increasing cpu loads , a regular deployment is used for continuous workloads and a Horizontal Pod AutoScaler(HPA) is configured to dynamically scale pods based on CPU utilization which represents **real world fluctuating demands**
- **Custom Scheduler Workloads:** For the Custom scheduler similar various job configurations are used to simulate increasing cpu loads , a regular deployment is used for continuous workloads and a Horizontal Pod AutoScaler(HPA) is configured to dynamically scale pods to evaluate the schedulers ability to manage dynamic scaling while **optimizing the energy efficiency**

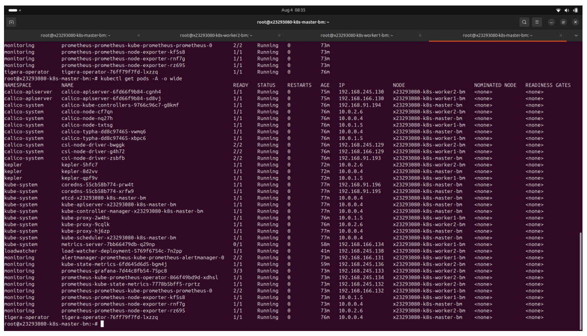
6.2 Monitoring and Metrics Collection

monitoring is required for accurate evaluation. **Prometheus** was deployed to collect real-time metrics from the cluster and nodes including cpu,memory and power consumption data from tools like kepler. Grafan dashboards are configured to visualize these metrics providing insights into node performance energy consumption trends and workload distribution . this setup helped for detailed analysis of the scheduler's behavior and their role on heterogeneous cluster.



```
NAMESPACE      NAME                                       READY   STATUS    RESTARTS          AGE   IP
kepler         kepler-exporter-7nz29                      1/1     Running   1 (9m19s ago)     25m   10.244.2.17
kepler         kepler-exporter-czmwn                      1/1     Running   2                 25m   10.244.0.41
kepler         kepler-exporter-ffwc6                      1/1     Running   0                 30s   10.244.1.66
kube-flannel   kube-flannel-ds-2l57c                      1/1     Running   0                 83s   10.12.60.27
kube-flannel   kube-flannel-ds-7zvvz                      1/1     Running   13 (4m43s ago)    62d   10.12.60.24
kube-flannel   kube-flannel-ds-d5xzj                      1/1     Running   10                62d   10.12.60.23
kube-system    coredns-674b8bbfcf-7lls9                   1/1     Running   11                62d   10.244.0.42
kube-system    coredns-674b8bbfcf-sxl6f                   1/1     Running   11                62d   10.244.0.40
kube-system    etcd-linux-vishnu                          1/1     Running   17                62d   10.12.60.23
kube-system    kube-apiserver-linux-vishnu                1/1     Running   19                62d   10.12.60.23
kube-system    kube-controller-manager-linux-vishnu       1/1     Running   53 (2m2s ago)     62d   10.12.60.23
kube-system    kube-proxy-8gtsp                           1/1     Running   0                 83s   10.12.60.27
kube-system    kube-proxy-w7z6b                           1/1     Running   9 (9m19s ago)     62d   10.12.60.24
kube-system    kube-proxy-ws62f                           1/1     Running   11                62d   10.12.60.23
kube-system    kube-scheduler-linux-vishnu                1/1     Running   59 (2m5s ago)     62d   10.12.60.23
kube-system    peaks-86f75b98d-gdxfr                      1/1     Running   1 (9m19s ago)     65m   10.12.60.24
loadwatcher    load-watcher-deployment-65977d945c-d4px9   1/1     Running   1 (9m19s ago)     65m   10.244.2.24
monitoring     alertmanager-main-0                        2/2     Running   2 (9m19s ago)     55m   10.244.2.18
monitoring     alertmanager-main-1                        2/2     Running   2 (9m19s ago)     55m   10.244.2.13
monitoring     alertmanager-main-2                        2/2     Running   2 (9m19s ago)     55m   10.244.2.21
monitoring     blackbox-exporter-d989f64d9-prtnn          3/3     Running   3 (9m19s ago)     65m   10.244.2.20
monitoring     grafana-6b4fbf6649-grjlw                   1/1     Running   1 (9m19s ago)     65m   10.244.2.14
monitoring     kube-state-metrics-76ddfbb447-664cq        3/3     Running   3 (9m19s ago)     65m   10.244.2.16
monitoring     node-exporter-hnm5c                        2/2     Running   0                 83s   10.12.60.27
monitoring     node-exporter-lt5mc                        2/2     Running   14 (9m19s ago)    61d   10.12.60.24
monitoring     node-exporter-zgn94                        2/2     Running   19                61d   10.12.60.23
monitoring     prometheus-adapter-599c88b6c4-fpklv        1/1     Running   1 (9m19s ago)     65m   10.244.2.15
monitoring     prometheus-adapter-599c88b6c4-r69s2        1/1     Running   1 (9m19s ago)     65m   10.244.2.22
monitoring     prometheus-k8s-0                           2/2     Running   2 (9m19s ago)     55m   10.244.2.12
monitoring     prometheus-k8s-1                           2/2     Running   2 (9m19s ago)     55m   10.244.2.19
monitoring     prometheus-operator-6dfcff54d6-nhgbl       2/2     Running   2 (9m19s ago)     65m   10.244.2.23
```

(a) Custom Scheduler Deployed



Aug 4 08:35
root@x23293080-k8s-master-bm: ~
root@x23293080-k8s-master-bm: ~
root@x23293080-k8s-worker2-bm: ~
root@x23293080-k8s-worker1-bm: ~
root@x23293080-k8s-master-bm: ~

```
monitoring         prometheus-prometheus-kube-prometheus-prometheus-0     2/2     Running   0          73m
monitoring         prometheus-prometheus-node-exporter-kf5s8              1/1     Running   0          73m
monitoring         prometheus-prometheus-node-exporter-rnf7g              1/1     Running   0          73m
monitoring         prometheus-prometheus-node-exporter-rz695              1/1     Running   0          73m
tigera-operator    tigera-operator-76ff79f7fd-lxzzq                       1/1     Running   0          76m
root@x23293080-k8s-master-bm:~# kubectl get pods -A -o wide
NAMESPACE          NAME                                                   READY   STATUS    RESTARTS   AGE   IP                NODE                        NOMINATED NODE   READINESS GATES
calico-apiserver   calico-apiserver-6fd66f9b84-cgnh4                      1/1     Running   0          75m   192.168.245.130   x23293080-k8s-worker2-bm    <none>           <none>
calico-apiserver   calico-apiserver-6fd66f9b84-sd8vj                      1/1     Running   0          75m   192.168.166.130   x23293080-k8s-worker1-bm    <none>           <none>
calico-system      calico-kube-controllers-9766c96c7-g8kmf                1/1     Running   0          76m   192.168.91.194    x23293080-k8s-master-bm     <none>           <none>
calico-system      calico-node-cf7qn                                      1/1     Running   0          76m   10.0.2.6          x23293080-k8s-worker2-bm    <none>           <none>
calico-system      calico-node-nq27h                                      1/1     Running   0          76m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
calico-system      calico-node-txtsg                                      1/1     Running   0          76m   10.0.1.5          x23293080-k8s-worker1-bm    <none>           <none>
calico-system      calico-typha-dd8c97465-vwmq6                           1/1     Running   0          76m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
calico-system      calico-typha-dd8c97465-xbpc6                           1/1     Running   0          76m   10.0.1.5          x23293080-k8s-worker1-bm    <none>           <none>
calico-system      csi-node-driver-bwggk                                  2/2     Running   0          76m   192.168.245.129   x23293080-k8s-worker2-bm    <none>           <none>
calico-system      csi-node-driver-g4h72                                  2/2     Running   0          76m   192.168.166.129   x23293080-k8s-worker1-bm    <none>           <none>
calico-system      csi-node-driver-zsbfb                                  2/2     Running   0          76m   192.168.91.193    x23293080-k8s-master-bm     <none>           <none>
kepler             kepler-5hfc7                                           1/1     Running   0          72m   10.0.2.6          x23293080-k8s-worker2-bm    <none>           <none>
kepler             kepler-8d2vv                                           1/1     Running   0          72m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
kepler             kepler-qpf9v                                           1/1     Running   0          72m   10.0.1.5          x23293080-k8s-worker1-bm    <none>           <none>
kube-system        coredns-55cb58b774-prw4t                               1/1     Running   0          77m   192.168.91.196    x23293080-k8s-master-bm     <none>           <none>
kube-system        coredns-55cb58b774-xrfw9                               1/1     Running   0          77m   192.168.91.195    x23293080-k8s-master-bm     <none>           <none>
kube-system        etcd-x23293080-k8s-master-bm                           1/1     Running   0          77m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
kube-system        kube-apiserver-x23293080-k8s-master-bm                 1/1     Running   0          77m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
kube-system        kube-controller-manager-x23293080-k8s-master-bm        1/1     Running   0          77m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
kube-system        kube-proxy-2w4hs                                       1/1     Running   0          76m   10.0.1.5          x23293080-k8s-worker1-bm    <none>           <none>
kube-system        kube-proxy-9cqlk                                       1/1     Running   0          76m   10.0.2.6          x23293080-k8s-worker2-bm    <none>           <none>
kube-system        kube-proxy-hj6zp                                       1/1     Running   0          77m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
kube-system        kube-scheduler-x23293080-k8s-master-bm                 1/1     Running   0          77m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
kube-system        metrics-server-7bb66479db-q29np                        0/1     Running   0          58m   192.168.166.134   x23293080-k8s-worker1-bm    <none>           <none>
loadwatcher        load-watcher-deployment-5769f6754c-7n2pp               1/1     Running   0          41m   192.168.245.138   x23293080-k8s-worker2-bm    <none>           <none>
monitoring         alertmanager-prometheus-kube-prometheus-alertmanager-0 2/2     Running   0          73m   192.168.166.131   x23293080-k8s-worker1-bm    <none>           <none>
monitoring         kube-state-metrics-6fd645d6d5-bgm4j                    1/1     Running   0          59m   192.168.245.136   x23293080-k8s-worker2-bm    <none>           <none>
monitoring         prometheus-grafana-7d44c8fb54-75pc8                    3/3     Running   0          73m   192.168.245.133   x23293080-k8s-worker2-bm    <none>           <none>
monitoring         prometheus-kube-prometheus-operator-866f49bd9d-xdhsl   1/1     Running   0          73m   192.168.245.134   x23293080-k8s-worker2-bm    <none>           <none>
monitoring         prometheus-kube-state-metrics-7778b5bff5-rprtz         1/1     Running   0          73m   192.168.245.132   x23293080-k8s-worker2-bm    <none>           <none>
monitoring         prometheus-prometheus-kube-prometheus-prometheus-0     2/2     Running   0          73m   192.168.166.132   x23293080-k8s-worker1-bm    <none>           <none>
monitoring         prometheus-prometheus-node-exporter-kf5s8              1/1     Running   0          73m   10.0.1.5          x23293080-k8s-worker1-bm    <none>           <none>
monitoring         prometheus-prometheus-node-exporter-rnf7g              1/1     Running   0          73m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
monitoring         prometheus-prometheus-node-exporter-rz695              1/1     Running   0          73m   10.0.2.6          x23293080-k8s-worker2-bm    <none>           <none>
tigera-operator    tigera-operator-76ff79f7fd-lxzzq                       1/1     Running   0          76m   10.0.0.4          x23293080-k8s-master-bm     <none>           <none>
root@x23293080-k8s-master-bm:~#
```

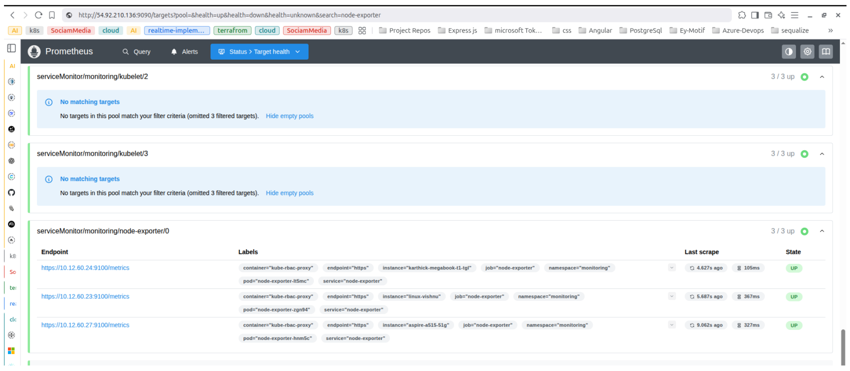
(b) All Components Required For Custom Scheduler

Figure 10: Custom Scheduler Monitoring Components

Figure 10a and 10b provides a snapshot of the running pods within the Kubernetes cluster, obtained via kubectl get pods -A -o wide . This view confirms the operational status of core Kubernetes system components (kube-system) and custom scheudler which we implemented is running on kube-system namespace. the Kepler energy monitoring agents, and various pods related to the Prometheus and Grafana monitoring stack shows a fully functional and observable cluster

6.3 Evaluation Methodology

The methodology has carried out by running both the default and custom scheduler with defined workloads .Data collected from the Prometheus and visualization in grafana was used to analyze to compare energy savings resource utilization patterns. This approach aligns with the best practices for evaluating scheduler redhat (2023)



serviceMonitor/monitoring/kubelet/2 — 3 / 3 up

No matching targets
No targets in this pool match your filter criteria (omitted 3 filtered targets). Hide empty pools

serviceMonitor/monitoring/kubelet/3 — 3 / 3 up

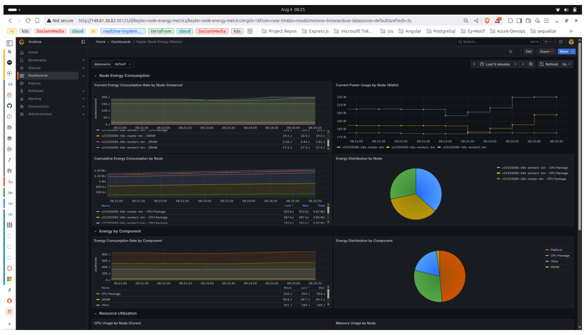
No matching targets
No targets in this pool match your filter criteria (omitted 3 filtered targets). Hide empty pools

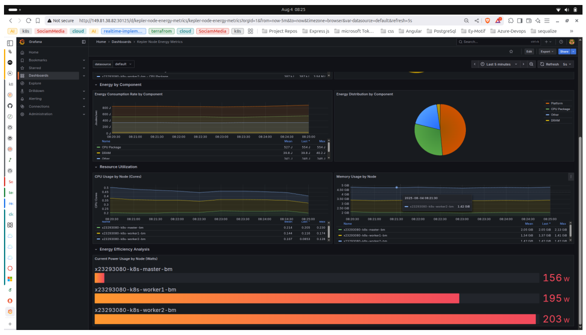
serviceMonitor/monitoring/node-exporter/0 — 3 / 3 up

Endpoint	Labels	Last scrape	State
https://10.12.60.24:9100/metrics	container="kube-rbac-proxy" endpoint="https" instance="karthick-megabook-t1-tgl" job="node-exporter" namespace="monitoring" pod="node-exporter-lt5mc" service="node-exporter"	4.627s ago 105ms	UP
https://10.12.60.23:9100/metrics	container="kube-rbac-proxy" endpoint="https" instance="linux-vishnu" job="node-exporter" namespace="monitoring" pod="node-exporter-zgn94" service="node-exporter"	5.687s ago 367ms	UP
https://10.12.60.27:9100/metrics	container="kube-rbac-proxy" endpoint="https" instance="aspire-a515-51g" job="node-exporter" namespace="monitoring" pod="node-exporter-hnm5c" service="node-exporter"	9.062s ago 327ms	UP

Figure 11: Prometheus Target Health

Figure 11 displays the target health status within Prometheus, indicating that Prometheus is successfully scraping metrics from various endpoints, including Kubernetes components and node exporters. This ensures that real-time performance and energy consumption data are continuously collected for analysis


(a) Energy Consumption Metrics



(b) Power Consumption Metrics

Figure 12: Grafana Dashboard for Node Energy Metrics

Figure 12a and 12b showcases a custom Grafana dashboard configured to visualize node energy consumption and other key metrics. This dashboard provides a clear, real-time overview of the cluster's energy profile, allowing for detailed analysis of power usage by node and component, and enabling the comparative study between schedulers

6.4 Comparative Analysis and Results

This section outlines the detailed results of energy efficiency and resource utilization of custom scheduler

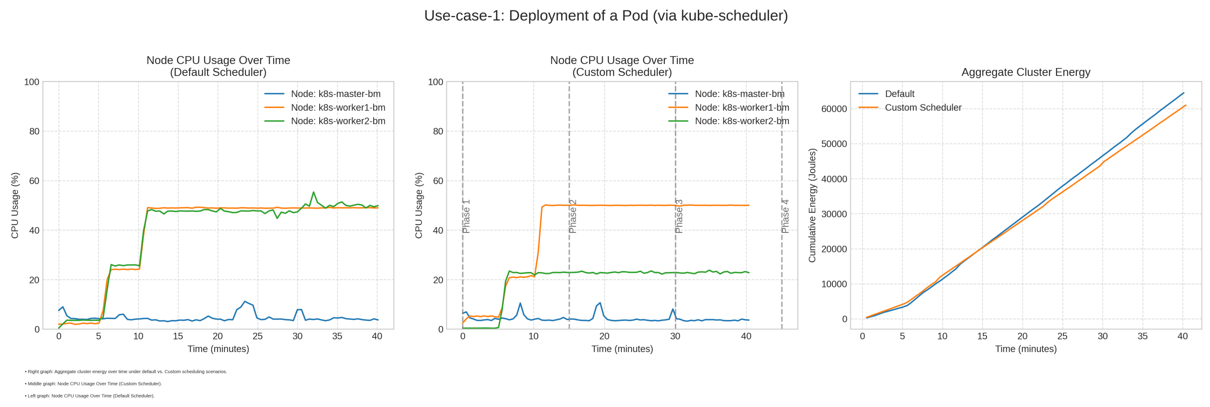
6.4.1 Energy Efficiency

The custom scheduler demonstrated improved energy efficiency achieved an average of 8% percentage energy efficiency improvement at the end of 15 minutes of workload execution and 12% improved energy efficiency at the end of 5 minutes of workload execution over the default scheduler. This improvement is mainly due to power energy efficiency custom scheduler's power aware scheduling decisions which prioritize workload placement on fewer more energy efficient nodes

Fig 13 right graph confirms which custom scheduler's cumulative energy consumption consistently below the default and Fig 14 where custom scheduler maintains a lower energy profile even during dynamic scaling when compared to Kubernetes scheduler

6.4.2 Resource Utilization

Custom scheduler achieved better cluster-wide resource utilization through intelligent workload placement(packing) particularly in the heterogeneous nodes environment. As showing in Fig 13 (middle graph) custom scheduler actively packs workloads onto fewer nodes (e.g., k8s-worker1-bm showing higher utilization), leaving others (k8s-worker2-bm) largely idle when compared to default scheduler (Left graph) which tends to spread workloads more evenly leading to even utilization and higher power consumption across the most active nodes

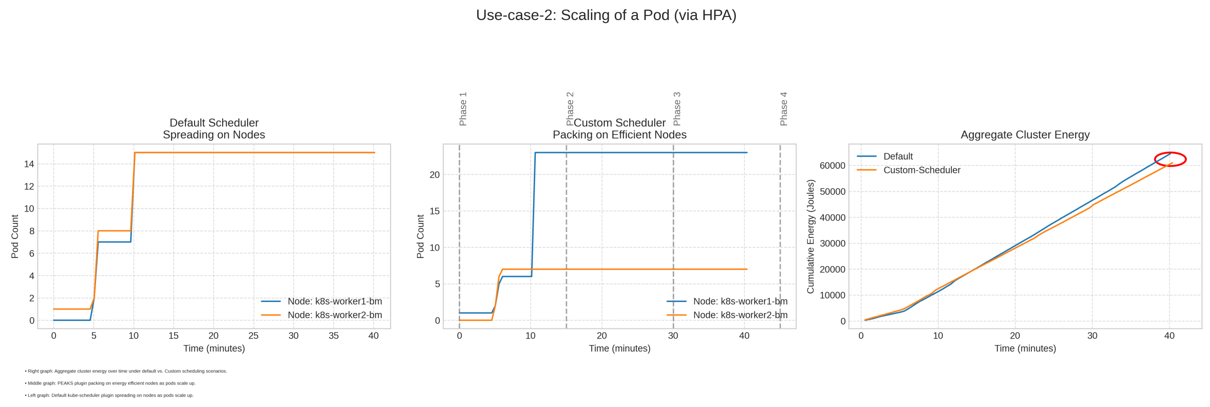


• Right graph: Aggregate cluster energy over time under default vs. Custom scheduling scenarios.

• Middle graph: Node CPU Usage Over Time (Custom Scheduler).

• Left graph: Node CPU Usage Over Time (Default Scheduler).

Figure 13: Use-case 1 - Deployment of a Pod



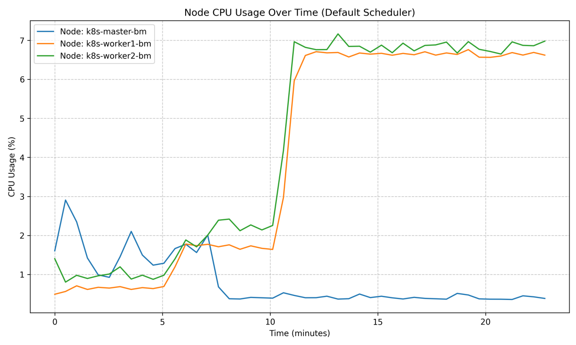
• Right graph: Aggregate cluster energy over time under default vs. Custom scheduling scenarios.

• Middle graph: PEAKS plugin packing on energy efficient nodes as pods scale up.

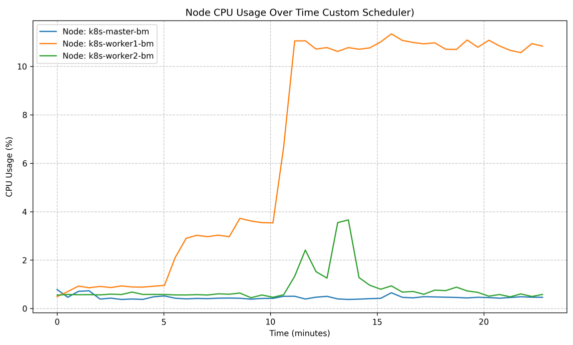
• Left graph: Default kube-scheduler plugin spreading on nodes as pods scale up.

Figure 14: Use-case 2 - Scaling of a Pod (via HPA)

Figures 15a and 15b provide a granular view of CPU utilization. Figure 15a (Default Scheduler) shows moderate CPU usage across multiple worker nodes. In contrast, Figure 15b (Custom Scheduler) clearly illustrates k8s-worker1-bm reaching significantly higher CPU utilization, while k8s-worker2-bm remains at very low utilization, demonstrating effective workload packing (scheduling) on the more powerful node. This strategy is particularly beneficial in heterogeneous environments, where maximizing utilization of high-capacity nodes can lead to greater energy savings.



(a) Node CPU Usage Over Time (Default Scheduler)



(b) Node CPU Usage Over Time (Custom Scheduler)

Figure 15: CPU usage Default vs Custom Scheduler

6.5 Discussion

The evaluation confirms the custom scheduler is better enhancing the energy efficiency in Kubernetes environments achieving 8% improvement without compromising performance. This aligns with our goal towards environmental sustainability , economic benefits through reduced operational costs and optimized resource utilization

7 Conclusion and Future Work

The Completion of this implementation demonstrated the potential of custom scheduler based on the PEAKS framework to enhance energy efficiency in cloud-native environments through various experimental evaluation on a heterogeneous bare-metal Kubernetes cluster. The custom scheduler achieved an average of 8% energy efficiency improvement when compared to default kubernetes scheduler which takes binding of pods based on power-aware decision which prioritizes workload distribution on fewer more energy efficient nodes. This approach minimizes energy waste especially during dynamic scaling events. This thesis highlights the critical role of power awareness scheduling in favor to environmental sustainability economic benefits through reduced operational costs and optimized resource utilization in cloud infrastructures

8 Future Work

Future work can further enhance the custom scheduler and sustainable cloud computing through several key consideration:

- **Advance Power Modeling**: Incorporate machine learning for more precise energy and power calculation for virtual machines
- **Diverse Workload Integration**: Extend optimization to memory-intensive, I/O-intensive and network intensive application
- **Large Scale Evaluation**: Validate Scalability and robustness on larger, geographically distributed hetrogeneous cluster and mulit cluster with `https://liqo.io/`

References

CNCF kepler conference (2022).
URL: *https://github.com/sustainable-computing-io/kepler/blob/main/doc/OSS-NA22.pdf*

CNCF peaks (2022).
URL: *https://github.com/sustainable-computing.io/peaks/blob/main/docs/design/overview.md*

Dakic, V., Đambić, G., Slovinac, J. and Redžepagić, J. (2025). Optimizing kubernetes scheduling for web applications using machine learning, *Electronics* **14**: 863.

Escrig, J., Fernández-Fernández, A. and Touma, R. (2024). Optimizing energy consumption of kubernetes clusters with deep reinforcement learning, *CCIA*.

Foundation, C. N. C. (2023). Cncf annual survey 2023.
URL: *https://www.cncf.io/reports/cncf-annual-survey-2023/*

James, A. and Schien, D. (2019). A low carbon kubernetes scheduler.

Li, H., Rao, W., Hu, B., Tian, Y. and Shen, J. (2025). Energy-aware elastic scaling algorithm for microservices in kubernetes clouds, *Journal of Network and Computer Applications* **242**: 104218.
URL: *https://www.sciencedirect.com/science/article/pii/S1084804525001158*

Ma, H. and Ding, A. (2022). Method for evaluation on energy consumption of cloud computing data center based on deep reinforcement learning, *Electric Power Systems Research* **208**: 107899.
URL: *https://www.sciencedirect.com/science/article/pii/S0378779622001298*

Medel, V., Tolón, C., Arronategui, U., Tolosana-Calasanz, R., Bañares, J. Á. and Rana, O. F. (2017). Client-side scheduling based on application characterization on kubernetes, *Economics of Grids, Clouds, Systems, and Services: 14th International Conference, GECON 2017, Biarritz, France, September 19-21, 2017, Proceedings 14*, Springer, pp. 162–176.

Piontek, T., Haghshenas, K. and Aiello, M. (2023). Carbon emission-aware job scheduling for kubernetes deployments, **80**(1).
URL: *https://doi.org/10.1007/s11227-023-05506-7*

redhat (2023). redhatenterprise.
URL: $https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_for_real_time/8/html/optimi$
$testing - real - time - systems - with - stress - ng_optimizing - rhel8 - for -$
$real - time - for - low - latency - operation$

Schmitt, N., Iffländer, L., Bauer, A. and Kounev, S. (2019). Online power consumption estimation for functions in cloud applications, *2019 IEEE International Conference on Autonomic Computing (ICAC)*, IEEE, pp. 63–72.

Senjab, K., Abbas, S., Ahmed, N. and Khan, A. u. R. (2023). A survey of kubernetes scheduling algorithms, *Journal of Cloud Computing* **12**(1): 87.

Shalf, J. (2020). The future of computing beyond moore's law, *Philosophical Transactions of the Royal Society A* **378**(2166): 20190061.

Shapi, M. K. M., Ramli, N. A. and Awalin, L. J. (2021). Energy consumption prediction by using machine learning for smart building: Case study in malaysia, *Developments in the Built Environment* **5**: 100037.
URL: *https://www.sciencedirect.com/science/article/pii/S266616592030034X*

Townend, P., Clement, S., Burdett, K., Yang, R., Shaw, J., Slater, B. and xu, J. (2019). Improving data center efficiency through holistic scheduling in kubernetes, pp. 156–15610.

Vijouyeh, L. N., Sabaei, M., Santos, J., Wauters, T., Volckaert, B. and De Turck, F. (2020). Efficient application deployment in fog-enabled infrastructures, *2020 16th International Conference on Network and Service Management (CNSM)*, pp. 1–9.

Xu, C., Rajamani, K. and Felter, W. (2018). Nbwguard: Realizing network qos for kubernetes, *Proceedings of the 19th international middleware conference industry*, pp. 32–38.