Bottom-Up Resource Orchestration in Edge Computing: A Pod Profile-Aware Agent-Based Approach

Marija Gojković[1,2], Melanie Schranz[2]

[1]Alpen-Adria University, Klagenfurt, Austria
[2]Lakeside Labs GmbH, Klagenfurt, Austria

Abstract

Modern distributed systems face growing challenges in scheduling workloads across heterogeneous cloud–edge infrastructures. Advanced pod orchestration techniques—pod cloning, dependency-aware scheduling, and parallel pod processing—are crucial for improving resource utilization, scalability, and fault tolerance. Pod cloning replicates workloads to handle spikes or failures, dependency management enforces correct task sequencing, and Kubernetes-native parallelism distributes tasks across concurrent pods. Despite their benefits, these strategies are seldom unified in adaptive, bio-inspired schedulers. This paper presents an emergent scheduler integrating cloning, dependency resolution, and parallelism within a swarm intelligence framework based on the Artificial Bee Colony (ABC) algorithm. Modeling the cluster as a multi-agent ecosystem, pods are treated as food sources managed via ABC's employed, onlooker, and scout phases. This enables decentralized decision-making that dynamically adjusts cloning, enforces dependencies, and tunes parallelism in response to real-time cluster states. Evaluated on a simulated edge–cloud testbed against random assignment, dependency-agnostic best-fit, and a static ABC baseline, our scheduler achieves superior latency and deadline satisfaction rates.

Keywords

multiagent systems, edge computing, bottom-up resource orchestration, edge micro data centers, dependency-aware scheduling

1. Introduction

Efficient workload scheduling across heterogeneous cloud-edge systems is a growing challenge in modern distributed environments. Key pod orchestration strategies—cloning, dependency-aware scheduling, and parallel pod processing—optimize resource use, scalability, and fault tolerance. Pod cloning dynamically replicates workloads to handle traffic spikes or failures, dependency management ensures correct task sequencing, and parallel processing accelerates execution via Kubernetes-native mechanisms [1]. Despite addressing challenges like resource contention and coordination latency, these strategies remain underutilized in adaptive schedulers, especially those leveraging bio-inspired algorithms. This paper integrates pod cloning, dependency resolution, and parallelization into an emergent scheduler based on the Artificial Bee Colony (ABC) swarm intelligence algorithm [2]. ABC's decentralized resource allocation suits the self-organizing needs of modern infrastructures [3]. We analyze how pod management techniques affect satisfaction rates. By combining ABC optimization with Kubernetes pod controls, our scheduler dynamically adjusts cloning, parallelism, and dependency handling, balancing overhead with performance—vital for real-time edge deployments.

The paper is structured as follows: Section 2 reviews related work. Section 3 details pod optimization strategies. Section 4 describes the system model and scheduler. Section 5 covers simulation setup and system behavior analysis. Section 5 discusses key findings, and Section 6 concludes with future directions.

CPSWS'25: CPS Summer School PhD Workshop, September 22, 2025, Alghero, Sardinia, Italy
gojkovic@lakeside-labs.com (M. Gojković); schranz@lakeside-labs.com (M. Schranz)
0009-0002-4618-8929 (M. Gojković); https://orcid.org/0009-0002-4618-8929 (M. Schranz)

2. Related Work

Efficient cluster resource management has inspired approaches such as the oversubscription framework in [4], maximizing utilization. Extending to edge platforms, [5] orchestrate workloads for multi-tenant IoT services with varying SLOs, but overlook pod interdependencies—critical to performance. Microservice interdependencies, shown in [6] to degrade performance if unmanaged, are often ignored by platforms like Kubernetes, which separate deployment and routing despite their latency correlation [7]. Resource allocation in multi-clouds [8] and communication reduction [9] improve efficiency but neglect pod-level coordination. Scaling strategies like in [10] vertically aggregate per-task signals and horizontally replicate tasks, yet remain system-specific and overlook fine-grained interdependencies.

Autoscaling remains central [11], distinguishing between horizontal scaling (adjusting pod count) and vertical scaling (adding resources), but often reacts to metrics rather than SLOs. Horizontal scaling, as in Kubernetes' HPA [12], dynamically responds to load but mainly operates at or before the edge. Elastic replica management [13] enhances QoS and efficiency, resembling pod cloning in our context, but still avoids orchestration beyond the edge.

We focus on fog-layer orchestration, selecting master agents on the cloud side and leveraging slack resources from rigid pods to support elastic workloads—an area underexplored by current autoscaling. Edge-edge collaboration [7] and fog computing [14] further highlight the need for low-latency, flexible deployment in future systems.

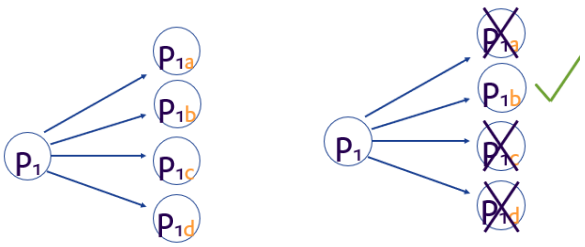


Figure 1: Pod cloning in the emergent scheduler. (a) Conceptual framework illustrating clone generation. (b) Execution workflow showing how cloned pods are scheduled and completed.

3. Types of Pod Optimization Strategies

Pod cloning, dependencies, and parallel processing are key to optimizing workload management across distributed cloud–edge environments. This boosts resource utilization, reliability, and performance in interconnected systems, enabling applications to adapt to fluctuating demands and network conditions.

Pod Dependencies and Dynamic Resource Management: Managing pod dependencies ensures correct and efficient execution in Kubernetes. In single deployments, independent pods enable horizontal scaling and resilience [15]. In multi-deployment setups, dependencies—e.g., a frontend relying on a backend or database—are critical. Kubernetes manages these via Services, DNS, and network policies [15], enabling multi-tier applications to remain scalable and fault-tolerant. Dynamic resource management techniques, such as pod cloning, integrate with dependency handling to enable autoscaling [16]. Thus, applications adjust replicas based on workload changes while preserving dependency constraints.

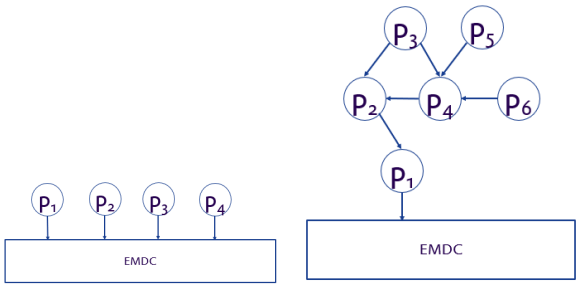


Figure 2: Key pod orchestration concepts in the emergent scheduler. (a) Parallel pod processing in the emergent scheduler, where pods process simultaneously based on shared constraints. (b) Inter-pod dependencies modeled in the emergent scheduler, including sequencing and coordination requirements.

Pod Cloning and Autoscaling: doesn't this already answer the Q: 3. how would observer mechanism work in kubernetes or computing infrastructure

Autoscaling dynamically adjusts computational resources to meet demand [16]. In Kubernetes, pod cloning replicates instances to scale horizontally during workload spikes, while load balancing distributes tasks for performance and fault tolerance. Cloning introduces scheduling complexity, as all replicas must respect original dependency rules. In complex workloads, this requires QoS-aware scheduling, as in [17], to ensure correct execution sequences. Techniques like topological sorting [18] help maintain execution order and optimize performance.

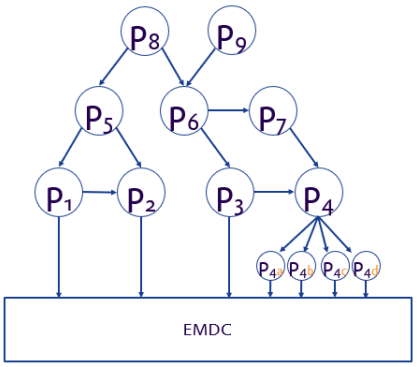


Figure 3: Overview of pod processing logic in the emergent scheduler, integrating standard, cloned, parallel, and dependent pod behaviors.

Parallel Pod Processing for Enhanced Performance: Parallel pod processing runs multiple pod instances concurrently, similar to task parallelism [19]. Computationally intensive tasks are split into subtasks and executed in parallel, improving throughput and reducing latency—vital in areas like real-time analytics and AI training. This relies on application-level parallelism, as Kubernetes does not parallelize within a single pod. Developers must implement multithreading or multiprocessing [20] to exploit concurrent execution effectively.

Efficient orchestration combines parallel processing for speed, cloning for scalability and fault tolerance, and dependency management for correctness. Together, these strategies enable robust performance in complex, distributed systems.

4. System Model

We adopt the discrete-time, agent-based simulation framework with all scheduling policies described in [2]. Therefore our emergent scheduler comprises of a master agent, a worker agent, and dynamically arriving pod agents. Pods are defined by type (rigid or elastic), resource demand, execution requirements, and queueing tolerance. The worker manages CPU and RAM allocation, using scheduling policies to accept or reject pods based on availability. For elastic pods, peer selection is performed via random, best-match, and a bottom-up resource orchestration algorithm inspired by the Artificial Bee Colony approach. The master maintains pod queues, coordinates scheduling, and employs a retry mechanism with a tunable parameter to balance rigid and elastic workloads.

To handle the pod processing methods outlined in Section 3, we implement a logic layer that pre-processes incoming pods. This step resolves their complexities—parallelism, cloning, and dependencies—transforming them into the sequential input format required by the emergent scheduler [2].

Parallel Pod Processing: If a pod p_i requires parallel processing, it is divided into n sub-pods [19] $p_{i_{p_1}}, p_{i_{p_2}}, \ldots, p_{i_{p_n}}$, where n is randomly selected from the range (1,10). Since each p_i must be processed by a single Edge Micro Data Center (EMDC), all its sub-pods are routed to the same queue, as described in [2]. To satisfy the sequential input requirement, the sub-pods are randomly ordered before insertion.

Pod Cloning: To simulate dynamic resource availability [16], the scheduler supports processing of cloned pods. If cloning is triggered, p_i is replicated n times, producing $p_{i_A}, p_{i_B}, \ldots, p_{i_N}$, with n randomly chosen between 1 and 10. Clones are placed in the same queue and randomly ordered for sequential scheduling. Each clone is assigned an observer, $o_{i_A}, o_{i_B}, \ldots, o_{i_N}$, which monitors its status. Once one clone (e.g., p_{i_A}) finishes, its observer notifies the others, causing them to terminate—mimicking unexpected resource release in the emergent scheduler.

Pod Dependencies: When p_i depends on other pods, meaning it cannot start until its prerequisites finish, dependencies are modeled as a DAG [18], with pods as nodes and dependencies as directed edges. For example, if p_i depends on p_j and p_k, edges go from p_j to p_i and p_k to p_i.

To ensure the scheduler's sequential input respects dependencies, we apply topological sorting [21]. This produces a linear order in which each pod appears after all its prerequisites. For example, if p_i depends on p_j and p_k, and p_j depends on p_l, the resulting order could be p_l, p_j, p_k, p_i, guaranteeing that all constraints are satisfied before execution.

5. System Behavior Analysis

This section analyzes how system performance evolves under different scheduling strategies and workload conditions. Through agent-based simulation, we evaluate the dynamic behavior of a distributed edge environment subjected to varying pod elasticity levels, inter-pod dependencies, and traffic intensities. The goal is to reveal how these factors influence pod satisfaction rate, i.e., how many of the available resources in the EMDC can a pod use (as rigid) and reuse (as elastic) before its assigned waiting time runs out [2].

Simulation Setup: The simulation environment is built using the MESA agent-based modeling framework in Python [22]. Its modular design enables custom agent classes with specific behaviors and decentralized execution, allowing each agent to operate independently. MESA's built-in tools facilitate modeling complex, interactive systems in a scalable way. Simulations run for 12,000 discrete time steps, tracking pod satisfaction rate under varying pod arrival rates λ from 0.55 (light load) to 0.75 and 0.95 (heavy load). This progression allows us to evaluate the scalability and robustness of the bottom-up scheduling strategy as the system approaches saturation. Two main factors are explored: pod elasticity and pod coordination constraints. We compare datasets with 30% elastic pods and with 70% elastic pods. Elastic pods provide scheduling flexibility but can also introduce overhead or contention with rigid workloads. To examine how coordination complexity interacts with elasticity, three pod-level features are included: parallel processing (10% of pods, grouped randomly between 1 and 10), cloning (10%), and inter-pod dependencies (20%). These extensions are evaluated against baseline scenarios from [2], which reflect elastic scheduling without additional pod dependencies.

Insights from Experimental Results: The results in Fig. 4–5 show pod elasticity, inter-pod dependencies, and varying λ influencing scheduling performance in terms of pod satisfaction rate. This is directly reflecting the cost of having introduced pod-dependency processing (compared against non-dependent pods), as well as the synchornization, i.e., pod coordination, costs.

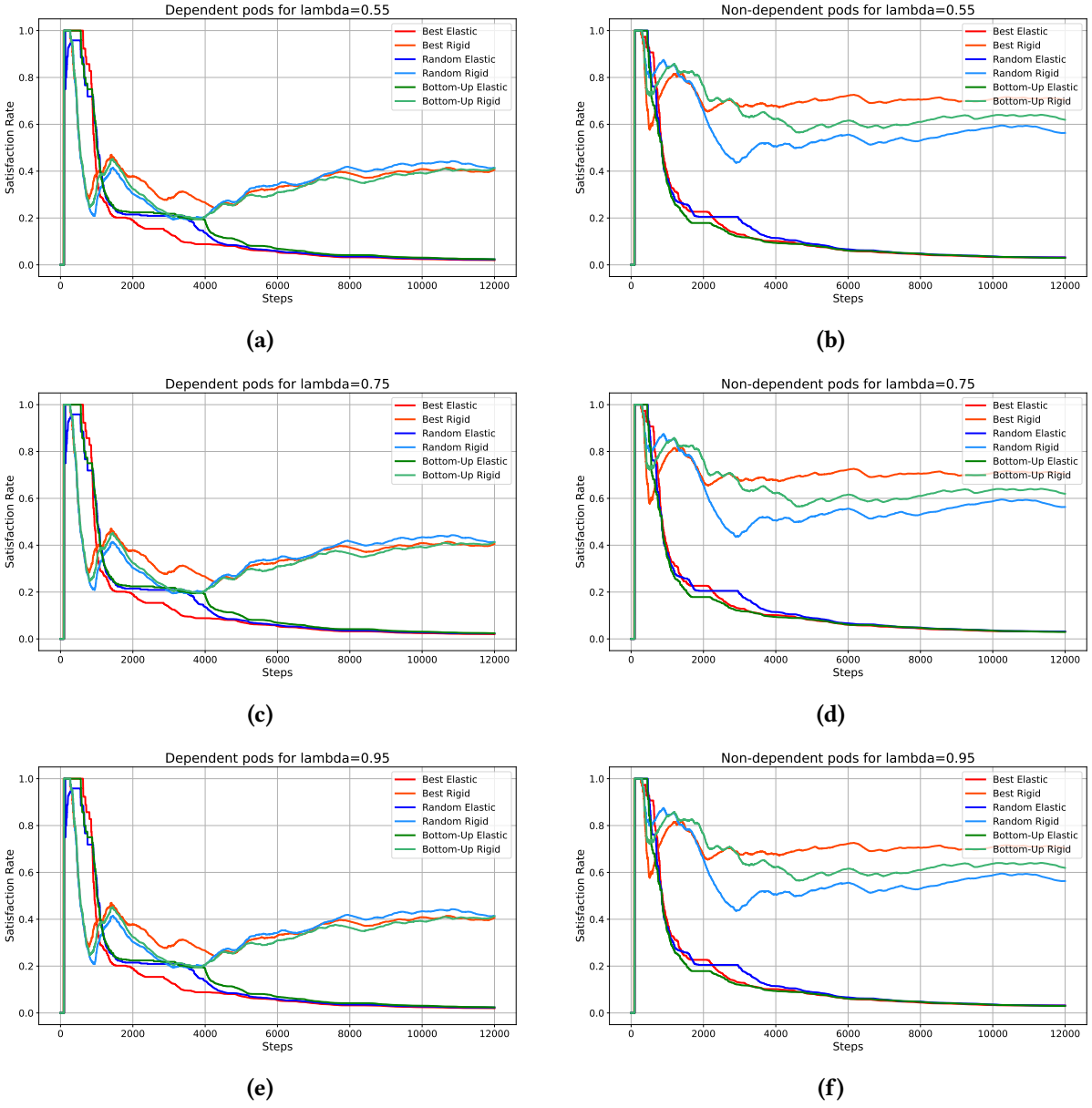
Our analysis proceeds along three key dimensions: (1) the impact of pod elasticity, (2) the effect of increasing λ, and (3) the role of pod coordination constraints such as parallel processing, cloning, and inter-pod dependencies. These perspectives together provide a detailed understanding of the trade-offs involved in bottom-up scheduling across diverse edge workload scenarios.

Satisfaction rate: (1) In panel (a), Fig. 4 (30% elastic pods), which includes pod dependencies, both the best-match and swarm intelligence (SI) strategies initially underperform compared to the random baseline. In panel (b), with independent pods, SI performs between random and best-match—a trend that continues as λ increases. (2) In panel (a), Fig. 5 (70% elastic pods), SI outperforms both baselines after timestep 5000, maintaining a satisfaction rate above 95%. For independent pods (panel b), all strategies achieve nearly perfect satisfaction, close to 100%, with minimal differences across schedulers.

Elastic pods improve performance notably under high load and elasticity, but only with adaptive scheduling. The SI method shows strong resilience and consistently outperforms others in complex scenarios. For simpler, independent workloads, advanced strategies offer little extra advantage.

Impacts and Observations: The simulation study reveals that pod elasticity and coordination requirements significantly influence satisfaction rates under varying traffic conditions. At moderate to high λ, elastic pods enable more flexible resource allocation, improving satisfaction rates—especially under the SI strategy. The addition of coordination mechanisms—such as parallel processing, cloning, and inter-pod dependencies—introduces extra complexity and can moderately reduce satisfaction in some scenarios. Nevertheless, the SI strategy shows resilience, maintaining relatively high satisfaction even under complex workloads. Interestingly, dependencies can sometimes improve stability in SI by preventing overly aggressive pod placement, resulting in more consistent satisfaction rates.

Figure 4: Satisfaction rates with 30% elastic pods under increasing arrival rates $\lambda \in 0.55, 0.75, 0.95$. Panels (a),(c) and (e) include pods with coordination constraints—cloning, parallelism, and interdependencies—while panel (b),(d) and (f) show results for independent pods. Results are shown for random, best-match, and SI scheduling strategies.

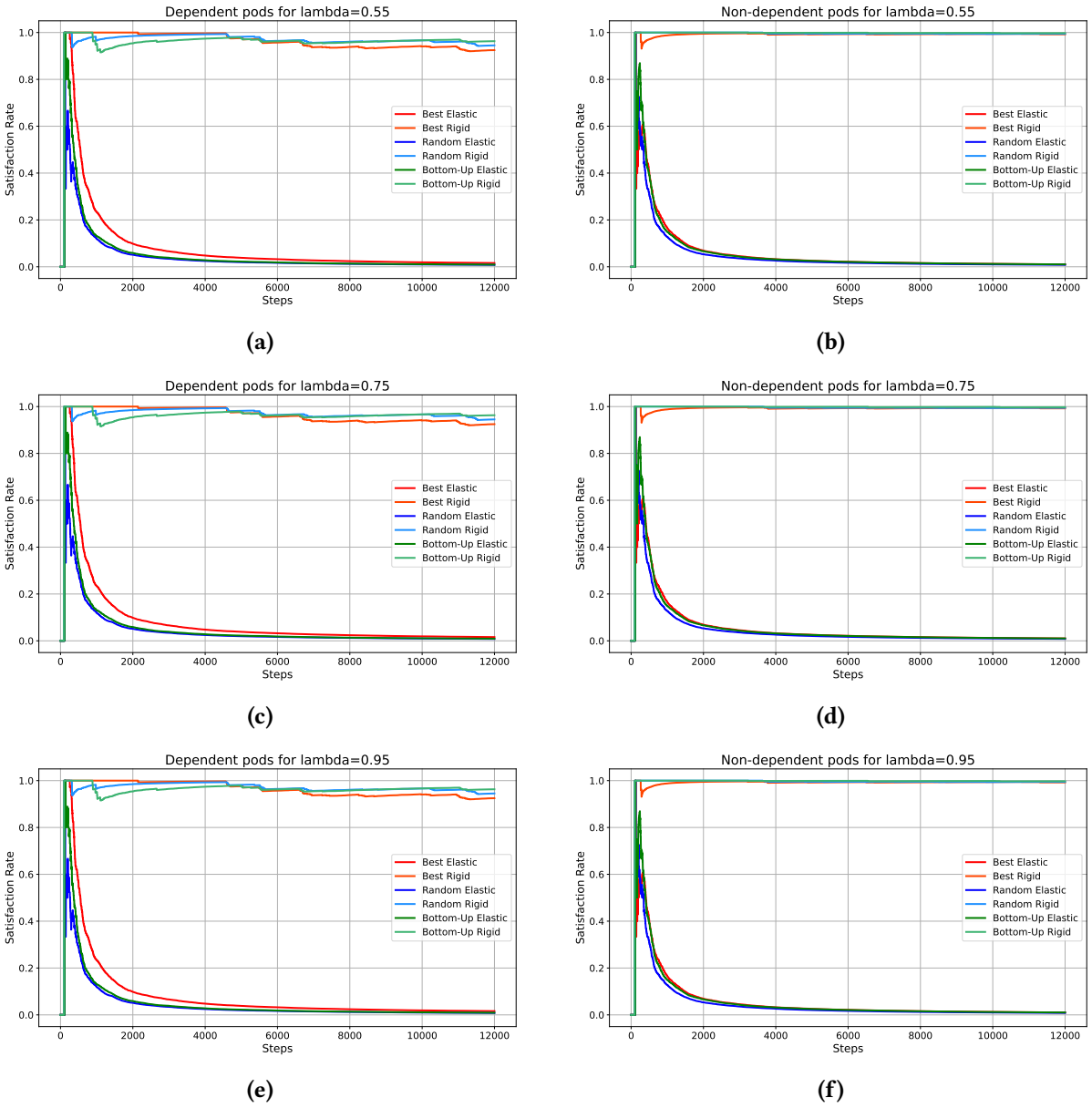


Overall, the results indicate that no single strategy dominates across all conditions. Best-match performs well in stable, predictable environments, while SI excels in elastic, dynamic, and uncertain conditions. These findings can inform the design of adaptive, decentralized orchestration frameworks for heterogeneous and evolving edge–cloud deployments.

6. Conclusion

Efficient workload scheduling across heterogeneous cloud-edge infrastructures is increasingly critical for modern distributed systems. Pod-level orchestration techniques—cloning, parallel processing, and dependency-aware execution—are essential for managing complexity, scalability, and fault tolerance, yet remain underutilized in decentralized schedulers. This work integrates these mechanisms into a swarm-intelligence-based scheduler using the Artificial Bee Colony (ABC) algorithm. Simulations

Figure 5: Satisfaction rates with 70% elastic pods under increasing arrival rates $\lambda \in 0.55, 0.75, 0.95$. Panels (a),(c) and (e) include pod coordination mechanisms; panel (b),(d) and (f) depict scenarios with independent pods. The SI strategy shows increased performance under high elasticity.



show that pod elasticity and coordination requirements strongly impact performance under high load. Elastic pods improve satisfaction rates with the SI strategy, though they may increase queue lengths. Best-match effectively controls queues but can struggle with slack estimation errors and complex coordination. Notably, dependencies like parallelism and sequencing can reduce queue buildup by moderating placement aggressiveness, providing a self-regularizing effect for SI. No single strategy dominates: best-match fits tightly constrained settings, while SI is more robust in dynamic, elastic, and uncertain environments. These insights guide the design of resilient, adaptive orchestration systems balancing flexibility, efficiency, and stability amid evolving workloads.

Acknowledgments

This work was performed in the course of the EU-project ACES supported by EU's Horizon Europe under the grant agreement No. 101093126 (HORIZON-CL4-2022-DATA-01-02).

Declaration on Generative AI

During the preparation of this work, the author(s) used Chat-GPT-4 and Grammarly in order to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] Kubernetes, Fine parallel processing using a work queue, 2025. URL: https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/.

[2] A. Ghasemi, M. Schranz, Bottom-up resource orchestration in edge computing: An agent-based modeling approach, in: 2024 IEEE 12th International Conference on Intelligent Systems (IS), IEEE, 2024, pp. 1–7.

[3] M. Umlauft, M. Gojkovic, K. Harshina, M. Schranz, Bottom-up bio-inspired algorithms for optimizing industrial plants., in: ICAART (1), 2023, pp. 59–70.

[4] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, C. Li, Rose: Cluster resource scheduling via speculative over-subscription, in: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 949–960. doi:10.1109/ICDCS.2018.00096.

[5] F. Guim, T. Metsch, H. Moustafa, T. Verrall, D. Carrera, N. Cadenelli, J. Chen, D. Doria, C. Ghadie, R. G. Prats, Autonomous lifecycle management for resource-efficient workload orchestration for green edge computing, IEEE Transactions on Green Communications and Networking 6 (2022) 571–582. doi:10.1109/TGCN.2021.3127531.

[6] M. Hu, Z. Guo, H. Wen, Z. Wang, B. Xu, J. Xu, K. Peng, Collaborative deployment and routing of industrial microservices in smart factories, IEEE Transactions on Industrial Informatics 20 (2024) 12758–12770. doi:10.1109/TII.2024.3424347.

[7] J. Qi, H. Zhang, X. Li, H. Ji, X. Shao, Edge-edge collaboration based micro-service deployment in edge computing networks, in: 2023 IEEE Wireless Communications and Networking Conference (WCNC), 2023, pp. 1–6. doi:10.1109/WCNC55385.2023.10119013.

[8] H. X. Nguyen, S. Zhu, M. Liu, Graph-phpa: Graph-based proactive horizontal pod autoscaling for microservices using lstm-gnn, in: 2022 IEEE 11th International Conference on Cloud Networking (CloudNet), 2022, pp. 237–241. doi:10.1109/CloudNet55617.2022.9978781.

[9] W. Lv, Q. Wang, P. Yang, Y. Ding, B. Yi, Z. Wang, C. Lin, Microservice deployment in edge computing based on deep q learning, IEEE Transactions on Parallel and Distributed Systems 33 (2022) 2968–2978. doi:10.1109/TPDS.2022.3150311.

[10] K. Rządca, P. Findeisen, J. Świderski, P. Zych, P. Broniek, J. D. M. Kusmierek, P. K. Nowak, B. Strack, P. Witusowski, S. Hand, J. Wilkes, Autopilot: workload autoscaling at google, Proceedings of the Fifteenth European Conference on Computer Systems (2020). URL: https://api.semanticscholar.org/CorpusID:218489692.

[11] A. A. Pramesti, A. I. Kistijantoro, Autoscaling based on response time prediction for microservice application in kubernetes, in: 2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA), 2022, pp. 1–6. doi:10.1109/ICAICTA56449.2022.9932943.

[12] L. H. Phuc, L.-A. Phan, T. Kim, Traffic-aware horizontal pod autoscaler in kubernetes-based edge computing infrastructure, IEEE Access 10 (2022) 18966–18977. doi:10.1109/ACCESS.2022.3150867.

[13] P. Zhao, P. Wang, X. Yang, J. Lin, Towards cost-efficient edge intelligent computing with elastic deployment of container-based microservices, IEEE Access 8 (2020) 102947–102957. doi:10.1109/ACCESS.2020.2998767.

[14] A. J. Fahs, G. Pierre, E. Elmroth, Voilà: Tail-latency-aware fog application replicas autoscaler, in: 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and

Telecommunication Systems (MASCOTS), 2020, pp. 1–8. doi:10.1109/MASCOTS50786.2020.9285953.

[15] Kubernetes, Pods, 2025. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[16] T. Lorido-Botran, R. N. Calheiros, R. M. Rodriguez, R. Buyya, C. Vecchiola, Autoscaling in the cloud: A survey, IEEE Transactions on Services Computing 8 (2015) 947–969. doi:10.1109/TSC.2014.2350938.

[17] J. Yu, R. Buyya, K. Ramamohanarao, Workflow scheduling algorithms for service-oriented cloud computing with blending of deadline and budget constraints, Proceedings of the 2008 IEEE International Symposium on Cluster Computing and the Grid (CCGRID) (2008) 427–436. doi:10.1109/CCGRID.2008.46.

[18] S. Even, Graph Algorithms, Cambridge University Press, 2011. Chapters on Directed Acyclic Graphs.

[19] S. Manvi, G. Shyam, Cloud Computing: Concepts and Technologies, 1st ed., CRC Press, 2021. doi:10.1201/9781003093671.

[20] P. S. Pacheco, An introduction to parallel programming, Morgan Kaufmann, 2011.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009. Section 22.4: Topological Sort.

[22] D. Masad, J. L. Kazil, Mesa: An agent-based modeling framework, 2015. URL: https://github.com/projectmesa/mesa/blob/master/CITATION.bib.