Towards a Framework for Optimized Microservices Placement in Cloud Native Environments

Riane Driss, Ettazi Widad, Ettalbi Ahmed
IMS Team-ADMIR Laboratory ENSIAS-Rabat IT Center, Mohammed V University in Rabat, Rabat, Morocco

***Abstract*—In recent times, cloud-native technologies have increasingly enabled the design and deployment of applications using a microservice architecture, enhancing modularity, scalability, and management efficiency. These advancements are specifically tailored for the creation and orchestration of containerized applications, marking a significant leap forward in the industry. Emerging cloud-native applications employ container-based virtualization instead of the traditional virtual machine approach. However, adopting this new cloud-native approach requires a shift in vision, particularly in addressing the challenges of microservices placement. Ensuring optimal resource utilization, maintaining service availability, and managing the complexity of distributed deployments are critical considerations that necessitate advanced orchestration and automation strategies. We introduce a new framework for optimized microservices placement that optimizes application performance based on resource requirements. This approach aims to efficiently allocate infrastructural resources while ensuring high service availability and adherence to service level agreements. The implementation and experimental results of our method validate the feasibility of the proposed approach.**

Keywords—Cloud native architecture; Service placement; containerization; Cloud resource allocation; microservices architecture

I. Introduction

Cloud computing revolutionizes IT infrastructure by offering on-demand access to a shared pool of configurable computing resources, such as servers, storage, and applications, over the internet. This model enhances flexibility, scalability, and cost efficiency, making it ideal for businesses of all sizes. Cloud-native development takes full advantage of cloud computing by building and deploying applications specifically designed to operate in a cloud environment. These applications leverage microservice architecture, which breaks down a monolithic application into smaller, independent services that communicate through APIs. This approach enhances agility, scalability, and resilience, as each microservice can be developed, deployed, and scaled independently. Containers further support this architecture by encapsulating microservices and their dependencies into lightweight, portable units, ensuring consistency across different environments. Technologies like Docker [1] and Kubernetes [2] facilitate container orchestration, automating deployment, scaling, and management, thereby streamlining the development and operational processes in a cloud-native landscape.

Deploying cloud-native applications involves leveraging containerization and container orchestration to achieve seamless scalability, flexibility, and resilience. Containers, which package applications with their dependencies, ensure consistency across different environments, from development to production. This approach simplifies the deployment process and enhances the portability of applications. Container orchestration, with Kubernetes being the most widely used platform, automates critical functions such as deployment, management, scaling, and networking of containers. Kubernetes manages containerized applications across a cluster of machines, ensuring optimal resource utilization and availability. It handles load balancing, scales applications based on demand, and provides self-healing capabilities by automatically restarting failed containers.

The deployment process of cloud-native applications typically begins with defining application components in declarative configuration files, which Kubernetes uses to create and maintain the desired state of the application. Integration with continuous integration and continuous deployment (CI/CD) pipelines further streamlines the process, allowing for rapid and reliable updates. CI/CD pipelines automate the building, testing, and deployment of code changes, reducing manual intervention and minimizing the risk of errors. This automation not only improves operational efficiency but also enhances the application's ability to adapt to changing workloads and recover from failures. By adhering to cloud-native principles, organizations can achieve greater agility, scalability, and resilience in their application deployments, ensuring they are well-prepared to meet evolving business demands.

The challenge of optimal microservices placement over multiple resources is a critical aspect of managing cloud-native applications, particularly in a dynamic and distributed cloud environment. As applications are decomposed into numerous microservices, each with distinct resource requirements and performance characteristics, determining the most efficient placement of these microservices becomes increasingly complex. This challenge is compounded by the need to balance multiple factors such as resource utilization, latency, service availability, and compliance with service level agreements (SLAs).

Effective microservices placement requires sophisticated algorithms that can analyze and predict resource demands, identify potential bottlenecks, and dynamically allocate resources to maintain optimal performance. These algorithms must also account for the heterogeneity of resources across different cloud environments, including various types of compute, storage, and networking resources. Furthermore, they must be resilient to changes in workload patterns and capable of quickly adapting to failures or unexpected spikes in demand.

Addressing these challenges is essential for maximizing infrastructural efficiency, reducing operational costs, and ensuring the high availability and reliability of cloud-native applications.

This article presents a new approach to optimizing the placement of microservices across multiple resources in cloud-native environments. By using PSO-based algorithm, our method strategically deploys microservices to enhance resource utilization and service performance. The proposed solution incorporates continuous monitoring to anticipate resource demands and mitigate bottlenecks, ensuring efficient distribution of workloads. This approach also employs container orchestration platforms, to automate the dynamic scaling and management of microservices, thereby maintaining high availability and resilience. Evaluation through comprehensive simulations highlights the efficacy of our method in optimizing the placement of microservices based on monitored data.

The following sections of this paper take a systematic approach to examine the optimal placement of microservices in cloud native environment. Section II reviews related work, identifying gaps and opportunities in existing methodologies. Section III details our proposed framework, including its design and implementation. In Section IV, we present the Sock Shop application as a use case for microservices deployment. Section V introduces our proposed algorithm for microservices placement using Particle Swarm Optimization (PSO), along with performance evaluations that highlight the results of our simulations and the advantages of our approach. Finally, Section VI provides a conclusive summary of our findings, discussing their implications for future research and practical applications in cloud-native environments.

II. Related Work

The placement of microservices in cloud-native environments is crucial for enhancing performance, scalability, and resource utilization. Numerous techniques and frameworks have been proposed to address microservices placement challenges. This section reviews existing work, categorized into heuristic approaches, optimization-based techniques, and frameworks. Heuristic approaches are popular for their simplicity and efficiency, offering sub-optimal solutions quickly, making them ideal for large-scale deployments. Greedy algorithms [8, 18], for instance, place microservices by iteratively selecting the best local option, such as prioritizing resource-intensive microservices to ensure adequate resource allocation.

Optimization-based techniques use mathematical models and algorithms to find near-optimal or optimal placement solutions, offering superior results in resource utilization and performance despite their higher computational demands. Methods like Linear and nonlinear Programming (LP) [3, 6] and Mixed-Integer Linear Programming (MILP) [5, 7] model the placement problem with linear constraints, providing powerful solutions but often at a high computational cost for large-scale problems. Metaheuristic algorithms, including Genetic Algorithms (GA) [4, 10, 13], and Particle Swarm Optimization (PSO) [14, 9], are also widely used. These algorithms are designed to escape local optima and thoroughly explore the solution space, making them well-suited for complex placement problems.

Various frameworks and tools streamline microservices placement in cloud-native settings by integrating placement algorithms with container orchestration platforms like Kubernetes, automating deployment. Kubernetes-native solutions [11, 19, 22] leverage features like node affinity/anti-affinity, taints and tolerations, and custom schedulers [12, 15], empowering developers to guide placement decisions based on resource needs and workload traits [16]. Additionally, service meshes such as Istio [21] and Linkerd [24] enhance placement strategies by dynamically altering request routing according to real-time performance metrics [17], bolstering traffic management [23] and observability capabilities.

However, there is a lack of continuous monitoring and real-time service redeployment in these approaches, which are critical for maintaining optimal performance and resource utilization in dynamic cloud environments. In our work, we focus on integrating these capabilities to ensure that microservices placement can adapt to changing workloads and cloud native infrastructure conditions in real-time.

III. Framework for Optimsed Microservices Placement

This section initially outlines the criteria necessary to meet microservices placement key points aligned with workload characteristics obtained from continuous monitoring. Following this, we introduce a design and implementation of our framework.

A. *Key Points*

In order to guarantee the performance of deployed applications on cloud native environment, we consider the following requirements:

1) Continuous monitoring for deployed microservices applications in a cloud-native environment is crucial for maintaining optimal performance, security, and reliability. By constantly tracking metrics such as CPU usage, memory consumption, network traffic, and response times, continuous monitoring provides real-time insights into the health and behavior of each microservice. This proactive approach allows for the rapid detection and resolution of issues, minimizing downtime and ensuring seamless user experiences. Furthermore, continuous monitoring supports scalability by identifying performance bottlenecks and guiding resource allocation decisions, ultimately enhancing the efficiency and resilience of cloud-native applications.

2) Collecting and analyzing workload data involves gathering detailed metrics on system performance and resource utilization under actual usage conditions. This process provides critical insights into how different workloads impact the system, revealing patterns and trends that are not apparent through continuous monitoring alone. For microservice applications in cloud-native environments, this data is invaluable. It helps developers and administrators optimize resource allocation, design better scaling strategies, and enhance overall performance. By understanding real-world usage patterns, teams can make informed decisions that improve the efficiency and reliability of their microservices deployments.

3) Placement strategies for microservices deployment based on heuristics that consider continuous monitoring and workload data collection are essential for optimizing performance in cloud-native environments. These strategies use real-time monitoring to track system health and resource usage, while data analysis provides insights into actual workload patterns. By combining these approaches, heuristic algorithm can make informed decisions about the optimal placement of microservices, ensuring efficient resource utilization, minimizing latency, and enhancing overall application resilience and scalability.

B. *Optimised Microservices Placement Framework Design*

Based on the key points presented in the previous section, we design the framework for optimized microservices placement as shown in Fig. 1 and Fig. 2. Our framework includes the following components: i) Workload continuous monitoring, ii) workload analysis, iii) Microservices Placement, iv) cloud-native infrastructure Management. The proposed framework handles cloud-native application requirements that include resource parameters and microservices inter-communication for efficient application deployment in cloud-native infrastructure such as Kubernetes platform.

The workload continuous monitoring provides real-time visibility into the performance and health of microservices by tracking key metrics. It measures CPU usage to detect overutilization or underutilization, monitors memory consumption to prevent leaks and ensure efficient usage, tracks network traffic to identify bottlenecks and optimize communication, and measures response times to ensure low latency and high performance. This comprehensive monitoring is crucial for maintaining optimal functionality and reliability of microservices in a cloud-native environment.

The workload analysis collects and processes detailed metrics on system performance and resource utilization under actual usage conditions. Key functions include:

- Data Collection: Aggregates performance data over time, providing a historical view of system behaviour.
- Pattern Identification: Analyzes data to identify trends, peak usage times, and typical workload patterns.
- Impact Assessment: Evaluate how different workloads affect system components, enabling resource allocation optimization.

The Microservices placement uses heuristic algorithm to make decision about where to deploy microservices. Key features include:

- Resource Allocation: Determines the optimal distribution of resources based on continuous monitoring and workload analysis.
- Scalability Management: Adjusts the number of instances of each microservice to match current demand, ensuring efficient resource usage.
- Latency Minimization: Places microservices in locations that reduce communication delays, enhancing overall system responsiveness.

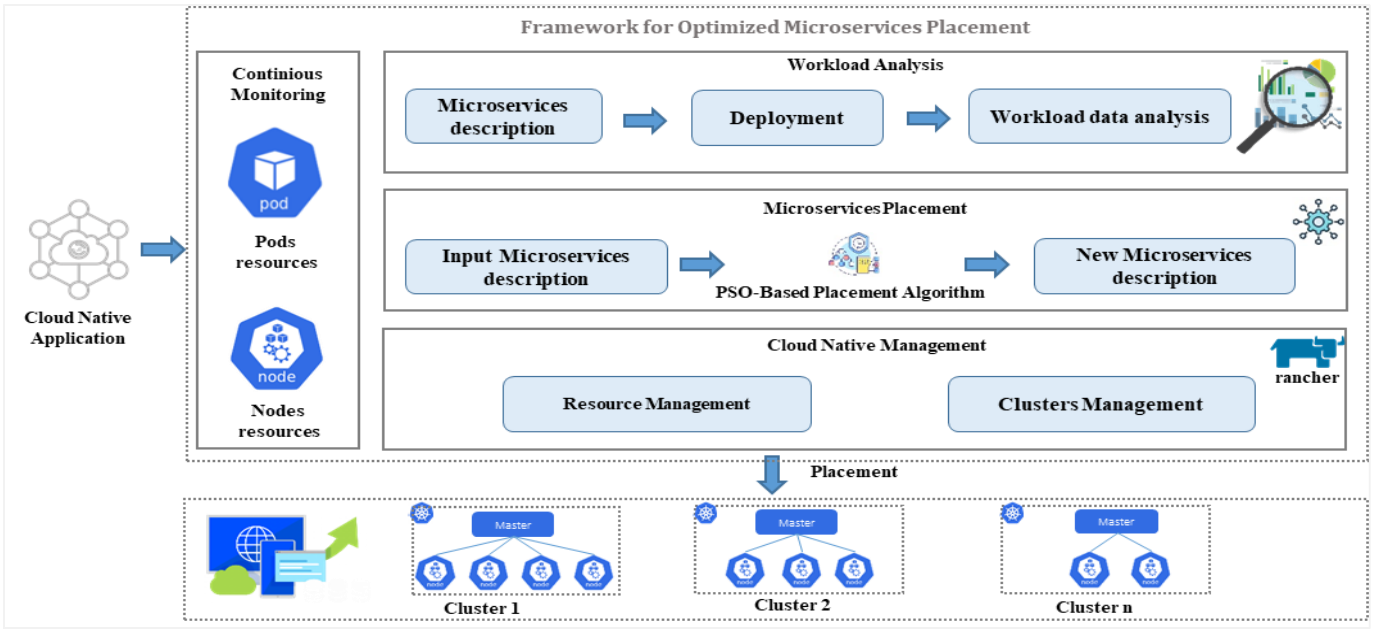


Fig. 1. Overview of optimised microservices placement framework design.

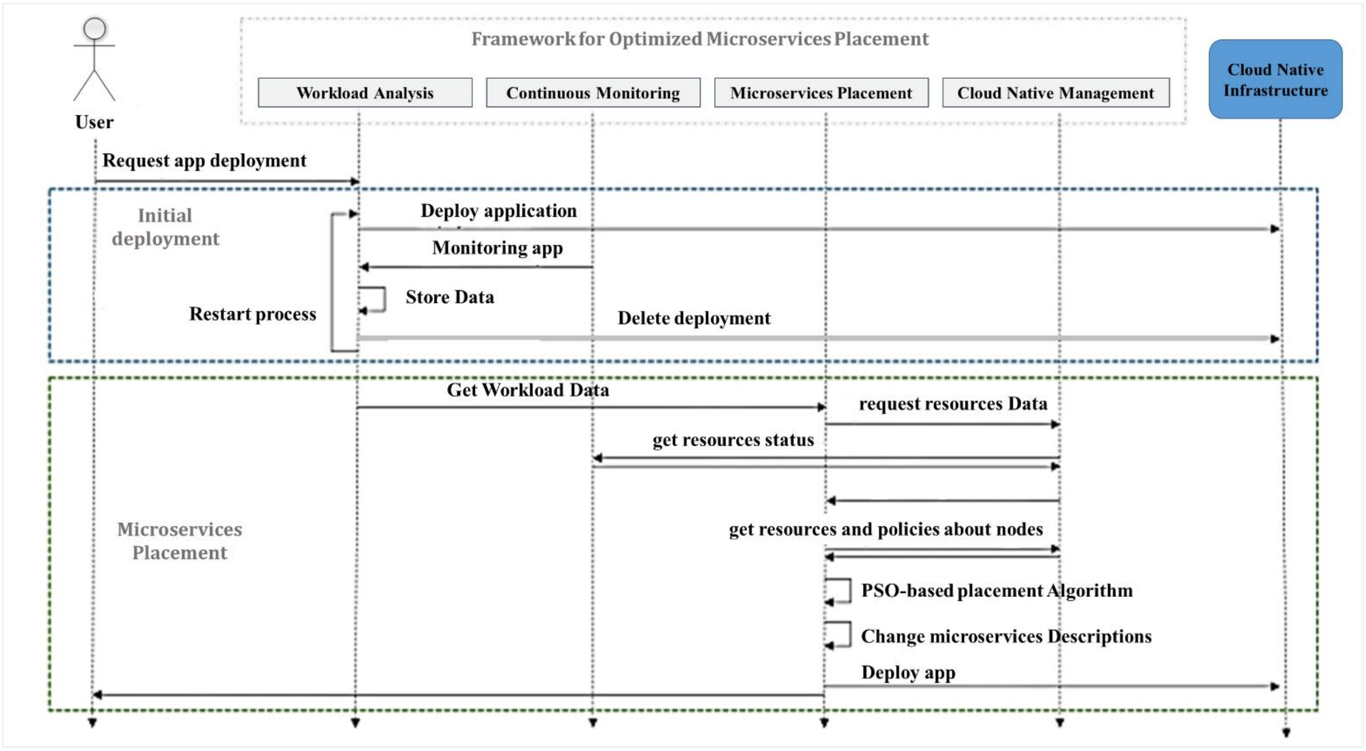


Fig. 2. Workflow diagram of optimised microservices placement framework.

The cloud-native management, such as a Kubernetes platform, provides the environment for deploying and managing microservices. Key capabilities include:

- Container Orchestration: Manages the deployment, scaling, and operation of containerized applications, ensuring consistent and reliable performance.
- Self-Healing: Automatically restarts failed containers, ensuring high availability and resilience.
- Load Balancing: Distributes incoming traffic across multiple instances of a service to optimize resource usage and prevent overloads.
- Resource Management: Dynamically allocates computing resources to meet the needs of deployed microservices based on real-time data.

Fig. 2 illustrates the workflow diagram of our framework. The process begins with users submitting requests to the framework, providing a YAML description (which includes specifying all the necessary components, such as microservices, their dependencies, resource requirements, environment variables, and network configurations), of the cloud-native application along with workload parameters, such as CPU and memory limits, replica counts, and other resource requirements.

The second steps consists of deploying the application the cloud native platform (e.g. kubernetes), monitoring and collecting workload data (i.e. resource usage for pods and nodes).

The third step focuses on microservices placement within cloud native platform. Based on the stored data from previous step, including resource status. This stage performs the placement Algorithm, quantifies each microservice's description and returns the placement results.

IV. Use Case: Sockshop Application

A. *Microservice Demo*

The Sock-Shop application [20], also known as the Microservices Demo, is a widely recognized reference application designed to illustrate microservices architecture in practice. It serves as a tool for demonstrating and testing microservice and cloud-native technologies. Simulating an e-commerce platform that sells socks, it provides developers and architects with a practical example to explore, learn, and experiment with microservices concepts, technologies, and best practices.

It is built using Spring Boot, Go kit, and Node.js, and is packaged within Docker containers. We use Locust [25] as a testing tool that allows defining user behavior and simulating traffic to create workloads for the application. This helps in evaluating how well your application handles different levels of user load. Fig. 3 provides more details about its overall architecture.

B. *Testbed Cloud Native Environment*

As illustrated in Fig. 4, we set up cloud-native environment using multiple Kubernetes clusters. We created two different Kubernetes clusters to test and experiment with the Sock-Shop cloud-native application. The first cluster consists of one master node and three worker nodes. The second cluster includes master node and two worker nodes.

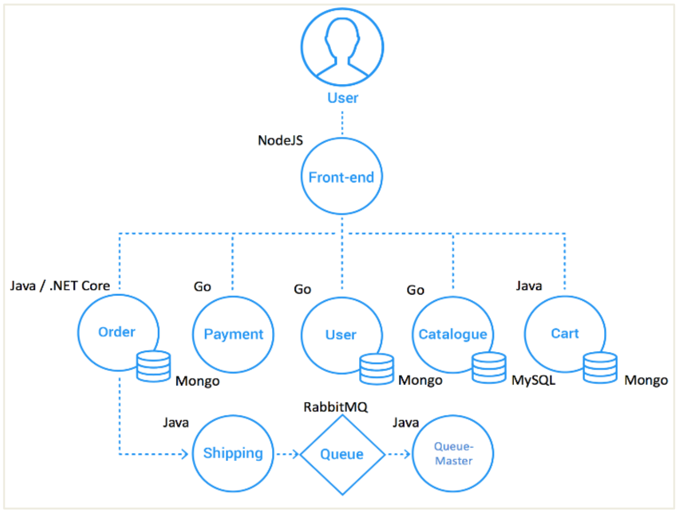


Fig. 3. Sock-Shop application architecture.

We use Rancher [26] tool that offers built-in monitoring capabilities through its integrated monitoring stack, which includes Prometheus for metrics collection and Grafana for visualization. Rancher allows monitoring resource usage, CPU and memory utilization, as well as workload data across multiple Kubernetes clusters.

After the application has been deployed, Rancher actively monitors the health status of Kubernetes pods to ensure optimal performance. Leveraging its robust automation capabilities, Rancher dynamically introduces various workloads tailored to the specific requirements of the application type. These workloads, meticulously crafted within a shell script and utilizing benchmarking tools, aim to emulate real-world scenarios and stress test the application's resilience. Following the injection process, Rancher diligently gathers resource utilization data from all microservices constituting the application, meticulously assessing CPU, memory, and network usage. Subsequently, this monitored data is securely stored in a dedicated profiling datastore, facilitating comprehensive analysis and enabling informed decision-making regarding resource allocation and performance optimization strategies.

Upon reaching the specified duration parameter, if the elapsed time of application deployment matches, Rancher initiates the termination process, dismantling the application infrastructure. This cyclic operation persists automatically until the desired number of iterations is achieved, ensuring thorough profiling and assessment of the application's performance under varying conditions. As the profiling process concludes, Rancher leverages its visualization capabilities to render the stored profiling data into a comprehensive graph format, typically presented as a scatter plot. This graphical representation provides stakeholders with valuable insights into the application's behavior, enabling informed decision-making regarding optimization strategies and resource allocation.

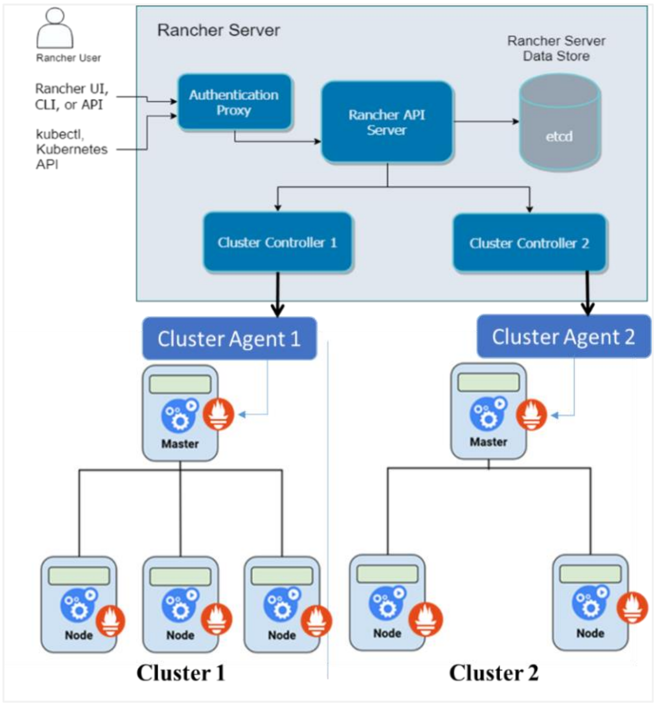
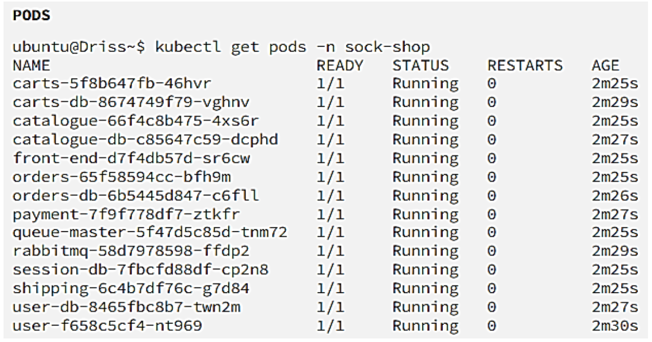


Fig. 4. Multiple Kubernetes cluster management with Rancher.

C. *Sock Shop Performance*

Fig. 5 shows the output of the command `kubectl get pods -n sock-shop` which lists the pods running in the "sock-shop" namespace. This indicates a healthy and stable deployment of the sock-shop microservices application in Kubernetes.



```
PODS

ubuntu@Driss~$ kubectl get pods -n sock-shop
NAME                           READY   STATUS    RESTARTS   AGE
carts-5f8b647fb-46hvr          1/1     Running   0          2m25s
carts-db-8674749f79-vghnv      1/1     Running   0          2m29s
catalogue-66f4c8b475-4xs6r     1/1     Running   0          2m25s
catalogue-db-c85647c59-dcphd   1/1     Running   0          2m27s
front-end-d7f4db57d-sr6cw      1/1     Running   0          2m25s
orders-65f58594cc-bfh9m        1/1     Running   0          2m25s
orders-db-6b5445d847-c6fll     1/1     Running   0          2m26s
payment-7f9f778df7-ztkfr       1/1     Running   0          2m27s
queue-master-5f47d5c85d-tnm72  1/1     Running   0          2m25s
rabbitmq-58d7978598-ffdp2      1/1     Running   0          2m29s
session-db-7fbcfd88df-cp2n8    1/1     Running   0          2m25s
shipping-6c4b7df76c-g7d84      1/1     Running   0          2m25s
user-db-8465fbc8b7-twn2m       1/1     Running   0          2m27s
user-f658c5cf4-nt969           1/1     Running   0          2m30s
```

Fig. 5. Running Pods for sock shopp application.

The performance analysis of the Sock Shop services after 15 iterations, as depicted in Fig. 6, reveals distinct patterns in resource utilization across various services. The front-end service exhibits the highest CPU usage at 1.2 cores, which indicates it is a critical component in terms of processing power. Similarly, the queue-master service shows a significant memory consumption of 2.7 GB, suggesting it handles substantial data throughput. In contrast, services like carts-db, orders-db, payment, and user-db have minimal CPU and memory usage, implying they are lightweight and less demanding on infrastructure resources.

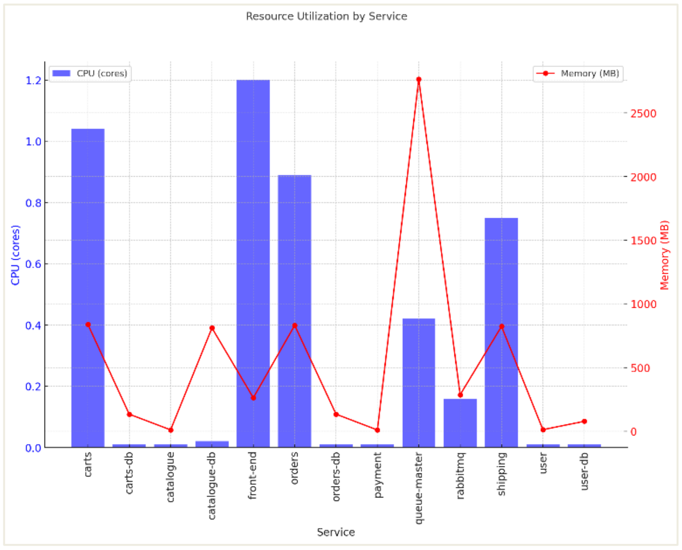


Fig. 6. Resources utilisation by services.

V. PSO-BASED PLACEMENT ALGORITHM

This section presents PSO PSO-based placement algorithm used in our framework in order to deploy microservices in the cloud native infrastructure. Then we evaluate the performances of algorithm and the results analysis.

A. *PSO-Based Microservices Placement Algorithm*

In this section, we present a model for microservices placement based on Particle Swarm Optimization (PSO), an optimization technique inspired by the social behavior of particles in nature. PSO has been adapted to address the microservices placement problem by effectively exploring the solution space to find optimal placement configurations.

In the context of microservices placement, the variables and notations used includes: *C* represents the set of available clusters where microservices can be deployed in the cloud-native infrastructure. H_n denotes the set of hosts within the nth cluster, which serves as the infrastructure for hosting microservices. *MS* signifies the set of microservices that need to be placed within the cloud-native environment. P_i refers to the position of particle *i* within the search space, where each particle represents a potential solution for microservices placement. V_i represents the velocity of particle *i* within the search space, indicating the rate and direction of movement as the algorithm progresses.

The update of particle positions and velocities in the PSO algorithm is performed using the following equations:

$$V_i(t+1) = w * V_i(t) + c_1 * rand() * \left(BestPos_i(t) - CurrentPos_i(t)\right) + c_2 * rand() * \left(GlobalBestPos_i(t) - CurrentPos_i(t)\right) \quad (1)$$

$$P_i(t+1) = CurrentPos_i(t) + V_i(t+1) \quad (2)$$

Where, w is the inertia weight, c_1 and c_2 are the acceleration coefficients, and $rand()$ is a function that generates a random number between 0 and 1.

Fitness of microservices placement solutions is evaluated based on resource (CPU and Memory) utilization. A placement solution is considered better if it minimizes resource usage while meeting performance and availability constraints.

Algorithm 1 outlines the steps for implementing the PSO algorithm to optimize microservices placement in a cloud-native environment. It includes the initialization of particles, updating their positions and velocities, and evaluating fitness based on resource utilization (CPU and memory) to find the best placement solution.

Algorithm 1: PSO-Based Microservices Placement

```
    Input:
    - C: Set of clusters
    - H_n: Set of hosts in cluster n
    - MS: Set of microservices
    - MaxIterations: Maximum number of iterations
    - PopulationSize: Number of particles in the swarm
    - w: the weight
    - c_1, c_2: Acceleration coefficients
    Output:
    - BestPlacement: Optimal microservices placement solution
1.  -Initialize a population of particles with random positions and velocities.
2.  -Initialize the best position found by each particle and the global best position.
3.  for iteration = 1 to MaxIterations do
4.      for each particle do
5.          -Evaluate the fitness of the particle's current position based on resource utilization (cpu and memory).
6.          -Update the best position found by the particle and the global best position if necessary.
7.          -Update the particle's velocity using the velocity update equation (1).
8.          -Update the particle's position using the position update equation (2).
9.      end for
10. end for
11. -Select the placement corresponding to the global best position as the optimized microservices placement solution.
```

B. *Performance Evaluation for PSO-Based Microservices Placement Algorithm*

In this section, we evaluate the performance of the PSO-based microservices placement algorithm by using the Sock Shop microservices demo as a test case. The Sock Shop demo is a widely recognized benchmark for demonstrating microservices architectures, consisting of various services that simulate an e-commerce application for selling socks. This demo provides a realistic and complex environment for testing and validating the efficiency and effectiveness of our placement algorithm.

1) Experimental setup: The experimental setup involves multiple components, including the configuration of Kubernetes clusters, the deployment of the Sock Shop microservices, and the definition of key parameters for the Particle Swarm Optimization algorithm. By creating a controlled environment that mirrors real-world conditions, we can systematically measure the impact of the PSO algorithm on resource utilization, load balancing, service latency, and overall system fitness.

Table I details the specific configurations and parameters used in the experiment, and the metrics employed for performance evaluation.

TABLE I. EXPERIMENTAL PARAMETERS

	Parameter	Detail
Workload	Kubernetes Clusters (C)	Multiple clusters set up to host the Sock Shop microservices.
	Hosts (H_n)	Each cluster consists of multiple hosts with random capacities of CPU and memory resources.
	Microservices (MS)	The Sock Shop application includes user, catalog, cart, order, payment, and shipping services.
PSO Parameters	w	0.5
	c_1	1.5
	c_2	1.5
	Poupulation size	30
	Max iterations	100
Microservices resource requirments	user	Cpu=1 , Memory =512
	catalog	Cpu=2 , Memory =1024
	orders	Cpu=2 , Memory =512
	Payment	Cpu=2 , Memory =1024
	Shipping	Cpu=1 , Memory =512
	Cart	Cpu=1 , Memory =512

2) Experimental result: To evaluate the efficiency of the PSO-based microservices placement algorithm, we conducted an analysis of resource utilization, specifically focusing on CPU and memory usage across all hosts. The assessment was performed before and after applying the PSO algorithm, and the results are depicted in Fig. 7 and Fig. 8.

The results demonstrate that the PSO-based placement algorithm significantly enhances resource utilization. By balancing the CPU and memory usage, the algorithm ensures that no single host becomes a bottleneck, thereby improving the overall performance and reliability of the microservices deployment. The reduction in resource hotspots contributes to a more efficient and resilient cloud-native environment, capable of handling varying workloads with greater stability.

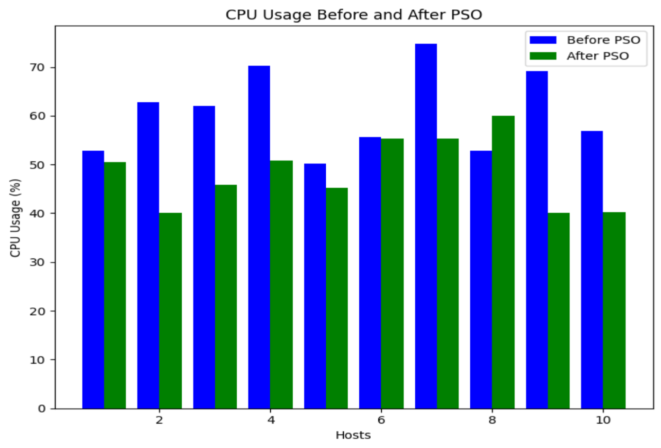


Fig. 7. CPU utilisation before and after PSO.

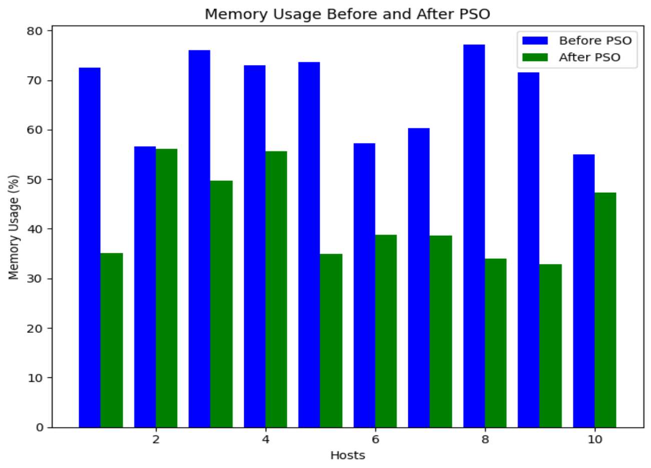


Fig. 8. Memory utilisation before and after PSO.

VI. CONCLUSION

In conclusion, this paper has explored the significant advancements enabled by cloud-native technologies in the design and deployment of applications utilizing a microservice architecture. These technologies enhance modularity, scalability, and management efficiency, facilitating a shift from traditional virtual machine-based approaches to container-based virtualization. The adoption of this cloud-native paradigm introduces new challenges, particularly in the optimal placement of microservices, which is crucial for maximizing resource utilization, ensuring service availability, and managing the complexity of distributed systems.

We proposed a new framework for optimized microservices placement, focusing on efficient resource allocation while maintaining high service availability and compliance with service level agreements. By leveraging Particle Swarm Optimization (PSO), our approach effectively addresses the challenges associated with microservices placement in cloud-native environments. The experimental results obtained from the Sock Shop application use case demonstrate the feasibility and effectiveness of our proposed method.

This work not only highlights the potential benefits of advanced orchestration and automation strategies but also paves the way for future research to further enhance microservices placement techniques. The implications of our findings suggest that continued innovation in this area will be essential for improving the performance and scalability of cloud-native applications, ultimately driving more efficient and resilient cloud infrastructures.

REFERENCES

[1] "Docker", 2024, [online] Available: https://www.docker.com/

[2] "Kubernetes", 2024, [online] Available: https://kubernetes.io/

[3] X. He, H. Xu, X. Xu, Y. Chen and Z. Wang, "An Efficient Algorithm for Microservice Placement in Cloud-Edge Collaborative Computing Environment", in IEEE Transactions on Services Computing, doi: 10.1109/TSC.2024.3399650.

[4] B. Natesha and R. M. R. Guddeti, "Adopting elitism-based genetic algorithm for minimizing multi-objective problems of iot service placement in fog computing environment", Journal of Network and Computer Applications, vol. 178, p. 102972, 2021.

[5] S. N. Srirama, M. Adhikari and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment", J. Netw. Comput. Appl., vol. 160, 2020.

[6] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources", ACM Trans. Internet Technol., vol. 20, no. 2, pp. 1-24, 2020.

[7] K. Cheng et al., "GeoScale : Microservice Autoscaling With Cost Budget in Geo-Distributed Edge Clouds", IEEE Trans. Parallel Distrib. Syst., p. 1–17, 2024.

[8] X. He, Z. Tu, M. Wagner, X. Xu and Z. Wang, "Online Deployment Algorithms for Microservice Systems With Complex Dependencies", in IEEE Transactions on Cloud Computing, vol. 11, no. 2, pp. 1746-1763, 1 April-June 2023, doi: 10.1109/TCC.2022.3161684

[9] A. M. Maia, Y. Ghamri-Doudane, D. Vieira and M. F. de Castro, "An improved multi-objective genetic algorithm with heuristic initialization for service placement and load distribution in edge computing", Comput. Netw., vol. 194, 2021.

[10] H. Liang and J. Chou, "HPA: Hierarchical Placement Algorithm for Multi-Cloud Microservices Applications", 2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Bangkok, Thailand, 2022, pp. 17-24, doi: 10.1109/CloudCom55334.2022.00013

[11] Z. Ding, S. Wang and C. Jiang, "Kubernetes-Oriented Microservice Placement With Dynamic Resource Allocation", in IEEE Transactions on Cloud Computing, vol. 11, no. 2, pp. 1777-1793, 1 April-June 2023, doi: 10.1109/TCC.2022.3161900

[12] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag and L. Veiga, "Practical Service Placement Approach for Microservices Architecture", in 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, Spain, 2017, pp. 401-410, doi: 10.1109/CCGRID.2017.28

[13] C. Tian et al., "Improving Simulated Annealing Algorithm for FPGA Placement Based on Reinforcement Learning", 2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference (ITAIC), Chongqing, China, 2022, pp. 1912-1919, doi: 10.1109/ITAIC54216.2022.9836761.

[14] Y. Li, H. Zhang, W. Tian and H. Ma, "Joint Optimization of Auto-Scaling and Adaptive Service Placement in Edge Computing", in IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS), Beijing, China, 2021, pp. 923-930, doi: 10.1109/ICPADS53394.2021.00121 .

[15] S. Pallewatta, V. Kostakos et R. Buyya, "MicroFog: A framework for scalable placement of microservices-based IoT applications in federated Fog environments", Journal of Systems and Software, Volume 209, 2024, 111910, ISSN 0164-1212, doi:10.1016/j.jss.2023.111910.

[16] R. Mahmud, S. Pallewatta, M. Goudarzi et R. Buyya, "iFogSim2 : An extended iFogSim simulator for mobility, clustering, and microservice management in edge and fog computing environments", J. Syst. Softw., p. 111351, mai 2022.

[17] S. Pallewatta, V. Kostakos et R. Buyya, "QoS-aware placement of microservices-based IoT applications in Fog computing environments", Future Gener. Comput. Syst., vol. 131, p. 121–136, juin 2022.

[18] Q. Yue, X. Liu, L. Fang, X. Wang and W. Hu, "A container service chain placement greedy algorithm based on heuristic information", J. Phys. Conf. Ser., vol. 1621, no. 1, 2020.

[19] T. Goethals, F. De Turck and B. Volckaert, "Extending Kubernetes Clusters to Low-Resource Edge Devices Using Virtual Kubelets", in IEEE Transactions on Cloud Computing, vol. 10, no. 4, pp. 2623-2636, 1 Oct.-Dec. 2022, doi: 10.1109/TCC.2020.3033807

[20] https://github.com/microservices-demo/microservices-demo

[21] L. Nathaniel, G. V. Perdana, M. R. Hadiana, R. M. Negara et S. N. Hertiana, "Istio API Gateway Impact to Reduce Microservice Latency and Resource Usage on Kubernetes", dans 2023 Int. Seminar Intell. Technol. Its Appl. (ISITIA), Surabaya, Indonesia, 26–27 juill. 2023.

[22] A. Aznavouridis, K. Tsakos et E. G. M. Petrakis, "Micro-Service Placement Policies for Cost Optimization in Kubernetes", dans Advanced Information Networking and Applications. Cham : Springer Int. Publishing, 2022, p. 409–420.

[23] C. Lübben, S. Schäffner and M. -O. Pahl, "Continuous Microservice Re-Placement in the IoT", NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, 2022, pp. 1-6, doi: 10.1109/NOMS54207.2022.9789780.

[24] "Linkerd", 2024, [online] Available: https://linkerd.io/2.15/overview/

[25] "Locust", 2024, [online] Available: https://locust.io/

[26] "rancher", 2024, [online] Available: https://www.rancher.com/