Dynamic Multi-Provider Cluster Autoscaling For The Computing Continuum

Dynamic Multi-Provider Cluster Autoscaling For The Computing Continuum

Stefano Galantino
stefano.galantino@polito.it
Politecnico di Torino
Torino, Italy

Riccardo Medina
riccardo.medina@polito.it
Politecnico di Torino
Torino, Italy

Attilio Oliva
attilio.oliva@polito.it
Politecnico di Torino
Torino, Italy

Fulvio Risso
fulvio.risso@polito.it
Politecnico di Torino
Torino, Italy

Giovanni Frattini
giovanni.frattini@eng.it
Engineering Ingegneria Informatica
Napoli, Italy

Abstract

Although existing Kubernetes cluster autoscalers dynamically adjust resources to match demand, they remain confined to the local provider's administrative domain, leading to inefficiencies, vendor lock-in, and are not appropriate for edge clusters. This paper introduces a novel multi-provider autoscaling architecture that enables the existing autoscaling paradigm to operate across multiple, independent federated clusters. The proposed solution leverages Liqo, an open-source framework enabling seamless cross-cluster federation, to expose remote compute capacity as virtual nodes within the local control plane, hence achieving transparent resource sharing. We design and implement a Liqo-aware Cluster Autoscaler that integrates remote resource discovery, enables policy-driven scaling decisions, and cost-aware placement across heterogeneous environments. Preliminary experimental evaluation demonstrates that our approach drastically improves resource utilization, especially in highly dynamic environments.

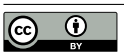
CCS Concepts

• **General and reference** → **Performance**; • **Computer systems organization** → **Peer-to-peer architectures**; **Cloud computing**; **Reconfigurable computing**; • **Information systems** → **Computing platforms**.

Keywords

Multi-cloud, Multi-cluster, computing continuum, cluster autoscaling, cluster federation

ACM Reference Format:
Stefano Galantino, Riccardo Medina, Attilio Oliva, Fulvio Risso, and Giovanni Frattini. 2025. Dynamic Multi-Provider Cluster Autoscaling For The Computing Continuum. In *the 3rd International Workshop on Middleware for the Computing Continuum (Mid4CC) 2025 (Mid4CC '25), December 15–19, 2025, Nashville, TN, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3774898.3778037



Mid4CC '25, Nashville, TN, USA

ACM ISBN 979-8-4007-2301-8/25/12
https://doi.org/10.1145/3774898.3778037

1 Introduction

Autoscaling is a core mechanism in cloud-native systems that dynamically adjusts resource allocation to match workload demand. By scaling compute capacity up or down, autoscaling minimizes over-provisioning, reduces operational costs, and enhances service availability [14]. In Kubernetes, autoscaling operates at multiple levels. Horizontal and Vertical Pod Autoscalers adjust, respectively, replica counts and pod resources while assuming that the cluster nodes are an invariant. Instead, Cluster Autoscaler (CA) manages the underlying infrastructure by adding or removing nodes when needed. Although conceptually different, both pod- and cluster-level autoscaling approaches coexist to support application execution. While pod autoscaling has been extensively studied, cluster autoscaling remains largely underexplored [3], especially in multi-cluster environments spanning remotely across multiple organizations. This gap stems from (*i*) technological limitations, since no fully functional multi-cluster autoscaling solutions currently exist, and from (*ii*) the inherent complexity introduced by the multi-tenant nature of the computing continuum [12, 15], which assumes the seamless integration of heterogeneous computing resources across geographical and operational boundaries [10]. Given that public cloud expenditure typically depends on node reservation time, keeping the minimum number of active nodes needed to satisfy the workload demand is essential to achieve cost and resource efficiency.

A Cluster Autoscaler (CA) monitors unscheduled pods and cluster utilization and triggers node provisioning when local capacity is insufficient (scale-out) or deprovisioning when wasteful (scale-in). The CA therefore transforms the cluster into an elastic resource: it creates or removes entire compute nodes in order to leave the scheduler with enough resources to allocate pods. However, current autoscaling mechanisms assume that new capacity can always be provisioned within the same site or cloud provider. When local capacity is exhausted (which is rather common in the case of clusters at the edge, which feature a limited number of physical servers), users face degraded availability or must rely on proprietary solutions tied to specific providers.

These locality and vendor-coupling constraints create both industrial and research challenges. Commercial multi-cluster bursting solutions are often closed and provider-specific, [1, 4], while most academic works focus on single-cluster models, neglecting cross-cluster coordination and policy enforcement [14, 18]. To address

this problem, we propose a federated autoscaling approach that transparently integrates the capacity available in remote clusters (e.g., running in the cloud or at the edge) within the resources available locally, thanks to a seamless extension of the Kubernetes control plane. This design allows autoscalers to reason about off-cluster resources without relying on provider-specific APIs.

In this paper, we leverage Liqo [5]—an open, production-grade technology enabling transparent cross-cluster resource sharing—to extend Kubernetes autoscaling beyond the original infrastructure boundaries. Our Liqo-based Cluster Autoscaler provides a twofold contribution: (*i*) it introduces the technological foundation for establishing a resource *federation*, enabling different entities or organizations to share computing resources according to the computing continuum paradigm widely discussed in the literature, and (*ii*) it integrates a cost-aware cluster autoscaling logic that can be easily adapted to overcome most of the limitations associated when handling distributed continuum infrastructures. Experimental results show that this approach reduces infrastructure costs by more than 60% while ensuring resource availability, automatic discovery, and avoiding provider lock-in.

This work is organized as follows. In Section 2 we describe the current state of the art of cluster autoscaling. In Section 3 we present the proposed solution based on Liqo, which is then evaluated in Section 4. Finally, Section 5 presents our concluding remarks.

2 State of the Art

The Kubernetes Cluster Autoscaler (CA), first introduced in 2016 [7], has progressively become the reference solution for dynamically resizing Kubernetes clusters according to workload demand. Through continuous development and integration efforts, the CA now supports all major public cloud providers, adapting its scaling mechanisms to the specific infrastructure and APIs offered by each platform[1]. Building upon the same principles, major vendors have released customized autoscaling frameworks, such as Google's GKE Cluster Autoscaler [4] and Amazon's EKS Karpenter [1]. However, although extremely effective when provisioning and resizing Kubernetes clusters based on the demand for computing resources, all of the commercial solutions proposed so far operate within the same hyperscaler domain, without the proper support for multi-provider infrastructures.

Academic research has largely focused on improving scaling algorithms, aiming for faster reactions to workload fluctuations and better cost–performance trade-offs [14]. Approaches are generally classified as **reactive** or **proactive** (predictive) [6]. Reactive controllers make scaling decisions based on real-time measurements (e.g., CPU, memory, or request rate) [9, 11], while proactive controllers forecast future workloads from historical data to pre-provision resources, effectively reducing provisioning delays [8].

Despite this extensive algorithmic work, few studies or open implementations address provider-agnostic autoscaling mechanisms leveraging remote resources over multiple clusters [17], especially over a federation where clusters are owned by multiple organizations and managed by different vendors [13]. To our knowledge, this topic has been only partially covered, with only one work [16] proposing to manage multi-cluster deployments with Karmada. However, Karmada centralized model requires all clusters to register under a single administrative authority, making it unsuitable for full-scale federations that involve multiple organizations.

Focusing on the infrastructure, the only suitable technologies supporting a multi-provider multi-cluster infrastructure are Open Cluster Management (OCM)[2] and Liqo [5]. Unlike the OCM model, Liqo enables *transparent* cluster resource extensions by leveraging resources from any already existing cluster. From the user perspective, remote capacity appears as a new (virtual) node within the local cluster, where pods can be scheduled as usual, without dealing with custom resources. This transparency makes Liqo particularly well-suited to setup the edge-cloud continuum, where workloads can seamlessly scale across heterogeneous and independently managed environments. However, Liqo lacks of automation, as the "cluster connectivity" must be performed (*i*) manually from the infrastructure operator, and (*ii*) based on a priori knowledge (i.e., it does not provide resource discovery). This work fills the above gap by defining the functional blocks required to implement the two features above, and it integrates the autoscaling logic with Liqo, hence bringing the capability to reshape a Kubernetes cluster within the whole computing continuum.

3 Liqo-based Cluster Autoscaler

This section presents the design and implementation of a custom Kubernetes *Cluster Autoscaler (CA)* provider based on Liqo. The proposed architecture extends the standard Kubernetes autoscaling mechanism to operate across federated clusters, dynamically acquiring computational resources from remote *peers* when local capacity becomes insufficient. This architecture builds on the concept of *resource federation*, an agreement among multiple organizations to share computing resources. This federation model advances the vision of the computing continuum, connecting cloud and edge resources under a unified, multi-provider infrastructure. The entire cluster autoscaling process includes three distinct stages, namely *federation enrollment*, *resource orchestration*, and *cluster scaling* (Figure 1). In the following, we outline each stage independently.

3.1 Federation Enrollment

The logical component responsible for managing the registration to the federation is the *Federation Broker*, an HTTP server handling cluster metadata, communication endpoints, and advertised resource availability. When joining the federation, each provider runs a dedicated *Federation Agent*, advertising the set of resources and capabilities that could be offered to the other members of the federation by means of a dedicated data structure (Figure 1, ①). Specifically, this data structure defines the configuration and resource description of a remote Kubernetes cluster within a federated or multi-cluster environment. It includes a Kubeconfig file containing encoded authentication and connection details to securely access the cluster (required by Liqo to extend the local cluster over the resource provided by a remote one), and a resources section advertising its computational capacity (e.g., 8 CPU cores, 16 GiB of memory, 100 pods, and one GPU) with the respective price per unit. Contextual metadata is provided through labels, such as

[1]List of supported providers by CA: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler

[2]https://open-cluster-management.io/

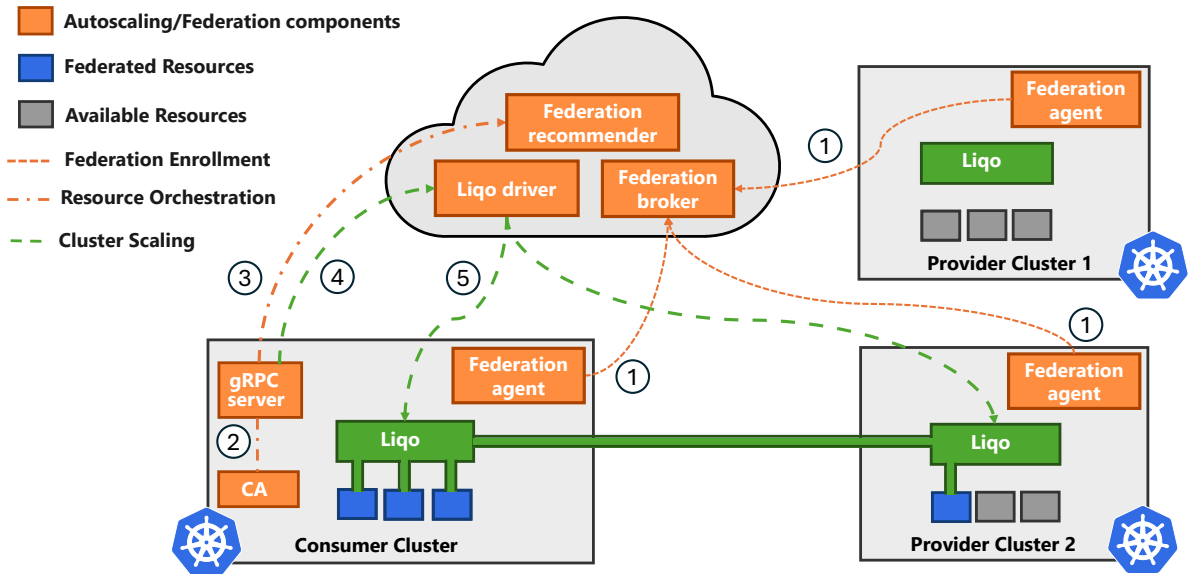


Figure 1: High-level architecture of the Liqo-enabled multi-provider Cluster Autoscaler, illustrating the interaction among the proposed components. Each cluster first registers its resources within the federation (1). When the CA initiates a scaling process, the gRPC server intermediates the request (2) and queries the available resources, which are ranked by cost by the *Federation Recommender* (3). Based on this information, the CA selects a scaling action and forwards the request to the *Liqo Driver* (4), which coordinates with the selected cluster Liqo deployment to provision or release the required resources on the target clusters (5).

geographical location and provider (e.g., private vs public cloud, etc.), enabling topology-aware scheduling and policy enforcement when deciding how to scale the local infrastructure. Finally, a dedicated networking section details network accessibility, specifying whether the cluster is directly reachable or located behind a NAT.

The *Federation Broker* is also in charge of authentication and authorization to grant resource visibility only to the desired entities in the federation, and manages the consistency in the resource usage, e.g., keeping track of the resources that are currently used, to prevent inconsistencies derived from simultaneous access to the peer information.

3.2 Resource Orchestration

The Kubernetes Cluster Autoscaler (CA) is responsible for dynamically adapting the cluster size according to the current workload demand, hence optimizing resource consumption and operational costs. When a new pod cannot be scheduled due to insufficient resources, the CA triggers a *scale-out* operation by requesting new nodes for the cluster. Conversely, underutilized nodes can be released through *scale-in* operations.

Our implementation follows an *out-of-tree integration* model according to the CA specifications, in which a (vanilla) Kubernetes CA connects to a standalone *gRPC server* (Figure 1, ②), acting as an intermediary towards the federated resource controller. The gRPC server exposes the set of mandatory methods required by the CA, which allow the autoscaler to retrieve the state of available node groups, modify their target size, and simulate scale-out operations. It provides the CA with a unified view of available resources, as supplied by the *Federation Recommender*, which ranks resources by cost to support optimal placement decisions (Figure 1, ③). Although the CA is the resource orchestrator and makes the final scaling decision, it can be influenced by assigning custom (monetary or logical) cost values —assigned to available resources by an external component, the Federation Recommender. This design preserves modularity and maintainability while avoiding modifications to the CA source code, and it allows future extensions such as predictive autoscaling without altering other components.

3.3 Cluster Scaling

Our architecture integrates the autoscaling mechanism with Liqo, a Kubernetes add-on that enables (local) a cluster to be extended over the resources provided by another (remote) cluster, hence providing seamless resource sharing across multiple clusters. When the CA detects resource scarcity in the source cluster, our Liqo-based extension requests additional capacity from remote clusters using the resource sharing mechanism natively provided by this technology. Upon successful peering, Liqo exposes remote resources as virtual nodes within the local cluster, enabling transparent cross-cluster scheduling while preserving Kubernetes-native semantics.

This final stage is carried out by the *Liqo Driver*, an HTTPs endpoint that serves as the execution backend for scaling operations, managing the lifecycle of the nodes being added or removed (Figure 1, ④⑤). It exposes a REST interface consumed by the gRPC server and interacts with the Liqo control plane to establish or tear down peerings according to CA requests. The modular nature of

Table 1: Testbed setup including one local node and four different nodes for the CA to operate on.

Category	N. nodes	CPU	Memory	Cost	Provider
Low cost	3	2 cores	4 GB	1 $/min	Pr1
High cost	1	4 cores	4 GB	5 $/min	Pr2

this component includes a set of handlers that manage the translation between HTTPS requests and internal data structures, a set of backend functions that are in charge of the scaling operations, and proper data type definitions. This design allows future, possibly customized, extensions to additional cloud or federation providers if needed. Specifically, based on the strategy (i.e., scale-in or scale-out), it interacts with Liqo to either tear down an existing Liqo peering or establish a new peering. It offers full support for specialized hardware components (e.g., GPUs) and it can trigger the Liqo peering even if the provider cluster is behind a NAT, making the solution compliant also for the edge or 5G-enabled settings. The *Liqo Driver* can be deployed within a single cluster to leverage federated resources locally or in a centralized cloud environment to coordinate multiple clusters, preventing cross-cluster interference and contention for the available federated resources.

4 Evaluation

This section evaluates the proposed Liqo-based Cluster Autoscaler, focusing on the dynamism when scaling in multi-cloud multi-provider environments.

4.1 Testbed Setup

The setup used for the evaluation includes a *federation* of 5 nodes, based on a local resource consumer (always operational) and four remote resource providers, each implemented as a VM, and each advertising the respective resource availability to the *Federation broker*. The different nodes are further classified into two major categories depending on the amount and the cost of the advertised resources. Table 1 summarizes the capabilities and resource cost (provided by the *Federation recommender*) of each category used in this evaluation.

A dynamic workload is submitted to the infrastructure to trigger the scale-in and scale-out requests from the CA and the subsequent Liqo peering to reshape the size of the local cluster. Specifically, the workload depicted in Figure 2 is replicated by creating a deployment based on `stress-ng`[3] to simulate CPU and memory-demanding tasks, with the number of replicas linearly increasing or decreasing depending on the desired behavior.

Three different configurations are then tested, namely (*i*) *Baseline*, accounting for a scenario in which the autoscaling is not enabled (i.e., resources are reserved beforehand and never freed for the entire test), (*ii*) *CA + Liqo (cost-unaware)* to assess the proposed integration of the cluster autoscaling logic with the multi-provider support of Liqo in a cost-unaware setting, i.e., without providing cost-based recommendations to the CA for the node selection process, and (*iii*) *CA + Liqo (cost-aware)* to assess the full potential of the autoscaling logic when prioritizing less expensive nodes. Notably, tests on vanilla Cluster Autoscaler are omitted, as neither it nor its plugins support scaling clusters with nodes from multiple resource providers, leaving no additional test cases to consider.

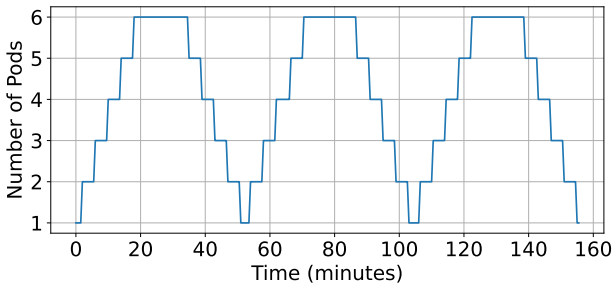


Figure 2: Load shape of the submitted workload represented as the number of pods running simultaneously in the autoscaled cluster.

With respect to the relevant software, we used the Kubernetes CA version 1.30.4, Liqo version 1.0.0, and the autoscaler connector for Liqo described in this paper in the latest version. [4] Furthermore, cluster metrics were collected using Prometheus to provide insights into the scaling process, focusing on the number of nodes of the cluster at any time, the CPU utilization, and the scaling cost based on the selected autoscaling method.

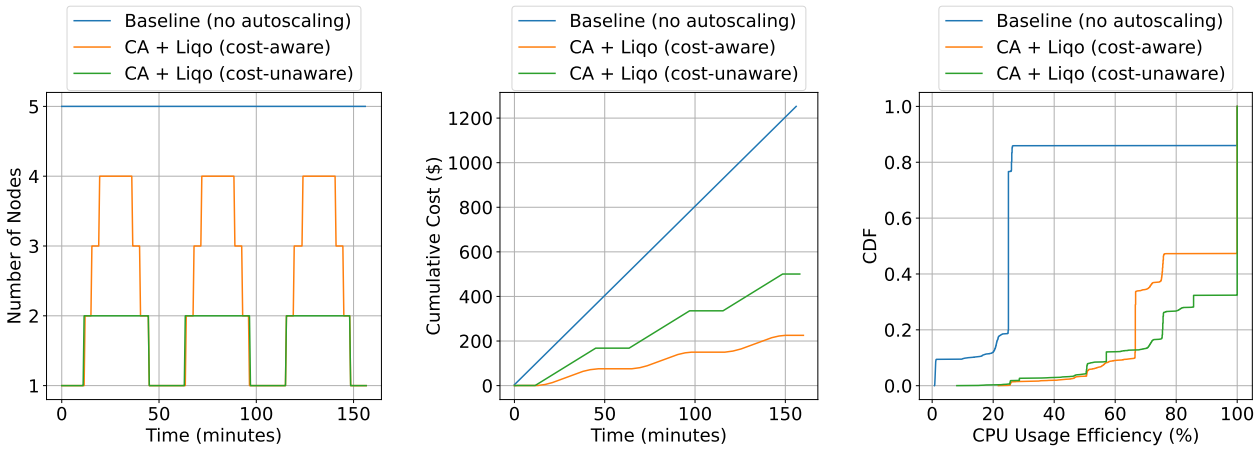
4.2 Resource Usage Effectiveness

The dynamic cluster reconfiguration highlights the capability of the CA to reshape the size of the Kubernetes cluster using Liqo according to the observed workload, borrowing resources from other entities in the continuum, either from the public or private cloud, depending on availability.

Figure 3a illustrates the number of worker nodes connected to the local Kubernetes cluster over time. Except for the baseline case, in which the number of worker nodes is fixed at five throughout the entire test, both the cost-aware and cost-unaware versions of the CA correctly resize the local cluster in response to the demand of the deployed applications. Specifically, the cost-unaware configuration adopts a more conservative approach, never exceeding two nodes and favoring larger but more expensive instances, whereas the cost-aware allocation trades off by selecting smaller yet cheaper nodes. Regardless of the configuration, these results demonstrate that the CA retains full visibility over the available resources across the continuum substrate, operating as if it were deployed within a single-provider infrastructure while extending its management capabilities toward remote clusters. It is also worth mentioning that the time required for the Liqo peering to be established (i.e., the time needed to add a new node) is in the order of a few seconds, allowing the infrastructure to scale even faster than for some hyperscalers. This is because, unlike most cloud infrastructures, in which worker nodes are VMs created and initialized on demand, in our case, remote nodes are operational Kubernetes clusters ready to accept incoming requests, enabling a much faster response time from the CA in response to fluctuating resource usage patterns.

[3] Stress test suite. GitHub repository: https://github.com/ColinIanKing/stress-ng

[4] Github repository: https://github.com/rmedina97/liqo-autoscaling-system



(a) Number of worker nodes. **(b) Cumulative infrastructure cost.** **(c) CPU usage efficiency.**

Figure 3: Dynamic cluster autoscaling in federated infrastructures. Figure 3a details the number of worker nodes joining the local Kubernetes cluster. Figure 3b details the cost resulting from the usage of the resources purchased from the continuum federation. Finally, Figure 3c represents the CDF of the CPU usage efficiency computed as the ratio between the amount of resources reserved and the actual usage.

Focusing on the effectiveness of the autoscaling, Figure 3c depicts the CPU usage efficiency across the entire federation. This efficiency metric η is defined as:

$$\eta = 100 \times \frac{\text{CPU}_{\text{used}}}{\text{CPU}_{\text{available}}}$$

The efficiency value ranges from 0 to 100%, where 0% represents a node offering resources that remain unused, and 100% indicates full utilization of the reserved resources. Moreover, a value of 100% is also assigned to nodes that are currently not used, hence considered available to other entities within the federation, as they can potentially be leveraged elsewhere in the continuum. As shown in Figure 3c, both autoscaling approaches manage to drastically improve the efficiency in terms of resource usage compared to the static baseline by triggering the Liqo peering only when needed. In this respect, the cost-unaware approach seems to provide slightly better performance thanks to a much more aggressive consolidation by favoring larger but more expensive nodes. In other words, the median CPU usage efficiency (i.e., drawing an imaginary line for y=0.5, omitted in Figure 3c for the sake of clarity) is about 25% in the baseline, compared to 75% in the cost-aware and almost 100% in the cost-unaware strategy.

Finally, Figure 3b illustrates the cost of running the entire Kubernetes cluster within our federated multi-provider setup. Both autoscaling approaches achieve a cost reduction of at least 60% compared to the baseline by removing worker nodes when not needed and favoring cheaper nodes in the case of the cost-aware configuration. These results demonstrate that, although in this specific test the cost represents a monetary value, it can be generalized and adapted to different use cases, leading to different outcomes. Indeed, our contribution enables the integration of customized cost logics (e.g., based on network latency, throughput, ...) to guide the cluster autoscaling process toward a globally optimal objective for the infrastructure. As a result, our contribution is a key enabler to build a continuum-native infrastructure able to dynamically adapt its allocated resources to a variable workload.

5 Conclusions

This paper presents a novel approach to extend the Kubernetes Cluster Autoscaler beyond single-provider boundaries, enabling dynamic, cost-aware, and policy-driven scaling decisions across federated clusters. By integrating the autoscaling logic with Liqo, we demonstrated the capability to transparently acquire remote computational resources from other domains in the computing continuum while maintaining full compatibility with Kubernetes-native semantics.

The proposed architecture relies on a modular, out-of-tree design composed of four key components—*Federation Broker*, *gRPC server*, *Federation Recommender*, and *Liqo Driver*—which together enable seamless cross-cluster coordination without modifying the core autoscaling logic. Experimental results confirm that the Liqo-aware Cluster Autoscaler can dynamically reshape the local infrastructure to follow workload demand, reducing both resource waste and operational costs compared to static or single-cluster setups. Moreover, the integration of cost-awareness allows for adaptive decision-making that optimizes not only performance but also economic efficiency in multi-provider environments.

Future work will explore the integration of predictive autoscaling models and context-aware policies to further enhance responsiveness and efficiency, and mechanisms such as the REAR (Resource Advertisement and Reservation) protocol [2] to negotiate the acquisition of remote resources. Additionally, extending the cost model to incorporate network metrics such as latency and bandwidth

will enable more fine-grained placement decisions throughout the continuum. Overall, this work demonstrates that federated autoscaling can serve as a cornerstone for realizing the vision of a truly dynamic, multi-provider computing continuum.

Acknowledgments

This work was partly supported by the project IPCEI-CIS AVANT - CUP B89J24002920005 - Funded by the European Union under the NextGenerationEU initiative, and by the Italian Ministry of University and Research with the project NEWTON (NEtWork programmabiliTy tOols at haNds) PRIN 2022 NEWTON - ID: 2022ZA8T22 - CUP: D53D23001630006. Finally, Attilio Oliva acknowledges the support from TIM S.p.A. through the PhD scholarship.

References

[1] Amazon Web Services. 2025. Scale cluster compute with Karpenter and Cluster Autoscaler (EKS). AWS EKS documentation. https://docs.aws.amazon.com/eks/latest/userguide/autoscaling.html

[2] Stefano Galantino, Elisa Albanese, Nasir Asadov, Stefano Braghin, Francesco Cappa, Andrea Colli-Vignarelli, Amjad Yousef Majid, Eduard Marin, Jacopo Marino, Lorenzo Moro, Liubov Nedoshivina, Fulvio Risso, Domenico Siracusa, Antonio Skarmeta, and Luca Zuanazzi. 2024. Building the Cloud Continuum with REAR. In *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. 67–72. doi:10.1109/NetSoft60951.2024.10588885

[3] Panagiotis Gkonis, Anastasios Giannopoulos, Panagiotis Trakadas, Xavi Masip-Bruin, and Francesco D'Andria. 2023. A survey on IoT-edge-cloud continuum systems: Status, challenges, use cases, and open issues. *Future Internet* 15, 12 (2023), 383.

[4] Google Cloud. 2025. About GKE cluster autoscaling / Cluster Autoscaler (GKE). Google Cloud documentation. https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler

[5] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. 2022. Computing without borders: The way towards liquid computing. *IEEE Transactions on Cloud Computing* 11, 3 (2022), 2820–2838.

[6] Byeonghui Jeong and Young-Sik Jeong. 2025. Autoscaling techniques in cloud-native computing: A comprehensive survey. *Computer Science Review* 58 (2025), 100791. doi:10.1016/j.cosrev.2025.100791

[7] Kubernetes Project. 2016. Autoscaling in Kubernetes. Kubernetes blog. https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/

[8] Zhihui Lu, Xueying Wang, Jie Wu, and Patrick CK Hung. 2017. InSTechAH: Cost-effectively autoscaling smart computing hadoop cluster in private cloud. *Journal of Systems Architecture* 80 (2017), 1–16.

[9] Tarek Menouer, Christophe Cérin, and Patrice Darmon. 2024. Reactive autoscaling of kubernetes nodes. In *Proceedings of the 4th workshop on flexible resource and application management on the edge*. 31–38.

[10] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. 2022. Cloud continuum: The definition. *IEEE Access* 10 (2022), 131876–131886.

[11] Vladimir Podolskiy, Anshul Jindal, and Michael Gerndt. 2018. IaaS Reactive Autoscaling Performance Challenges. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 954–957. doi:10.1109/CLOUD.2018.00144

[12] Berat Can Şenel, Maxime Mouchet, Justin Cappos, Timur Friedman, Olivier Fourmaux, and Rick McGeer. 2023. Multitenant containers as a service (CAAS) for clouds and edge clouds. *IEEE Access* 11 (2023), 144574–144601.

[13] Galantino Stefano, Marino Jacopo, Risso Fulvio, Braghin Stefano, Nedoshivina Liubov, and Siracusa Domenico. 2025. Edge-to-Cloud Continuum Made Real. *Computer* (2025). doi:10.1109/MC.2025.3607210 In press.

[14] Mulugeta Ayalew Tamiru, Johan Tordsson, Erik Elmroth, and Guillaume Pierre. 2020. An experimental evaluation of the kubernetes cluster autoscaler in the cloud. In *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 17–24.

[15] Mauro Tortonesi. 2025. The Compute Continuum: Trends and Challenges. *Computer* 58, 03 (2025), 105–108.

[16] Minh-Ngoc Tran and Younghan Kim. 2025. Hybrid Resource Quota Scaling for Kubernetes-Based Edge Computing Systems. *Electronics* 14, 16 (2025). doi:10.3390/electronics14163308

[17] Minh-Ngoc Tran, Dinh-Dai Vu, and Younghan Kim. 2022. A Survey of Autoscaling in Kubernetes. In *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*. 263–265. doi:10.1109/ICUFN55119.2022.9829572

[18] Mingming Wang, Dongmei Zhang, and Bin Wu. 2020. A cluster autoscaler based on multiple node types in kubernetes. In *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Vol. 1. IEEE, 575–579.