Enabling Energy-Efficient Kubernetes Cluster Autoscaler

Enabling Energy-Efficient Kubernetes Cluster Autoscaler

Riccardo Miracapillo, Stefano Galantino, Attilio Oliva, Fulvio Risso
Dept. of Computer and Control Engineering
Politecnico di Torino
Torino, Italy
{name.surname}@polito.it

***Abstract*—Energy waste in underutilized bare-metal clusters remains a critical challenge for cloud providers. Powering off such servers could significantly reduce operational costs. However, selecting the best server to be powered off is a complex optimization problem; incorrect selection risks resource saturation and compromised service availability. This work presents DREEM, a Kubernetes-based autoscaling mechanism specifically designed for bare-metal infrastructures. DREEM integrates a server selection algorithm that reduces overall energy consumption while preserving application performance, according to user-defined preferences. We showcase DREEM operating on a small cluster, dynamically adapting its size to the running workload. The results highlight how DREEM effectively reduces the average number of active nodes, lowering energy consumption without compromising performance or responsiveness, while consistently selecting near-optimal servers according to the defined constraints.**

***Index Terms*—cloud computing, energy optimization, Kubernetes, auto-scaling**

I. Introduction

Public cloud computing offers several advantages, such as removing the need for physical server maintenance and the availability of managed, ready-to-use services. Nevertheless, on-premises infrastructures still remain widely adopted due to benefits such as data privacy, full control over hardware resources, and the absence of recurring cloud costs. In such a context, achieving dynamic infrastructure scaling is a critical yet elusive goal. Ideally, an efficient on-premise cluster should scale up to accommodate increasing demand and scale down during idle periods to reduce expenditures, particularly by powering off underutilized nodes and lowering energy costs.

However, replicating the seamless autoscaling of public clouds in a resource-constrained bare-metal environment poses several challenges. The high setup time typical of on-premises bare-metal servers clashes with the reactive approach used by standard tools. For instance, Cluster Autoscaler [1], with its reactive policies, struggles to account for delays inherent in the provisioning of bare-metal servers. Furthermore, its scale-down operations often struggle with efficiency unless aggressive configurations are applied. Conversely, Karpenter [2] addresses some of these inefficiencies, but it remains limited to cloud-provider ecosystems (e.g., AWS), making it unsuitable for on-premises deployments. Similarly, research on energy-aware consolidation [3], [4] largely focuses on Virtual Machines, failing to distinguish between critical and disposable containerized tasks, i.e., following a black-box approach. Even Kubernetes-specific studies often rely on virtualization layers [5] or theoretical models that overlook hardware degradation and persistent storage [6].

Indeed, current autoscaling approaches often lack "context awareness", failing to integrate application-specific requirements, orchestration logic, and physical infrastructure constraints. This oversight may lead to suboptimal or risky decisions, such as terminating nodes hosting critical stateful workloads. Consequently, achieving a balance between aggressive energy efficiency and high-performance availability remains an unaddressed challenge, requiring a multi-layered model capable of managing these conflicting trade-offs through calibrated, objective-driven strategies.

To overcome these limitations, we introduce **DREEM** (*Dynamic node REallocation for Energy-optiMized Kubernetes clusters*), an autoscaling system specifically designed for bare-metal Kubernetes infrastructures. It explicitly models the node reallocation problem as a constraint-aware decision process. In particular, DREEM distinguishes between hard constraints, which must always be satisfied to preserve cluster correctness and application safety, and soft constraints, which express optimization preferences such as energy efficiency. This separation enables DREEM to reason about trade-offs that are largely unaddressed by current autoscaling solutions. Leveraging historical resource usage data, DREEM continuously evaluates the cluster state and dynamically provisions or decommissions nodes when needed. Its decision logic is configurable, allowing operators to adapt both constraints and objectives to different deployment scenarios and operational priorities, rather than enforcing a single predefined scaling strategy.

The remainder of this paper is structured as follows: Section II details DREEM architecture and future improvements; Section III describes the demonstration, and Section IV discusses directions for future work.

II. DREEM

DREEM dynamically powers nodes on and off in a Kubernetes cluster based on overall measured CPU utilization. It integrates natively with Kubernetes, enabling seamless cloud-native deployment and orchestration. As shown in Figure 1, DREEM is composed of three Kubernetes controllers, each responsible for a specific phase of the scaling workflow.

Once the operator is installed, users only need to configure a minimal set of parameters, including scaling thresholds, the minimum and maximum number of active nodes allowed in the infrastructure to ensure safe scaling boundaries, and the scaling profile (performance-, balanced-, or energy-oriented). The DREEM operator performs three primary actions: (*i*) it executes a decision algorithm to determine whether nodes should be added or removed according to the user-defined configuration; (*ii*) it applies a selection algorithm to identify the *optimal node* to scale, based on the chosen profile and constraints; and (*iii*) it enforces the scaling action.

A. Decision Algorithm

The first action performed by DREEM is to determine whether a scaling action is required or not through a dedicated scaling decision algorithm. Two approaches are supported and selectable by the user: a *reactive* strategy, suitable for relatively stable environments, and a *predictive* strategy, designed for highly dynamic workloads and to reduce the inherent latency in reactive decisions. The reactive approach collects and analyzes the average CPU usage over the last N minutes, where N is user-configurable, and compares it against the same CPU user-defined thresholds to determine the appropriate scaling action. The predictive alternative is based on a *Long Short-Term Memory Neural Network* (LSTM), a type of neural network well-suited for time-series prediction tasks due to its ability to capture temporal dependencies. The model, trained on part of the Alibaba Cluster Trace [7] dataset, was optimized through various hyperparameter configurations.

The system forecasts future CPU utilization and compares the average predicted value against user-defined thresholds. When the upper threshold is exceeded, a scale-up action is triggered by adding a node to the cluster. When the predicted utilization falls below the lower threshold, a scale-down process is initiated, subject to additional constraints such as resource availability, affinity rules, and server energy efficiency, before decommissioning a node. The use of predictive modeling is designed to address the significant boot-up latency of physical hardware (typically up to 20 minutes), allowing the cluster to scale preemptively and maintain service availability.

Given a configurable interval, the scaling algorithm periodically asserts the scaling necessity. If deemed necessary, a scaling request is pushed by creating a **ClusterConfiguration** resource containing the desired size. A dedicated controller performs preliminary validations, such as node availability and process consistency, and aborts the operation in case of errors. If the checks succeed, a **NodeSelecting** resource is created to trigger the node selection phase.

B. Node Selection Algorithm

The **NodeSelecting** resource triggers the execution of the selection algorithm to identify the most suitable node to be provisioned or decommissioned. It evaluates candidate nodes against multiple criteria. While scaling up is generally a straightforward process, scaling down may present specific criticalities. Consequently, the scale-down operation is divided

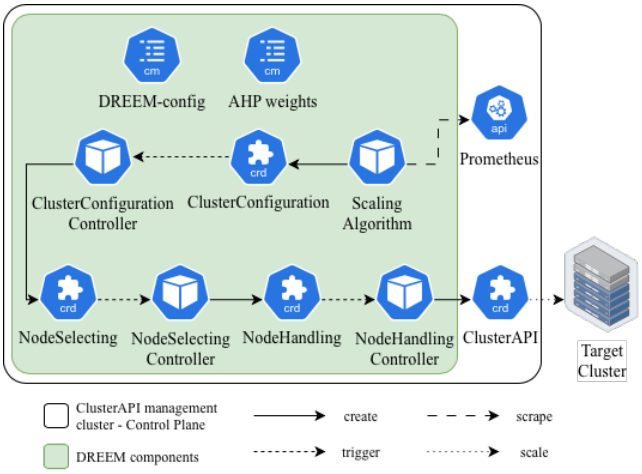

Fig. 1. Overview of DREEM architecture.

into two distinct stages: **(i)** *hard constraints* are enforced to ensure that all pods running on a candidate node can be safely rescheduled elsewhere in the cluster; nodes violating any mandatory condition are discarded. These checks are performed through a lightweight *simulated scheduling* phase, whose goal is not to replace the Kubernetes scheduler, but to ensure that at least one valid pod placement configuration exists. **(ii)** *Soft constraints* are evaluated to rank the remaining candidates according to optimization criteria. They account for application-level constraints, e.g., preferences in node/pod affinity for the scheduling process, the number of pod replicas running simultaneously on the same node. Soft constraints also account for infrastructure-level characteristics such as *Power Cycles* metric, defined as the the cumulative number of system reboots and minimized to balance hardware degradation, and the *Energy Profile* metric, derived from the WAP metric [8] and normalized to account for heterogeneous CPU core counts. The latter provides a core-normalized measure of power consumption relative to utilization for each server. The profile is dynamically updated using a script that aggregates power data from the on-board Baseboard Management Controller (BMC) and utilization telemetry from Prometheus. This dual-source approach ensures that the Energy Profile reflects the current heterogeneous state of the cluster. The full set of scale-down constraints is summarized in Table I.

As these criteria may be conflicting (e.g., minimizing energy consumption while preserving affinity preferences), node selection is formulated as a *Multi-Criteria Decision Analysis (MCDA)* problem. DREEM employs the Analytic Hierarchy Process (AHP) to derive criterion weights and TOPSIS to rank candidate nodes, selecting the highest-ranked one for the scaling action. As previously described, users can select a scaling profile during DREEM configuration according to their optimization goals, which dynamically adjusts the weights and leads to different node selection outcomes. After a node has been selected, all running services must be drained and restarted on another machine.

Instead, scaling up is generally less critical than scaling down; therefore, the selection model is simplified and con-

TABLE I
LIST OF CONSTRAINTS

Constraint	Type
Available resources (CPU, RAM), Taints, Node Affinity, Pod Affinity/Anti-affinity	HARD
Preferred Node Affinity, Preferred Pod Affinity/Anti-Affinity, Number of running services, Power Cycles, Energy Profile	SOFT

siders only a subset of constraints. In particular, DREEM accounts for the *Energy Profile*, favoring the most energy-efficient machines, and the *Power Cycles*, to balance node usage and mitigate hardware degradation.

At the end of the selection, the controller creates a new **NodeHandling** resource, identifying the target node.

C. Enforce the scaling

Adding or removing a node from a Kubernetes cluster is a complex process that involves multiple setup or teardown tasks (e.g., join/drain). DREEM automates these tasks through ClusterAPI [1], an open-source Kubernetes project designed for declarative cluster lifecycle management. Cluster API supports multiple infrastructure providers such as Metal3 [2] for bare-metal infrastructures.

The **NodeHandling** resource is responsible for contacting the target node and, in coordination with ClusterAPI, the controller initiates the procedures required for node provisioning or termination. Since DREEM leverages native K8s Resources, this approach can easily be extended to any number of worker nodes.

III. SETUP AND DEMONSTRATION

The objective of this work is to illustrate how DREEM is deployed in a Kubernetes cluster and how it selects the optimal node during scaling operations. The experimental setup consists of a Kubernetes cluster of one control plane + three worker nodes with 2x Intel(R) Xeon(R) Gold 6442Y CPUs and 512 GBs RAM each. DREEM's scaling algorithm is configured to maintain average CPU utilization between 40% and 75%, as illustrated by a preliminary experiment in Figure 2 which shows how the average CPU utilization changes with the number of nodes over time. To evaluate DREEM's behavior, dynamic CPU load is injected into the cluster to reproduce three scenarios: scale-up, scale-down, and no-scale. The load is generated using the µBench [9] benchmark software to deploy a Kubernetes-based microservice application. The microservices are scaled using a Horizontal Pod Autoscaler (HPA) [10] in response to incoming traffic.

Starting from a vanilla cluster, DREEM is configured and deployed, after which the scaling algorithm begins collecting monitoring data. As Locust generates traffic, cluster load increases until the average CPU usage exceeds the upper threshold, triggering the provisioning of an additional node. When the load subsequently drops below the lower threshold, DREEM initiates a scale-down operation, consolidating workloads onto the remaining nodes. During the demonstration, the

[1]https://cluster-api.sigs.k8s.io
[2]https://metal3.io

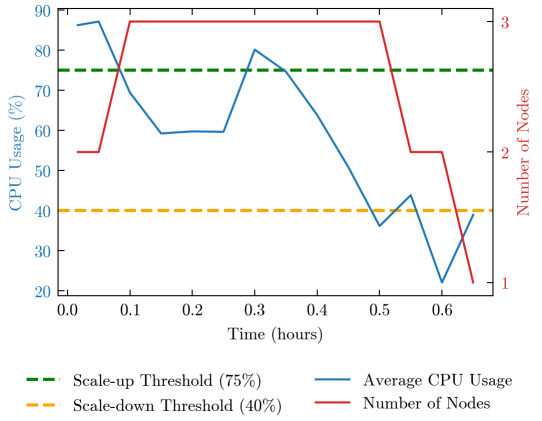

Fig. 2. Cluster average CPU usage vs number of active workers.

phases of the scaling process are detailed to highlight how the selected node is among the optimal candidates according to the defined constraints.

IV. CONCLUSIONS AND FUTURE WORKS

This work presented DREEM, a scaling framework for Kubernetes clusters designed to improve resource usage efficiency and reduce energy consumption. Future developments will leverage the modular DREEM architecture to refine enhanced node selection through service dependency awareness, improve the integration with hyperconverged storage systems, and incorporate service criticality to preserve overall Quality of Service during scaling operations.

REFERENCES

[1] Kubernetes clusterautoscaler. [Online]. Available: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler
[2] Karpenter. [Online]. Available: https://karpenter.sh/
[3] P. Sanjeevi and P. Viswanathan, "Workload consolidation techniques to optimise energy in cloud: review."
[4] Liting Hu, H. Jin, Xiaofei Liao, Xianjie Xiong, and Haikun Liu, "Magnet: A novel scheduling policy for power reduction in cluster with virtual machines," in *2008 IEEE International Conference on Cluster Computing*. IEEE, pp. 13–22. [Online]. Available: https://ieeexplore.ieee.org/document/4663751/
[5] B. Thurgood and R. G. Lennon, "Cloud computing with kubernetes cluster elastic scaling," in *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*. ACM, pp. 1–7. [Online]. Available: https://dl.acm.org/doi/10.1145/3341325.3341995
[6] I. Dimolitsas, D. Spatharakis, D. Dechouniotis, and S. Papavassiliou, "AHP4hpa: An AHP-based autoscaling framework for kubernetes clusters at the network edge," in *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pp. 2566–2571. [Online]. Available: https://ieeexplore.ieee.org/document/10001214/
[7] Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces. [Online]. Available: https://www.researchgate.net/publication/333790600_Who_limits_the_resource_efficiency_of_my_datacenter_an_analysis_of_Alibaba_datacenter_traces
[8] S. Malla and K. Christensen, "Choosing the best server for a data center: The importance of workload weighting," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. IEEE, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8710831/
[9] A. detti, l. funari, and l. petrucci, "bench: An open-source factory of benchmark microservice applications," in ieee transactions on parallel and distributed systems, vol. 34, no. 3, pp. 968-980, 1 march 2023, doi: 10.1109/tpds.2023.3236447.
[10] Horizontal pod autoscaling. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/