Scheduling Energy-Aware Multi-Function Serverless Workloads in OpenFaaS

Rauian Chiorescu, Karim Djemame

School of Computing
University of Leeds
Leeds LS2 9JT
UK

Abstract

The paper investigates the prediction capabilities of a Machine Learning model in real-time scheduling applications on Kubernetes in a serverless computing environment with the aim to achieve a degree of energy efficiency. A highly pluggable framework for integrating a learning-based model into the Kubernetes scheduler is proposed and evaluated in a serverless setup on OpenFaaS. The experimental results in a cloud-native deployment demonstrate that, while maintaining Quality of Service for the application, an overall 8% in power reduction is achieved at a minimal performance loss.

1 Introduction

In the current climate, cloud computing plays a vital role in sustaining various aspects of our digital lives, from social media to large-scale data analysis. As the demand for these services continues to skyrocket, so does the need for more cloud data centres, which in turn increases overall energy consumption worldwide. As a consequence, energy efficiency has become an imperative issue for cloud computing service providers to lower costs as well as minimise the energy consumption impact on the environment.

Serverless computing is an architectural paradigm that enables developers to focus on application functionality rather than the underlying infrastructure. By abstracting away the underlying servers, serverless computing allows for automatic scalability, flexibility, and potential cost reductions, as users only pay for the number of invocations of their specific functions. This paradigm shift has transformed how applications are developed and deployed, leading to its widespread adoption in the cloud computing industry. On the other hand, as with any relatively novel technology, new challenges are introduced in the field. One of these challenges is to measure and potentially optimise energy usage in a cloud environment, as users do not have access to the underlying infrastructure. Moreover, a serverless function is essentially a proxy for energy usage as a unit of (serverless) compute and therefore a cost, making functions instantiation and orchestration significantly energy and resource efficient [7].

OpenFaaS [1] is an open-source serverless platform that provides a framework for building serverless functions by allowing developers to package any process or container as a function facilitating their deployment without requiring extensive adaptation. OpenFaaS relies on Kubernetes for scheduling containers that run the serverless functions and is considered as an attractive serverless engine due to Kubernetes's extensibility. The evaluation of OpenFaas has revealed superior power efficiency as compared to a Docker containers setting [4]. As Machine Learning (ML) has gained increased traction and is being adopted in some critical areas of resource scheduling, planning and control, this paper addresses the question of whether the integration of a ML-based model into the Kubernetes scheduler can achieve *better* power efficiency as compared to the default scheduler. A change at the function or scheduler level must therefore be made in order to achieve energy-aware serverless functions.

The paper makes the following contributions:

- we propose a highly pluggable framework for integrating a learning-based model into the Kubernetes scheduler;

- we evaluate the performance of the proposed scheduler extension against the default Kubernetes Scheduler;
- we demonstrate that significant power efficiency is gained by integrating a learning-based model into the Kubernetes scheduler.

The paper is structured as follows: Section 2 reviews the related work and looks into the research landscape surrounding scheduling functions on serverless computing platforms. The design addressing the framework for integrating a learning-based model into the Kubernetes scheduler is presented in Section 3. The experimental environment setup and the results of the evaluation of the custom scheduler are described in Section 4. Section 5 concludes with a summary of the research findings and suggestions for future work.

2 Related Work

There have been a multitude of research investigations in the area of energy consumption in the serverless space. Alhindi et al. [4] investigate the difference in power usage between OpenFaaS and Docker across four scenarios, in a on-premises hardware, virtual machine-based experimental setup: standalone Docker containers, OpenFaaS with faasd, a lightweight runtime of OpenFaaS that runs on containerd directly under the hood, OpenFaaS on Kubernetes and Docker containers on Kubernetes. They discovered that OpenFaaS with faasd tends to consume less power on a memory-heavy benchmark than its Kubernetes counterpart with a decrease of 58%, which is expected, as the runtime is more lightweight.

Jia and Zhao [13] propose a mechanism for energy-aware resource allocation in a serverless context to minimise power consumption named RAEF. An agent running at the function level is proposed, constructed of four components, where the most important are the predictor and the resource explorer. The findings show that the proposed solution can reduce energy consumption anywhere from a noticeable 9.7% to a significant 21.2% across different workloads. Moreover, the control-plane components of OpenFaaS consumes an insignificant amount of power compared to the function runtime plane.

Rocha et al. [15] implement a Kubernetes scheduler extension based on multiple linear regression models in order to predict energy and performance based on the workload, CPU and memory requirements. The approach includes a user set weight between 0 and 1 in decimal increments, that determines the desired balance between energy and performance. 7.1% improvement in terms of energy is achieved at the detriment of ∼10% performance, while their maximum performance setting improves runtime by ∼20% at a 5% energy increase.

Toka et. al [16] propose a Kubernetes scaling engine that makes the auto-scaling decisions apt for handling the actual variability of incoming requests. Four different approaches to a learning-based solution are devised auto-regressive (AR), unsupervised (HTM), deep learning (LSTM) and reinforcement learning, to determine which one performs better by letting them compete against each other via a scoring mechanism based on past predictions. The experimental results show that at a slight increase in resource usage (2-9%) the system manages to decrease the loss in requests from 22% at the lowest to 72% percent at the highest.

The work of Das et al. [6] focuses on cost efficient execution of multi-function serverless applications on hybrid cloud deployments based on OpenFaaS and AWS Lambda. A framework named Skedulix is proposed based on a greedy solution to scheduling after it was modelled into a Mixed Integer Linear Program (MILP). Serverless applications are modelled as directed acyclic graphs (DAGs), in which each node represents a different function. The goal of the scheduler is to speed up processing at the lowest cost possible, by maximising the use of the private cluster and offloading any incompletable work to the public cloud in cases where the deadline cannot be met.

Fan and He [9] target the all too well-known issue of cold start by devising a new scheduling strategy that allows creating multiple pods at a time instead of the default pod-by-pod approach that Kubernetes adopts as it has to traverse each node to compute the score that determines the most beneficial placement which can cause a slowness in start-up latency. Essentially, they propose a simple algorithm based on Mixed Integer Programming (MILP), that divides pods into groups and schedules a group at a time when it passes a set overload threshold and achieves sizeable reduction in latency in simulation, ranging from ∼20-60% depending on the number of pods that need to be scheduled.

In summary, work has addressed scheduler-level optimisations in Kubernetes, although none seem to consider energy consumption in a serverless context. Most are centred around performance but none considers the public cloud in their experiments. In terms of approach and workload, to the best of our knowledge no work targets a multi-function workload while considering energy efficiency. This work is aimed at an image processing pipeline on OpenFaaS and uses a learning-based approach for performing scheduling decisions. While RAEF focuses on function-level adjustments [13], this paper explores possibilities at the scheduler level, although similarly it aims to thread the needle between Quality of Service (QoS) provision boundaries in order to minimise power consumption.

3 System Design

The principal research question addressed in this paper is: *Following the integration of a ML-based model into the Kubernetes scheduler, are results reliable in presenting a realistic trade-off between performance and energy consumption?* This concludes if, following an evaluation of the custom Kubernetes scheduler, the results do illustrate a clear trade-off between energy consumption and performance, and if the reduction in energy is worthwhile in the sense that it maintains an acceptable QoS.

3.1 Architecture

Figure 1 illustrates the architectural design and flow of our proposed implementation of the system. The flow is initiated with an agent situated on the monitoring stack of the solution. It invokes an endpoint part of an interim component between the ML model and the Kubernetes Cluster. Following the invocation, the container API retrieves resource usage metrics from the monitoring infrastructure Prometheus [3] and forwards them to the model to retrieve a prediction, sending it back to the agent which stores on each node their own version of the prediction result, being either schedulable or unschedulable at a specific point in time. Then, an adapter for Prometheus formats the result of the prediction, and passes it to the control plane of the cluster via the API server component. Finally, the custom scheduler implementation processes the formatted scheduling instruction metric and schedules new pods on the predicted nodes accordingly.

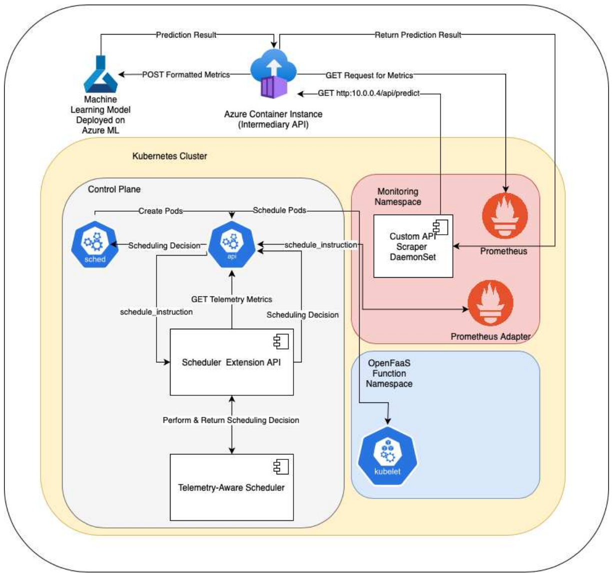

Figure 1: Flow and Architecture

The core component of the system is a custom variation of Intel's *Telemetry Aware Scheduler* [12]. Essentially it is used to perform scheduling decisions based on various metrics. The manner in which it operates is based on files designated as telemetry policies. The user can define multiple scheduling strategies based on the desired metrics that should take part in scheduling decisions. Every metric can be assigned one of three operators, *Equals*, *LessThan* and *GreaterThan*. In our case this is signified by the *schedule_instruction* metric, a pre-formatted prediction result, based on which the custom solution performs scheduling decisions.

3.2 Machine Learning Model

For the machine learning model Scikit-learn's Random Forest Classifier was employed [14]. This was chosen for its ability to handle a large number of features and its robustness against over-fitting.

CPU and the memory are the most important resources to be considered in terms of performance and power consumption in the context of applications execution [8]. Resource consumption metrics (CPU and Memory Usage, Power, Pod Count) were collected from Prometheus on each node over multiple load experiments to serve as training and validation data. These experiments involved varying the load on the Kubernetes cluster in a controlled manner and collecting metrics at regular intervals.

The predictions outputted by the model are in the form of
k8s_worker_[NODE_ID] and represent which node would be the most appropriate to schedule a new pod on based on the current metrics that are fed into it.

The model was deployed in Azure Machine learning lab. in its own custom environment [5]. It exposes a REST endpoint that can be accessed with a POST request with the required usage data in order for it to output a prediction. This manner of utilising the capabilities of Azure ML and Azure Container Instances is what deems the proposed solution highly-pluggable as models are trivial to interchange, and any of the metrics that are desired for the model to predict a node require simple modifications at the level of the API running as the Container Instance.

3.3 Research Hypotheses

As a result of the architectural design and implementation of the proposed solution, a total of three hypotheses are formulated as predictions for the experimental outcomes. As such, experiments will test those hypotheses and the results of each experiment will prove or disprove them.

H1. *The ML solution will be reliable in scheduling pods, with little added delay between scheduling decisions.* This hypothesis is based on research publications that used a learning-based approach in building custom scheduling solutions for Kubernetes.

H2. *Towards the latter end of the experiments, where resource usage is high both solutions will perform similarly.* This is based on the fact that if all the compute resources in the cluster are needed, there will not be any technique that will consistently schedule pods differently. Simply the cluster would be filled up with all CPU cores being pinned close to 100%.

H3 *The default scheduler will outperform the custom scheduler in every scenario in terms of throughput.* This hypothesis is based on previous literature in terms of energy and performance trade-offs, where in seldom cases you can have both. The added overhead of custom scheduling decisions will likely impact the scaling capabilities of the cluster.

4 Experiments and Results

4.1 Experimental Setup

As the path of experimentation in this paper is to evaluate a chain-function setup, a CPU and Memory intensive workload was chosen, namely, image processing. The pipeline consists of four serverless functions:

- Director Function, a design pattern in multi-function setups as it only requires the client to submit a single HTTP request to fully process an image instead of individual requests to each serverless function. Its purpose is to call all the other functions
- Image Resize Function (Upscaling)
- Image Transform Function (Rotation)
- Image Recolour Function (RGB Manipulation)

The benchmarking client is a Standard_8s_v5 Virtual Machine in Azure configured with accelerated networking to allow for high request throughput. This size was chosen to ensure the reliability of running hundreds of concurrent clients that send web requests to the director functions deployed onto the Kubernetes cluster.

Apache JMeter [11] is chosen as the primary load testing tool due to its high configurability of flows and easy extensibility when it comes to benchmarking different function deployment scenarios with varying amounts of initial replicas. The test plan employs a multi-threaded approach, simulating a large number of virtual users to stress-test the system.

To measure power, the proposed power model in Fan et al. [10] is adopted as it is deemed more reliable than, e.g. a tool such as PowerTop [2], in a virtualised environment. This model applies the Thermal Design Power (TDP) values of the processors running within the cluster which are Intel Xeon Platinum 8730C, featuring 32 cores and 64 threads with a TDP of 270W. In our system, there are 40 vCores (Virtual Cores) worth processing power which is equivalent to 40 threads.

Experiments are run over a varying amount of initial function deployments, in order to observe the differences in resource usage metrics and power consumption between them. Five runs of each scenario are run for consistency and validity of results, each deployment is tested on the custom ML-

Accuracy	**Precision**	**Recall**	**F1 Score**
94.59%	95.80%	94.59%	94.79%

Table 1: Performance Metrics of the Random Forest Classifier

agent based scheduler as well as Kubernetes default scheduler. The runs are broken down as follows:

- 10 deployments per function (40 pods total with a theoretical maximum of 200 pods)
- 20 deployments per function (80 pods total with a theoretical maximum of 400 pods)
- 30 deployments per function (120 pods total with a theoretical maximum of 600 pods)
- 40 deployments per function (160 pods total with a theoretical maximum of 800 pods)
- 50 deployments per function (200 pods total with a theoretical maximum of 1000 pods)

4.2 ML Model Performance

The model was trained on a dataset comprised of features extracted every 2 minutes from the Kubernetes cluster while being under load for a total of 6 hours resulting in a total of approximately 3000 data points. The evaluation metrics were computed using a test set, 20% of the entire dataset, that was not part of the training data. The confusion matrix was constructed to capture the true positives, true negatives, false positives, and false negatives for each class. Table 1 illustrates the performance metrics achieved for the ML model, broken down into accuracy, precision, recall and F1 Score.

Figure 2 presents the confusion matrix generated following training and validation. The matrix is comprised of two categories, predicted label and true label, with the main diagonal representing true values and the corners signifying false predictions.

4.3 Custom Scheduler Evaluation

Figure 3 shows the breakdown of average power usage at varying levels of deployments between the two scheduling solutions. At 10 deployments, the custom scheduler consumes approximately 15.7% more (Watt) power (117.07W)

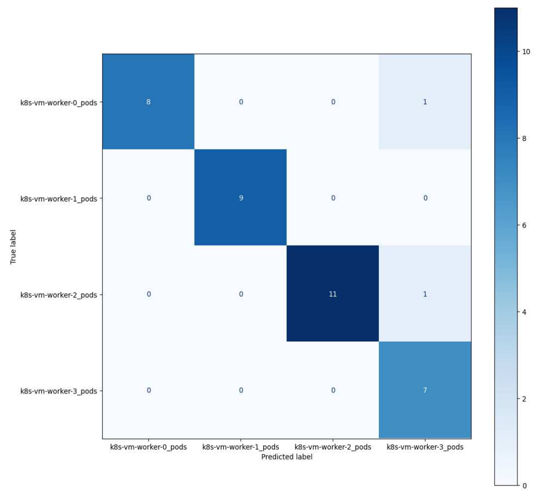

Figure 2: Confusion Matrix of Test Set Predictions of the Random Forest Classifier

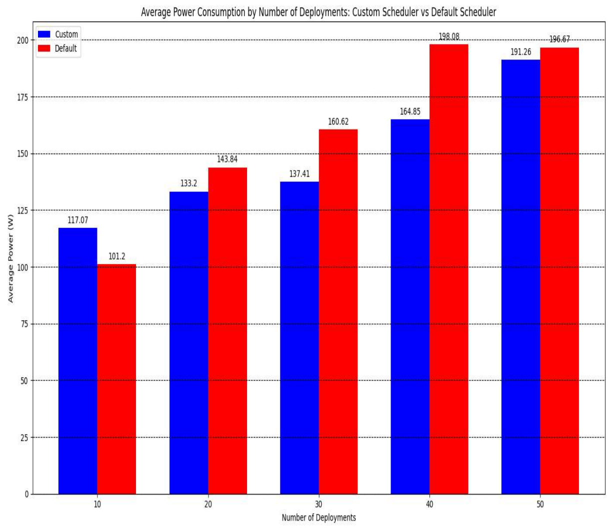

Figure 3: Average Power Usage by Number of Deployments

than the default scheduler (101.2W). However, as the number of deployments increases, the custom scheduler tends to become more power-efficient. At 20 deployments, the custom scheduler consumes about 9.6% less power (130.01W) compared to the default (143.84W). At 30 deployments, the power saving increases to 14.5% (137.41W for custom vs. 160.62W for default). The trend continues at 40 deployments with a 16.7% reduction (164.95W for custom vs. 198.08W for default). Finally, at 50 deployments, the custom scheduler consumes approximately 2.8% less power (191.26W vs. 196.67W for default). At 50 deployments, the CPU usage is pinned towards 100% signifying that the hardware limitations of the cluster is reached. If further testing were to be conducted on a larger cluster, we would most likely observe a similar pattern to the 20,30 and 40 deployment's power values.

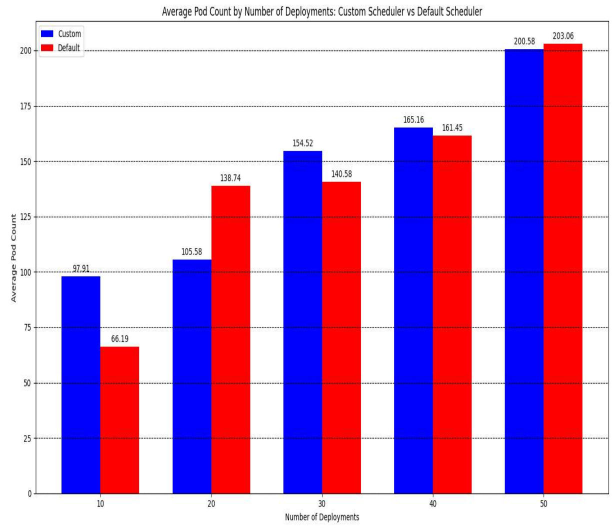

Figure 4: Average Pod Count by Number of Deployments

The number of pods metric 4, provides insights into container orchestration efficiency as well as how many pods per function are needed for the functions to execute at an acceptable rate. At 10 deployments, the custom scheduler uses about 31.8% fewer pods (97 vs. 66 for default). At 20 deployments, the default scheduler uses a significant 31.4% more pods (138 vs. 105 for custom). However, at 30 deployments, the custom scheduler uses about 10% more pods (154 vs. 140 for default). At 40 and 50, the gap closes, with custom scheduler uses about 2.5% more pods (165 vs. 161 for default) and respectively, 1.5 % more (200 vs. 203 for default).

Figure 5 presents the average CPU usage, which is directly proportional to 3 as power estimations are directly based on CPU. At 10 deployments, the custom scheduler uses about 10.7% more CPU (62.68% vs. 56.56% for default). However, at 20 deployments, the CPU usage is nearly identical, with the custom scheduler using about 2.8% less CPU (69.06% vs. 71.72% for default). At 30 deployments, we see a marginal increase in CPU, although, the custom scheduler uses 12.7% less CPU (69.82% vs. 79.73% for default). The gap widens at 40 deployments, where the custom scheduler uses 14.6% less CPU (82.5% vs. 96.51% for default). Finally, at 50 deployments, the difference narrows to about 2.1% less CPU usage for the custom scheduler (93.7% vs. 95.84% for default). As mentioned in the presentation for power usage, the CPU tends to be pinned close to maximum with 50 deployments, not allowing a varied placement of pods on the nodes when essentially each node is maxed out in usage.

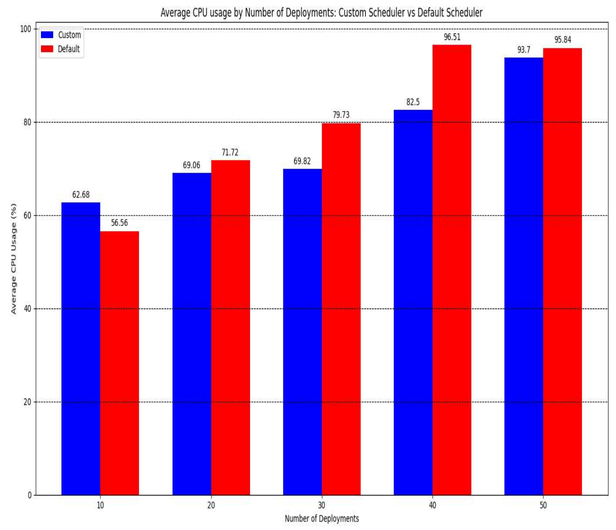

Figure 5: Average CPU Usage by Number of Deployments

In Figure 6, memory consumption tends to be lower for the custom scheduler in small deployments with an outlier at the minimum number of 10. The main consumer of memory is the resize serverless function, which upscales the chosen image to a 1920x1080 resolution, and the values suggest that the performance of the resize function, should in theory, be similar. At 10 deployments, the custom scheduler uses about 15% less memory (11.16 GiB vs. 13.85 GiB for default). At 20 deployments, the memory saving is significant at 38% (8.61 GiB for custom vs. 12,91 GiB for default). However, at 30 deployments, the custom scheduler uses about 29% less memory (12.61 GiB vs. 17.8 GiB for default). At 40 deployments, the memory usage is almost identical, with the custom scheduler using about 5% less

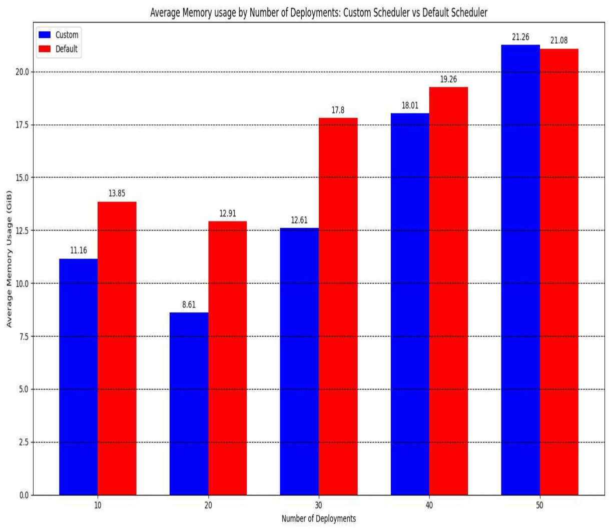

Figure 6: Average Memory Usage by Number of Deployments

No.	Avg.	Med.	90th	95th	99th	Err%	Thr/s
10	1167	445	3056	5548	9119	41.18%	36.2
20	4885	5079	9103	9575	9943	39.9%	41.5
30	4785	4719	8906	9474	9933	44.14%	50.4
40	6834	7263	9536	9795	9993	62.74%	33.8
50	7423	7716	9624	9844	10015	72.25%	29.3

Table 2: Request Execution Times (ms) - Custom Scheduler

No.	Avg.	Med.	90th	95th	99th	Err%	Thr/s
10	1450	270	6064	8176	9672	53.44%	24.3
20	3965	3614	8066	8974	9820	29.47%	56.6
30	6299	6551	9344	9693	9972	45.86%	50.0
40	7193	7453	9551	9802	10004	58.11%	45.5
50	7089	7316	9538	9798	10005	69.68%	31.6

Table 3: Request Execution Times (ms) - Default Scheduler

memory (18.01 GiB vs. 19.26 GiB for default). Finally, at 50 deployments, both schedulers use virtually the same amount of memory (21.26 and 21.08 GiB).

Tables 2 and 3 present the average, median, 90, 95 and 99th percentiles response times of requests in all the deployments evaluated. With this knowledge, a comparison of the custom and the default schedulers is performed with the aim to conclude if an acceptable QoS has been achieved by custom scheduler through observing the energy vs performance trade-off. Note that the metrics displayed are only for successful (200 OK) requests and not errors (500 Internal Server Error).

Throughput (Thr/s): the custom scheduler generally shows higher throughput in lower numbers of deployments (36.2 vs. 24.3 at 10 deployments). However, as the number of deployments increases, the default scheduler starts to outperform the custom one, which is expected, and especially noticeable at 20 deployments (41.5 vs. 56.6), considering the previous comparison in power consumption.

Average Execution Latency (Avg.): the custom scheduler starts with a lower average latency at 10 deployments (1167 vs. 1450 ms). Although, as the number of deployments grows, the custom scheduler's average latency becomes competitive or even better than the default scheduler, particularly noticeable at 20 and 30 deployments.

Percentile Execution Latencies (90, 95, 99th): Both schedulers show increasing latencies as we move from the median to the 99th percentile, which is expected. The custom scheduler generally has higher 99th-percentile latencies, indicating that it may be less reliable under heavier loads.

The error rates are notably high for both schedulers but are generally higher for the custom scheduler across all numbers of deployments. These values in error rate can be attributed to the limitations of the OpenFaaS Gateway scaling capabilities.

4.4 Research Hypotheses Evaluation

H1. *The ML solution will be reliable in scheduling pods, with little added delay between scheduling decisions.* This has proven to be true, with the proposed model achieving a 94.59% accuracy at predicting nodes on the test set. The accuracy metric is given by how similar the output of the model is to that of the default scheduler i.e. effectively the node label that a pod was scheduled on by the default scheduler. In the real-world deployment, it has added negligible overhead in terms of scheduling latency. If the logs of the scheduling extension are inspected, decisions are made in less than a second, deeming the solution reliable for real-time scheduling.

H2. *Towards the latter end of the experiments, where resource usage is high both solutions will perform similarly.* This hypothesis has proven to be entirely true as towards 50 deployments, the resources in the cluster were completely saturated, and both scenarios performed similarly across the board with a difference 1-3%, in terms of all the metrics considered, including performance. The results do truly show that scheduling decisions do not matter all that much when there is a lack of resources present.

H3. *The default scheduler will outperform the custom scheduler in every scenario in terms of throughput.* This hypothesis has proven to be partially true. We were intrigued by the results in the case of the two outliers in this experiment at 10 and 30 deployments. At 10 deployments the increase in performance directly correlates with the resources used in our custom scheduler scenario. However, the most intriguing result was at 30 deployments. Despite the custom scheduler consuming significantly less power, it managed to achieve a throughput of 50.4, essentially on par with the default scheduler's 50.0. The custom scheduler was able to deliver nearly identical throughput while being more power-efficient by 14.5%. This performance was consistent across multiple runs of the experiment though.

4.5 Summary

Apart from a couple outliers, the results do show a realistic trade-off between performance and power consumption. If all the scenarios to calculate an average in power reduction are considered, an 8% decrease in power consumption is achieved. With the outliers at 10 deployments and 30 deployments removed, the total average savings are 10.7%. In terms of performance loss the average, median, 90, 95 and 99th percentile values tend to be rather close to each other when comparing the two solutions. Finally, to answer the main research question, the proposed custom scheduler can provide an acceptable QoS depicted by the differences of error rate and throughput, while the implementation does pose higher values for dropped requests and lower for throughput in most cases. The reduction in power consumption can be considered worthwhile as the losses are an expected outcome when trying to save power and they situate themselves in the same realm to those of the default scheduler.

5 Conclusion

This paper has presented an energy-efficient pluggable solution to scheduling in Kubernetes in a serverless computing environment through the integration of a learning-based model into the scheduler. In terms of overall metrics this solution achieved 8% reduction in energy consumption at the cost of a directly correlated loss in performance, thus maintaining the application QoS. Future work will include the generalisation of the proposed model to other workloads such as data analytics and video processing, as well as experimentation with other solutions for scheduling in Kubernetes, e.g. investigate the use of Deep Learning-based schedulers, Mixed-Integer Linear Programming (MILP), or other optimisation techniques for scheduling tasks in Kubernetes.

References

[1] Openfaas - serverless functions, made simple, 2023. https://openfaas.com/.

[2] Powertop, 2023. https://wiki.archlinux.org/title/powertop.

[3] Prometheus. from metrics to insight, 2023. https://prometheus.io/.

[4] A. Alhindi, K. Djemame, and F. Banaie. On the power consumption of serverless functions: an evaluation of openfaas. In *Proc. of the 15th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 366–371, Vancouver, USA, Dec. 2022. IEEE.

[5] Microsoft Azure. Azure machine learning, 2023. https://azure.microsoft.com/en-gb/products/machine-learning.

[6] A. Das, A. Leaf, C.A. Varela, and S. Patterson. Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. In *2020 IEEE 13th International Conference on Cloud Computing*, pages 609–618, Beijing, China, 2020. IEEE.

[7] K. Djemame. Energy efficiency in edge environments: a serverless computing approach. In K. Tserpes, J. Altmann, J.A. Bañares, O. Agmon Ben-Yehuda, K. Djemame, V. Stankovski, and B. Tuffin, editors, *Economics of Grids, Clouds, Systems, and Services*, pages 181–184, Cham, 2021. Springer International Publishing.

[8] K. Djemame, M. Parker, and D. Datsev. Open-source Serverless Architectures: an Evaluation of Apache

OpenWhisk. In *Proc. IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 329–335, Leicester, UK, 2020. IEEE.

[9] D. Fan and D. He. A scheduler for serverless framework base on kubernetes. In *Proc. 2020 4th High Performance Computing and Cluster Technologies Conference & 3rd International Conference on Big Data and Artificial Intelligence*, HPCCT & BDAI '20, page 229–232, NY, USA, 2020. ACM.

[10] X. Fan, W.D. Weber, and L.A. Barroso. Power provisioning for a warehouse-sized computer. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, page 13–23, NY, USA, 2007. ACM.

[11] Apache Software Foundation. Apache jmeter, 2023. https://jmeter.apache.org/.

[12] Intel. Telemetry aware scheduling, Nov. 2021. https://www.intel.com/content/www/us/en/developer/articles/technical/telemetry-aware-scheduling.html.

[13] X. Jia and L. Zhao. Raef: Energy-efficient resource allocation through energy fungibility in serverless. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems*, pages 434–441, Los Alamitos, CA, USA, Dec 2021. IEEE.

[14] O. Kramer. *Scikit-Learn. In: Machine Learning for Evolution Strategies*, pages 45–53. Springer, Cham, 2016.

[15] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. Heats: Heterogeneity-and energy-aware task-based scheduling. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 400–405, Pavia, Italy, 2019. IEEE.

[16] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly. Machine learning-based scaling management for kubernetes edge clusters. *IEEE Transactions on Network and Service Management*, 18(1):958–972, 2021.