Towards a Decentralised Application-Centric Orchestration Framework in the Cloud-Edge Continuum

Amjad Ullah[§], Andras Markus[†] Hacı İsmail Aslan[*], Tamas Kiss[#],
Jozsef Kovacs[+], James Deslauriers[#], Amy L. Murphy[‡], Yiming Wang[‡], Odej Kao[*]

[§]A.Ullah@napier.ac.uk, Edinburgh Napier University, Edinburgh, United Kingdom
[†]andras.markus@frontendart.com, FrontEndART Software Ltd., Szeged, Hungary
[*]{aslan, odej.kao}@tu-berlin.de Technische Universität, Berlin, Germany
[#]{T.Kiss, J.Deslauriers}@westminster.ac.uk, University of Westminster, London, United Kingdom
[+]jozsef.kovacs@sztaki.hu, Institute for Computer Science and Control (SZTAKI), Budapest, Hungary
[‡]{murphy, ywang}@fbk.eu, Fondazione Bruno Kessler, Trento, Italy

***Abstract*—Managing complex distributed applications in the Cloud-Edge continuum, including deployment on diverse resources and runtime operations, presents significant challenges. Orchestrators play a key role by automating resource discovery, optimisation, deployment, and lifecycle management while ensuring system performance. This paper introduces Swarmchestrate, a decentralised, application-centric orchestration framework inspired by self-organising Swarms. Our initial findings, based on the implementation in a Cloud-Edge simulator, demonstrate Swarmchestrate's potential, offering insights into resource coordination and optimised allocation for scalable systems.**

***Index Terms*—Cloud-Edge, Orchestration, Decentralised, Resource selection, Swarm computing, Self-organisation**

I. INTRODUCTION

The rapid growth of Cloud-Edge ecosystems has reshaped how distributed applications provision, and manage resources. Orchestration solutions are commonly used for this purpose and to handle associated challenges such as ensuring seamless access and coordination between heterogeneous cloud, fog, and edge resources, optimising conflicting QoS goals (e.g., cost, performance, energy, etc.), and addressing scalability, adaptability, and efficient monitoring of workloads [1], [2]. Addressing these challenges has drawn significant attention toward developing orchestration solutions [3], [4]. These solutions, based on their control topology, can be classified into centralised and decentralised.

Centralised approaches are easy to implement and offer consistent decision-making; however, they face issues such as scalability, a single point of failure, performance bottlenecks, and cybersecurity risks. Decentralised approaches address these issues by providing multiple decision-making entities (orchestrators) distributed across the continuum. Several studies (see Section II) have explored such strategies, demonstrating their potential to manage the complexities of the continuum. In the same realm, this paper presents Swarmchestrate, a decentralised application-centric orchestration framework inspired by the self-organisation of Swarms. More specifically, our key contributions include the design of a novel decentralised orchestration architecture based on our earlier work [5], followed by a simulation-based implementation covering the overall application deployment process. Lastly, a thorough evaluation of the proposed approach to demonstrate its applicability.

II. RELATED WORK

Several studies explored hierarchical architectures. For example, mF2C [6] employs an N-layered model, from edge (Layer-N) to cloud (Layer-0), with agents at each layer collaborating on service execution while prioritising lower layers to reduce latency. Oakestra [7] uses a two-layered approach, where cluster orchestrators manage local resources, and a root orchestrator oversees multiple resource clusters under separate administration.

Other studies have explored P2P models. For example, HYDRA [8] establishes a P2P overlay where each node functions as both a resource and an orchestrator, managing applications at varying levels of granularity. Caravela [9] follows a similar approach but incorporates a market-oriented model, incentivising volunteer resources. Other perspectives include a dedicated orchestrator for each application, as proposed by Castellano et al. [10]. EPOS Fog is introduced by Zeinab et al. [11], which is a multi-agent system where each node acts as an agent, determining service deployment within its neighbourhood. Lastly, Zolton [12] proposed partitioning the

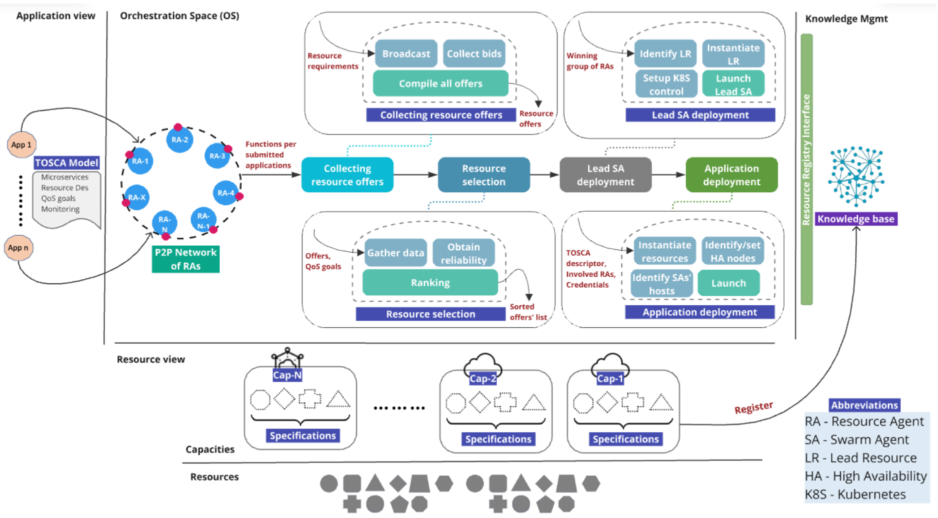


Fig. 1: Swarmchestrate architecture

infrastructure into isolated segments, called fog colonies, enabling independent optimisation using either isolated or shared resources.

Our approach against existing solutions stands out for its highly decentralised, application-centric focus and self-organising capabilities, enabling higher adaptability and resilience. Unlike hierarchical or P2P models, we adopt a hybrid architecture. The hierarchical aspect stems from its two-layered structure: the interface—a dynamic network of distributed resource agents—and application spaces composed of Swarm agents managing individual applications. The operational mechanisms at each layer are designed around a P2P model.

III. SWARMCHESTRATE: PROPOSED APPROACH

Figure 1 illustrates the Swarmchestrate application deployment process, organised into four sections. The **Application view** allows operators to submit applications in the TOSCA format[1]. The **Resource view** features a two-layered structure: *Resources*, representing computational resources from various cloud and edge providers (e.g., Amazon, Microsoft), and *Capacity*, a logical grouping of resources within the Swarmchestrate ecosystem, which must be registered for discovery and deployment. The **Knowledge Management** component acts as a distributed knowledge base, managing resource descriptions, interactions, discovery, and trust.

The **Orchestration Space (OS)** leverages Decentralisation, Swarms, and Intelligence for efficient, optimised, and trusted orchestration. Decentralisation allows to operate without central control. Swarm computing enables dynamic, cooperative management of applications; and Intelligence, driven by machine learning and optimisation algorithms, informs resource selection and decision-making. The following subsections detail the OS component.

A. Application

Swarmchestrate supports microservices-based applications, described in TOSCA, covering four key aspects: (a) The details of application components such as container images, environment variables, etc; (b) The specific needs for application resources, such as cloud/edge instances, instance types, and hardware limits (CPU/RAM/Storage); (c) The desired QoS specifications, including performance, cost, energy efficiency, trust, placement, etc; and (d) the specification of custom metrics to be monitored by Swarmchestrate.

B. Resource Agent

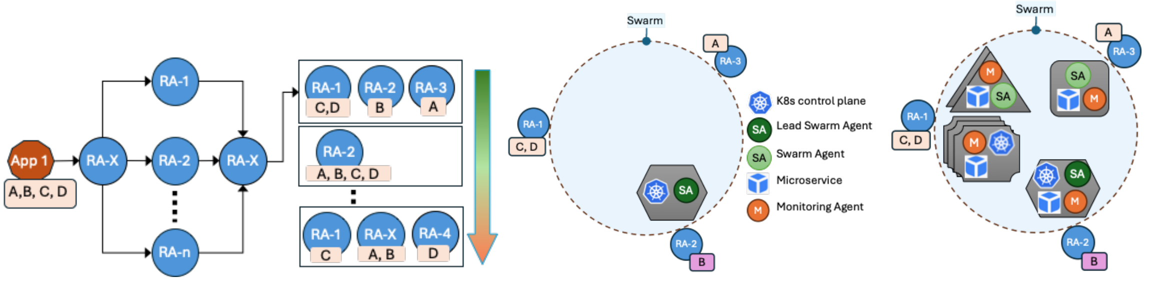
The Resource Agent (RA) manages one or more Capacities, providing access to their resources. Additionally, by collaboration with other RAs, it facilitates the discovery of suitable resources across the resource stack for submitted applications. In Swarmchestrate, an RA is instantiated, when the Capacity provider registers the Capacity resources with attributes like processing power, memory, hardware type, VM instances, pricing, locality, and energy metrics. Once instantiated, the RA connects to other RAs via a P2P network, forming a decentralised OS interface. The TOSCA description is submitted to the interface, where an RA receives it and initiates the deployment process, outlined in the next section.

C. Overall deployment process

We illustrate the deployment process using a simple example, featuring an application (app1) comprising four microservices, having four resource requirements (A, B,

---

[1] https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html



(a) Resource offers collection and ranking (b) Lead resource (c) Post-deployment

Fig. 2: Illustrative example of application deployment in Swarmchestrate

C, and D). Upon receiving the application (Figure 2a), RA-X—selected randomly, however, can be based on any particular logic—initiates the following steps:

1) Collecting resource offers: RA-X broadcasts resource requirements to all available RAs, requesting offers (Figure 2a). Each RA evaluates its Capacity via Knowledge Management and classifies its coverage as Partial (some requirements met), Full (all met), or Zero (none met). RAs respond with their coverage, and RA-X compiles unique groups of potential offers.

2) Resource selection: This step selects the optimal resource set from previous offers to maximise QoS goal fulfilment. This optimal set, once configured, forms the **Swarm** that will serve the application. The inputs for this process consist of the resource offers from the previous step, the application QoS goals from the TOSCA description, and the dynamically obtained reliability metrics (e.g. failure frequency, availability, resource accuracy, etc.), representing the impact on the achievement of QoS goals, for each resource offer. An optimisation algorithm (Section IV-A) ranks offers, selecting the top-ranked set (e.g., RA-1, RA-2, RA-3 in Figure 2a) for deployment.

3) Lead SA deployment: This step initiates the Swarm formation by deploying the lead Swarm Agent (SA). Multiple SAs operate at the Swarm level, ensuring self-organisation and reconfiguration. The lead SA—first instance of SA—assembles the Swarm (Section III-C4). Deployment starts by RA-X using the following substeps: (a) RA-X selects the Lead Resource (LR) from the top-ranked offer to host the lead SA and Kubernetes control plane, considering factors like CPU, storage, and networking; (b) Once identified, the LR is instantiated (e.g., a cloud resource is dynamically created); (c) The Kubernetes control plane is set up on the LR for container orchestration; (d) Lastly, the lead SA is initiated to assemble the Swarm (see next section). After these steps, the system reaches the state in Figure 2b, where only the LR (type B from RA-2) is active, hosting the Kubernetes control plane and lead SA within the Swarm.

TABLE I: Experimental settings for simulation

Parameter	**Value**
Components	Compute:3 Storage:1
CPU (pc)	1 — 6
RAM (GB)	1 — 6
Storage (GB)	1 — 10
Image size (MB)	1 — 500
Instances	1 — 3
Msg size (KB)	2

(a) Applications

Parameter	**Value**
Location	EU, US
Provider	AWS, Azure
CPU (pc)	16 — 100
RAM (GB)	16 — 100
Storage (GB)	16 — 100
Idle power (W)	150 — 225
Max power (W)	500 — 3500
Latency (ms)	15 — 100
Price (€/hour)	0.025 — 25

(b) Capacities (Nodes)

4) Application deployment: The Lead SA finalises Swarm formation and deploys the application using the TOSCA description, RA details, and credentials. More specifically, (a) the Lead SA, with the involved RAs, instantiates the remaining resources, and makes them part of the cluster setup on the LR; (b) to ensure high availability (HA), additional resources are selected using the same criteria as LR, and HA configurations are applied; (c) to ensure self-organisation, a group of SAs are identified to be hosted alongside application components; (d) additional SAs, application components, and monitoring agents are deployed via the Kubernetes scheduler by the lead SA. Completion of these steps take the system to the state in Figure 2c, with all required resources (greyed boxes) integrated into the Swarm, running four microservices of app1.

IV. EVALUATION

To evaluate the feasibility and performance of the proposed framework, we extended DISSECT-CF-Fog [13], a widely used discrete event simulator[2] known for its realism and customisability in Cloud-Edge simulations, with the necessary constructs (e.g., RA, Capacity) to support Swarmchestrate.

A. Application, RA, Capacities, and resource selection

The simulator accepts the application in TOSCA format and supports two types of application components:

---

[2] https://github.com/sed-inf-u-szeged/DISSECT-CF-Fog

Compute, having a container image, and hardware (CPU and memory) limits for instantiation; and *Storage* with the size of the allocated partition only. The RA is modelled as a virtualised resource, whereas, the Capacities are represented as the physical resources. For our experimentation, 8 Capacities each represented by one RA and six applications are utilised to assess system behaviour. Table Ia and Ib present the interval-based specification used for applications and Capacities respectively.

Upon application submission, an RA (e.g., RA-X in Figure 2a) manages deployment. RA-X broadcasts the request, and each receiving RA evaluates it using a first-fit strategy, sorting components by CPU requirements—50% in ascending and 50% in descending order—to balance allocation. Components are mapped based on available capacity, reserving resources upon a successful match. RA-X then compiles unique offers, ensuring each component appears once per combination. These offers are then ranked using the following methods based on the submitted application's QoS objective, consisting of four attributes including latency, cost, bandwidth, and energy consumption. Each of these attributes is defined with a priority reflecting the application owner's preferences. Additionally, reliability, as explained in Section III-A, is also considered in decision-making.

1) Cost Function: This approach calculates a cost value for each offer and then ranks all in descending order of overall cost. For each offer, all QoS attributes are first normalised to a 0–1 range for comparability as can be seen from Equation (1) for raw data r_q of each QoS attribute $q \in Q$. For attributes like bandwidth, values are inverted to reflect their desirability. Each normalised value is then weighted by its QoS priority p_q, and the total cost for an offer i is calculated using (2).

$$\text{nor}_q = \begin{cases} 0, & \text{if } \mathit{max}(r_q) = \mathit{min}(r_q) \\ \frac{r_q - \min(r_q)}{\max(r_q) - \min(r_q)}, & \text{otherwise} \end{cases} \tag{1}$$

$$\text{total\_cost}_i = \sum_{q \in Q} p_q \cdot \text{nor}_{q,i} \tag{2}$$

Lastly, to incorporate reliability (R) into ranking, two approaches are used: (A) **Additive**, where R is subtracted from the total cost ($\text{total\_cost}_i - R_i$), lowering costs for more reliable offers; and (B) **Multiplicative**, where R scales the total cost ($\text{total\_cost}_i = (1 - R_i) \cdot \text{total\_cost}_i$), adjusting cost proportionally to reliability.

2) Borda Voting: This approach ranks offers based on their relative positions across QoS attributes. Each attribute is ranked independently (e.g., bandwidth in descending order, latency in ascending order), and offers receive scores based on their rank, with ties sharing the highest score for their position. Scores are then weighted by attribute priorities to determine the final ranking. Lastly, reliability—either as an additive or multiplicative approach—is incorporated into the ranking process. More formally, Equation (3) defines the Borda score S_i of an offer i, where $\text{score}_q(i)$ and $\text{score}_R(i)$ represent the Borda scores for QoS attribute q and reliability R, respectively; whereas, Equation (4) and (5) represents the final Borda scores with reliability as additive and multiplicative factors.

$$S_i = \sum_{q \in Q} p_q \cdot \text{score}_q(i) \tag{3}$$

$$S_i = \text{score}_R(i) + \sum_{q \in Q} p_q \cdot \text{score}_q(i) \tag{4}$$

$$S_i = R_i \sum_{q \in Q} p_q \cdot \text{score}_q(i) \tag{5}$$

B. Application deployment

Once the ranking is performed, RA-X deploys the application using the top-ranked offer. Next, RA-X selects the lead resource (LR) based on the highest CPU core count as the selection criteria (Section III-C3). Furthermore, to simulate real-world deployment, we integrate a Docker Hub-like registry with 1000 Mbps bandwidth in DISSECT-CF-Fog for storing and transferring container images. Once the LR is chosen, the images are deployed and associated capacities are marked as *allocated*. Lastly, to assess the long-term impact of the deployment decision, we ran each *Compute* component at full CPU capacity for 30 minutes.

C. Results

The evaluation assessed Swarmchestrate's ability to handle six simultaneous applications while varying QoS priorities. We evaluated six strategies: four where a single QoS attribute had priority 1.0 while others were set to 0.1, one with equal priorities, and one with random assignment. For comparison, the following metrics are considered: 1) *Simulation Time*, duration from submission to all tasks completion; 2) *Total Price*, resource costs based on hourly rates; 3) *Avg. Deployment Time*, time from submission to deployment, influenced by latency and bandwidth; and 4) *Total Energy*, cumulative energy consumption per node.

Table II presents the results, with the best values highlighted in green and the worst in red. The proposed ranking algorithm consistently excelled when a priority value of 1.0 was assigned (rows 1-8). For instance, a price-aware strategy effectively reduced operating costs. While the Cost function method produced some worse results (red columns), it generally outperformed the Borda method in meeting priority-specific objectives. The Equal strategy balanced cost and deployment efficiency, while the bandwidth-aware strategy outperformed the latency-aware approach, underscoring the critical role of bandwidth in deployment.

TABLE II: Simulation results for different priorities and resource selection methods

Priority	Method	Simulation Time (min)	Total Price (EUR)	Avg. Deployment Time (min)	Total Energy (KWh)
Energy	Borda	37.946	0.053	3.636	2.232
	Cost	37.946	0.046	3.779	2.170
Price	Borda	35.044	0.015	2.451	2.292
	Cost	37.946	0.032	3.645	2.211
Latency	Borda	34.641	0.077	1.352	2.360
	Cost	34.562	0.079	1.285	2.363
Bandwidth	Borda	34.385	0.075	1.288	2.316
	Cost	32.175	0.115	0.968	2.260
Equal	Borda	34.318	0.036	1.620	2.229
	Cost	34.562	0.076	1.285	2.363
Random		34.437	0.082	1.365	2.310

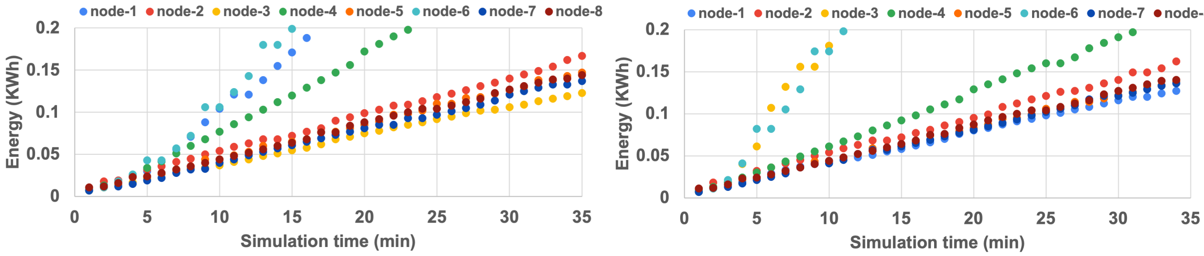


Fig. 3: Energy consumption per node with energy priority (left) and latency priority (right)

Figure 3 illustrates accumulated energy consumption per node for energy- and latency-aware scenarios. Measurements cover the period from application submission to completion, excluding cold start and infrastructure setup. In the energy-aware approach, CPU-heavy tasks begin after five minutes, whereas the latency-aware strategy enables faster deployment (*Avg. Deployment Time* in Table II), with tasks starting after three minutes.

V. CONCLUSION

This study presented Swarmchestrate, a decentralised orchestration framework for Cloud-Edge applications. By adopting an application-centric approach, it tackles scalability, resource heterogeneity, self-organisation, and multi-QoS balancing. Simulation results demonstrated its effectiveness in seamless deployment across diverse providers. Ongoing work focuses on implementing self-organisation for runtime reconfiguration, with plans to prototype the framework on four real-world industry use cases, further establishing its scalability and applicability for next-generation distributed systems.

ACKNOWLEDGMENT

This work was funded by the EU Horizon programme within the Swarmchestrate project (no. 101135012).

REFERENCES

[1] G. J. Hu and T. Vardanega. An architectural view on the compute continuum: Challenges and technologies. *Available at SSRN 4328069*, 2023.

[2] A. Ullah, H. Dagdeviren, R. C Ariyattu, J. DesLauriers, T. Kiss, and J. Bowden. MiCADO-Edge: Towards an Application-level Orchestrator for the Cloud-to-Edge Computing Continuum. *Journal of Grid Computing*, 19(4):1–28, 2021.

[3] A. Ullah et al. Orchestration in the cloud-to-things compute continuum: taxonomy, survey and future directions. *Journal of Cloud Computing*, 12(1):135, 2023.

[4] S. Böhm and G. Wirtz. Cloud-edge orchestration for smart cities: A review of kubernetes-based orchestration architectures. *EAI Endorsed Transactions on Smart Cities*, 6(18):e2–e2, 2022.

[5] T. Kiss et al. Swarmchestrate: Towards a fully decentralised framework for orchestrating applications in the cloud-to-edge continuum. In Leonard Barolli, editor, *Advanced Information Networking and Applications*, pages 89–100, Cham, 2024. Springer Nature Switzerland.

[6] X. Masip-Bruin et al. Managing the cloud continuum: Lessons learnt from a real fog-to-cloud deployment. *Sensors*, 21(9):2974, 2021.

[7] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. Oakestra: A lightweight hierarchical orchestration framework for edge computing. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 215–231, 2023.

[8] L. L. Jimenez and O. Schelen. Hydra: Decentralized location-aware orchestration of containerized applications. *IEEE Transactions on Cloud Computing*, 10(4):2664–2678, 2020.

[9] A. Pires, J. Simão, and L. Veiga. Distributed and decentralized orchestration of containers on edge clouds. *Journal of Grid Computing*, 19:1–20, 2021.

[10] G. Castellano, F. Esposito, and F. Risso. A service-defined approach for orchestration of heterogeneous applications in cloud/edge platforms. *IEEE Transactions on Network and Service Management*, 16(4):1404–1418, 2019.

[11] Z. Nezami, K. Zamanifar, K. Djemame, and E. Pournaras. Decentralized edge-to-cloud load balancing: Service placement for the internet of things. *IEEE Access*, 9:64983–65000, 2021.

[12] Z. A. Mann. Decentralized application placement in fog computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3262–3273, 2022.

[13] A. Markus, V. D. Hegedus, J. D. Dombi, and A. Kertesz. Synergizing fuzzy-based task offloading with machine learning-driven forecasting for iot. In *2024 IEEE 8th International Conference on Fog and Edge Computing (ICFEC)*, pages 71–78, 2024.