Future Generation Computer Systems 174 (2026) 108012

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Adaptive multi-objective swarm intelligence for containerized microservice deployment

Jiaxian Zhu [a], Weihua Bai [a], Huibing Zhang [b,*], Weiwei Lin [c], Teng Zhou [d,*], Keqin Li [e]

[a] School of Computer Science, Zhaoqing University, Zhaoqing, China
[b] The Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, China
[c] School of Computer Science and Engineering, South China University of Technology, Guangzhou, China
[d] School of Cyberspace Security (School of Cryptology), Hainan University, Haikou, China
[e] Department of Computer Science, State University of New York, New Paltz, NY, USA

ARTICLE INFO

Keywords:
Microservice deployment
Containerized microservice
Swarm intelligence optimization algorithms
Multi-objective optimization

ABSTRACT

Container-based microservice architecture is essential for modern applications. However, optimizing deployment remains critically challenging due to complex interdependencies among microservices. In this paper, we propose a formalized deployment model by systematically analyzing the interdependencies within Service Function Chains (SFCs). To achieve this, we design a novel swarm intelligence optimization algorithm, named Multi-objective Sand Cat Swarm Optimization with Hybrid Strategies (MSCSO-HS), for multi-objective optimization in microservice deployment. Our algorithm effectively optimizes inter-microservice communication costs and enhances container aggregation density to improve application reliability and maximize resource utilization. Extensive experiments demonstrate that MASCSO outperforms state-of-the-art algorithms for all optimization metrics. Our model achieves improvements of 23.76% in communication latency, 47.51% in deployment density, 38.70% in failure rate, 58.50% in CPU utilization, and 53.81% in RAM usage. The MASCSO framework not only enhances microservice performance and reliability but also provides a robust solution for resource scheduling in cloud environments for microservice deployment.

1. Introduction

Service Function Chains (SFCs) are a sequence of virtualized network functions that are interconnected to provide specific services. Container technology, integrated with microservice architecture, has revolutionized the integration, deployment, and scheduling of service function chains (SFCs) in modern applications, offering enhanced elasticity, flexibility, and maintainability. This container-based microservice deployment has emerged as a dominant paradigm in contemporary application development and a cornerstone for resource allocation and service scheduling across cloud, mobile edge, and fog computing environments. The deployment of containerized microservices necessitates addressing multiple objectives, such as optimizing performance, ensuring energy efficiency, facilitating maintenance, achieving load balancing, enhancing responsiveness, reducing costs, and enabling elastic invocation. Microservice architecture breaks down coarse-grained applications into fine-grained, functionally independent service units, which can be deployed autonomously. These microservices collectively form an SFC or business chain, representing a business workflow modeled as a Directed Acyclic Graph (DAG) of interdependent invocations. Container technology significantly improves efficiency by enabling resource sharing at both the operating system and image repository levels, thereby streamlining application packaging, deployment, and management.

Recent research advances container deployment and resource scheduling in microservice architectures by introducing novel algorithms and optimization strategies. Mendes et al. [1] enhance energy efficiency by integrating an on-demand oversubscription-based container scheduling algorithm into Docker Swarm. Their approach significantly improves resource utilization and reduces energy consumption, demonstrating the effectiveness of adaptive scheduling techniques in dynamic environments.

They allocate additional requests by oversubscribing CPU and memory resources, improving utilization and energy efficiency. Mao et al. [2,3] propose a Dynamic and Resource-Aware Placement Scheme (DRAPS) for Docker Containers to address system heterogeneity, resource utilization, and stability. However, DRAPS increases network consumption. Lv et al. [4] design a two-stage scheduling approach for container placement and reallocation in data centers, minimizing

* Corresponding authors.
E-mail addresses: zhanghuibing@guet.edu.cn (H. Zhang), teng.zhou@hainanu.edu.cn (T. Zhou).

https://doi.org/10.1016/j.future.2025.108012
Received 9 February 2025; Received in revised form 21 May 2025; Accepted 6 July 2025
Available online 21 July 2025

communication costs and balancing resource usage to boost utilization. While these studies advance deployment strategies and orchestration techniques, they prioritize isolated objectives such as resource utilization, energy efficiency, or performance. Existing approaches [5] neglect critical challenges, including selective container deployment, optimal microservice allocation, and dependency management between microservices.

Microservices architectures employ containers as execution environments for microservices, where communication dependencies, service interdependencies, and microservice failures critically influence system performance, resource utilization, operational costs, and energy consumption. Pallewatta et al. [6–8] propose a scalable, QoS-aware multi-objective set-based Particle Swarm Optimization (QMPSO) strategy to schedule microservices-based IoT applications in fog environments. Their approach optimizes completion time, budget, and throughput while maximizing limited fog resources through batch microservice placement. Faticanti et al. [9] address locality challenges in fog computing by proposing a heuristic approach to manage external object and computational resource access for client microservices. These contributions advance deployment strategies but require further integration of dependency management and failure resilience mechanisms to holistically address system-wide tradeoffs. Current research prioritizes addressing cross-domain containerized microservice dependencies to enhance user experience, QoS, and resource utilization. However, existing approaches focus narrowly on singular optimization objectives and neglect to unify critical factors such as data center performance, cluster resource utilization, service fault tolerance, and resource provider cost-profit dynamics within a holistic framework.

Determining how many instances each microservice needs and placing them on the right computational nodes requires understanding microservice dependencies and reducing communication overhead. In this paper, we propose a Multi-Objective Aware Optimization Algorithm (MASCSO) for container-based microservice placement. To achieve this, we design a Multi-objective Sand Cat Swarm Optimization with Hybrid Strategies (MSCSO-HS). Our MASCSO balances multiple objectives under limited resources, varying resource distributions, and diverse application requirements. It manages combinatorial dependencies and user demands in a Service Function Chain (SFC) by deciding the proper number of microservice instances based on load and performance. MASCSO optimizes microservice aggregation, reduces cross-container communication overhead, and lowers microservice failure rates. It then deploys these instances to appropriate containers, virtual machines, or physical servers. MASCSO improves resource utilization, system responsiveness, load balancing, and the reliability of application services.

The main contributions of this paper are as follows.

- We design a novel formal model for container-based microservice deployment by analyzing microservice interdependencies within Service Function Chains.
- We propose a novel swarm intelligence optimization algorithm, named Multi-objective Sand Cat Swarm Optimization with Hybrid Strategies (MSCSO-HS), which serves as the foundation for MASCSO to address multi-objective challenges in microservice deployment.
- We validate MASCSO's feasibility and effectiveness through extensive experiments, highlighting its potential to enhance resource utilization, improve system reliability, and optimize performance across diverse runtime environments.

2. Related work

In cloud computing, container-based microservice deployment strategies drive research and serve as a crucial technology. These strategies directly influence cloud data center performance indicators, such as system performance, resource efficiency, energy consumption, and cost.

2.1. Heuristic-based method

We model container-based microservice deployment as an NP-hard integer programming problem. Many researchers apply heuristic algorithms to solve such problems, offering significant benefits for microservice instance deployment.

Mahmoud et al. address system availability, scalability, resource utilization, and power consumption by proposing a Many-Objective Genetic Algorithm Scheduler (MOGAS) in [10]. This scheduler targets multiple objectives to produce solutions with better performance. MOGAS outperforms the Ant Colony Optimization (ACO) algorithm by allocating a higher proportion of tasks on average and reducing energy consumption.

Omogbai et al. examine the container placement scheduling problem in edge computing and shows in [11] that many optimization modeling frameworks convert the problem into multi-objective or graph network models solvable by algorithms. Meanwhile, scheduling algorithms use heuristic-based methods to rapidly find suboptimal solutions.

Zhou et al. propose GGA-HLSA-RW (GHW) as a novel genetic algorithm. GHW optimizes cloud utilization and energy consumption to tackle the Multiple Dimensional Resources Scheduling Problem (MDRSP) [12,13] and MOEA/D [12,14] to yield GHW-NSGA II and GHW-MOEA/D. Experimental results confirm the effectiveness of GHW's growth strategy and dimension-reduction approach in cloud computing.

Heuristic optimization algorithms excel at multi-objective optimization and demonstrate robust global search capabilities [15]. These algorithms also maintain a simple and efficient structure. Inherent randomness in parameter settings strongly influences performance, and it often leads to non-reproducible results. These algorithms usually produce near-optimal solutions, and the problem scale or encoding method can affect solution quality.

2.2. Metaheuristic/swarm intelligence optimization methods

Researchers employ metaheuristic and swarm intelligence optimization methods. These methods replicate biological evolution and group behavior. They explore complex solution spaces and often produce optimal or near-optimal solutions [16]. They also demonstrate strong capabilities in multi-objective optimization tasks for cloud resource scheduling [3].

Lin et al. introduced a multi-objective ant colony optimization algorithm for microservice invocation. This method addresses resource utilization, communication overhead, and service failure rate in microservice-based applications [17].

Ouyang et al. devised a service deployment strategy based on Accelerated Particle Swarm Optimization (APSO). This strategy enhances efficiency in cloud data centers by optimizing resource allocation and scheduling during service deployment [18,19]. Empirical results confirm that this model reduces system response times and improves resource utilization in cloud data centers.

Researchers have proposed additional algorithms based on metaheuristic and group intelligence optimization. These methods address microservice deployment and scheduling in cloud computing, the Internet of Things, and fog computing. They include the Evolutionary Game Algorithm [20], Non-dominated Sorting Genetic Algorithm-II (GA-NSGA-II) [21], the knowledge-driven evolutionary algorithm (MGR-NSGA-III) [22]. Researchers report that meta-heuristic and group intelligence algorithms provide strong advantages for complex and dynamically changing scheduling problems. Parameter tuning can unlock each algorithm's potential and mitigate its limitations.

2.3. Methods based on deep learning/reinforcement learning

Deep learning and reinforcement learning solutions for microservice deployment and container orchestration have attracted considerable attention, and many studies have examined these approaches.

Saravanan et al. introduced a hybrid optimum and deep learning approach for dynamic, scalable task scheduling (DSTS). This method addresses the task scheduling problem in cloud container environments [23]. Muthakshi proposed an Optimized Task Scheduling Algorithm with Deep Learning (OTS-DL). This algorithm automatically allocates containers [24]. Tom et al. presented a multi-agent environment with a deep reinforcement learning-based decision mechanism (Multi-Agent Deep Reinforcement Learning). This environment deploys containers to suitable cloud servers [25]. Cheng et al. developed a task scheduling strategy optimization algorithm based on an improved asynchronous advantage actor-critic (A3C). This algorithm tackles the multi-objective problem of minimizing average response time and energy consumption [26]. Deep reinforcement learning methods for container or microservice deployment offer strong decision-making capabilities but are vulnerable to changes in the execution environment. Their reliance on a specific training environment creates a significant challenge.

Existing methods struggle to balance multiple conflicting objectives in cloud-based environments. Communication overhead, container density, system failure rate, and CPU/RAM utilization often conflict, and single-objective or fixed-weight approaches fail to maintain optimal performance across all metrics. Many swarm intelligence algorithms succumb to local optima and show limited robustness under boundary conditions or dynamic changes. Most approaches also lack an efficient solution archival mechanism, which hinders the real-time utilization of nondominated solutions, especially in high-dimensional scenarios with large solution sets.

We integrate a vigilance/random-walk mechanism into swarm intelligence algorithms and employ a Hypergrid-based external archiving scheme. This adaptive scheduling framework addresses the challenges of balancing diverse objectives, escaping local optima, and improving scalable solution storage.

3. Designing the deployment problem formulation

The microservice architecture divides an application into small, independently deployable service units. These units form service function chains (SFCs) according to invocation dependencies. A container-based approach places the required instances on the appropriate virtual or physical machines.

3.1. Microservice deployment based on service function chains (SFCs)

Fig. 1 illustrates an example of microservice deployment. The left section presents two service function chains: $SF_1 : \{ms_1, ms_2, ms_3, ms_4, ms_6\}$ and $SF_2 : \{ms_1, ms_3, ms_4, ms_5\}$. The invocation dependencies are defined as: $F_1 : \{ms_1 \rightarrow ms_2,\ ms_2 \rightarrow ms_3,\ ms_3 \rightarrow ms_4,\ ms_4 \rightarrow ms_6\}$ and $F_2 : \{ms_1 \rightarrow ms_4,\ ms_4 \rightarrow ms_3,\ ms_3 \rightarrow ms_5\}$. Users $User_1$ and $User_2$ access SF_1 and SF_2, respectively.

The orchestrator deploys microservice instances on specific nodes, such as: $Node_1 : \{ms_1, ms_3\}$, $Node_2 : \{ms_1, ms_4\}$, and $Node_3 : \{ms_2, ms_5, ms_6\}$. The deployment process considers constraints, including available resources such as CPU, memory, network bandwidth, and storage capacity. Communication latencies vary by node due to distinct network conditions and data transfer volumes. Each microservice has unique resource requirements that dictate container allocation.

The right section of Fig. 1 presents the logical architecture of the container-based microservice deployment model in a cloud environment. This model comprises four layers: the User Layer, Workload Manager Layer, Resource Pool Control Layer, and Resource Infrastructure Layer. The Workload Manager Layer acquires microservice instances on demand, orchestrating containers based on user requests. The Resource Pool Control Layer employs the MASCSO algorithm to determine optimal deployment and scheduling plans executed in the Resource Infrastructure Layer.

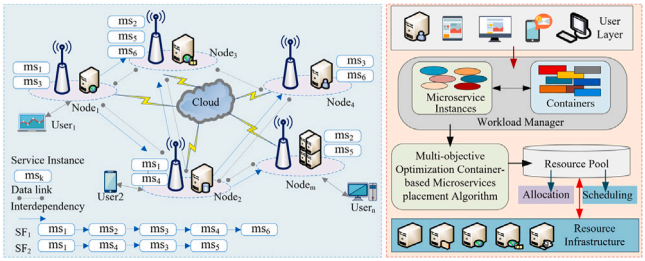

Fig. 1. Microservice instance deployment application scenario.

3.2. Container-based microservice deployment model

The multi-objective-aware microservice deployment algorithm facilitates container-based deployment. Users submit requests, and the system evaluates the resource demands of SFC microservices. The algorithm targets optimal configurations and quantities of microservices and execution containers while adhering to resource and deployment constraints. This approach minimizes system response times, enhances cluster resource utilization, and reduces service failure rates, ensuring a high-quality user experience.

Each service function chain consists of sequentially ordered microservices. Let SF_i represent the ith requested application, where $1 \leq i \leq M$, with M denoting the total number of user-requested applications.

Each application SFi utilizes the tuple (MS, F) to define its microservice composition:

- MS denotes the set of microservices within the service function chain, expressed as $MS = \{ms_j \mid 1 \leq j \leq n_{SFi}\}$, where $n_{SFi} = |MS|$ represents the total number of microservices in SFi.
- F encapsulates the functional dependencies among these microservices, defined as

$$F = \{\langle ms_i, ms_j \rangle \mid i, j \in [1, n_{SFi}], i \neq j\}.$$

The notation $\langle ms_i, ms_j \rangle$ indicates a call from microservice ms_i to microservice ms_j. These dependencies are stored within the matrix $F' \in \mathbb{R}^{n_{SFi} \times n_{SFi}}$, where $f_{ij} = 1$ if the dependency $\langle ms_i, ms_j \rangle$ exists.

The visual representation shows SF_i as a directed acyclic graph (DAG). The set MS forms the vertex set, and the function dependencies F produce the directed edges. Each application includes a group of microservices connected by call dependencies. This structure yields a DAG.

The system represents each microservice ms_k with a quadruple $\left(r_k^{\text{CPU}}, r_k^{\text{RAM}}, v_k, v_k^T\right)$:

- r_k^{CPU} and r_k^{RAM} denote the minimum CPU and RAM resources a server needs to handle ms_k. The server must have at least r_k^{CPU} and r_k^{RAM} available.
- v_k indicates the number of concurrent requests that ms_k can handle.
- v_k^T specifies the threshold number of requests that ms_k can process before reaching maximum concurrency limits imposed by its container.

Each server node NS_p appears as a tuple (Rs_p, Pos_p).

- $Rs_p = \{(rs_p^{\mathrm{CPU}}, rs_p^{\mathrm{RAM}}) \mid 1 \le p \le M\}$, where M indicates the number of server nodes. The parameters rs_p^{CPU} and rs_p^{RAM} specify the remaining CPU and RAM resources of NS_p. The system ensures $rs_p^{\mathrm{CPU}} \le Rs_p^{\mathrm{CPU}}$ and $rs_p^{\mathrm{RAM}} \le Rs_p^{\mathrm{RAM}}$, with Rs_p^{CPU} and Rs_p^{RAM} denoting the total resource allocations for NS_p.
- Pos_p marks the node's network location and affects communication latency. The network distance between NS_i and NS_j is $\mathrm{Ds}_{ij} = \|\mathrm{Pos}_i - \mathrm{Pos}_j\|$, which illustrates the latency coefficient between them.

Let U_{rms_k} denote the number of requests for microservice ms_k. The system deploys and executes microservice instances within containers. The variable v_k indicates the number of concurrent requests for ms_k, while v_k^T represents the concurrency threshold of the deployed container.

The container remains stable when the condition $v_k^T \ge (U_{rms_k} \cdot v_k)$ is satisfied. Conversely, it enters an overloaded state when $v_k^T < (U_{rms_k} \cdot v_k)$. In response to overload conditions, the system dynamically scales the number of container instances to achieve load balancing. The number of container instances to be expanded, denoted as ExC_{Conc}, is given by:

$$ExC_{Conc} = \left\lceil \frac{U_{rms_k} \cdot v_k}{v_k^T} \right\rceil. \tag{1}$$

The container-based microservice deployment scheme $CSps$ addresses constraints for user-requested microservices, the resources they require, and the resources available on servers. This scheme is defined as follows:

$$\mathrm{CSps} = \left[\mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{X}_S\right]^T. \tag{2}$$

In this model, let $\mathrm{X}_i \in \mathbb{R}^{1\times M}$ for $i = 1, 2, \ldots, S$. The term S, defined as $S = \sum_{k=1}^{M} n_{SF_k}$, represents the total number of microservices across all applications, where $|ms|$ refers to the overall set of microservices. The system defines M as the total number of server nodes available for hosting microservices. Thus, $\mathrm{CSps} \in \mathbb{R}^{S\times M}$.

We define $\mathrm{alloc}(ms_j)$ as the set of nodes that are responsible for hosting the microservice ms_j along with its execution container Con_c, thus enabling its operational functionality. The system applies relevant constraints when deciding whether NS_p belongs to $\mathrm{alloc}(ms_j)$. The node NS_p appears in $\mathrm{alloc}(ms_j)$ when it hosts ms_j, and does not appear if it does not host ms_j. Each element x_{ij} of the matrix CSps characterizes the deployment status of microservice ms_i on node NS_j as follows:

$$x_{ij} = \begin{cases} 1 & NS_j \in \mathrm{alloc}(ms_i) \\ 0 & NS_j \notin \mathrm{alloc}(ms_i) \end{cases} \tag{3}$$

where the variable $x_{ij} \in [0, 1]$ indicates whether the microservice ms_i is deployed (1) or not deployed (0) on node NS_j.

In this container-based microservice deployment model, we deploy S microservices ms along with their corresponding containers from all applications SF across M server nodes NS. Each server node can host multiple microservices or container instances. We assert that identifying a valid container-based microservice deployment scheme $CSps$ under these constraints is NP-hard. This claim is supported by reducing our problem to the well-known NP-hard problem of graph coloring, which establishes its computational complexity.

3.3. Objective evaluation function

We develop a multi-objective microservice deployment optimization algorithm. The algorithm targets four objectives: minimizing microservice communication latency, maximizing container deployment density, enhancing resource utilization within the cluster, and reducing the microservice failure rate.

3.3.1. Communication latency evaluation function for microservices

The cluster connects its nodes through high-speed networks, reducing communication latency among microservices, accelerating system response, and enhancing user experience. Index i denotes microservice ms_i, while i' refers to $ms_{i'}$ called by ms_i, and j identifies server nodes. Microservice ms_i is hosted in container instance Con_i with its container image sourced from node NS_j. The communication overhead comprises four factors: (1) container image size SiCon_i, (2) network distance Ds'_{jj} between nodes, (3) data volume $Dd_{ii'}$ per interaction, and (4) access frequency v_{ii} for the invoked microservice.

Assuming $\mathrm{alloc}(ms_j) \ge 1$ implies that a microservice's container instance can operate across multiple server nodes. The average data transmission time between container pairs influences overall communication latency. The call dependency matrix $F' \in \mathbb{R}^{S\times S}$ captures interdependencies among all microservices SF.

As the number of microservices increases, the frequency of communication among them rises, resulting in higher network traffic and system load. Additionally, network latency may worsen due to communications spanning multiple nodes. To evaluate this, we can apply the communication latency evaluation function (Eq. (4)) to quantify changes in communication costs with increasing microservice complexity. Specifically, as S increases, the call dependency matrix F' leads to more interactions, thereby significantly increasing communication delays and overhead. The experimental section will provide specific results comparing communication costs for different microservice counts.

The communication latency evaluation function $\mathcal{F}_{\mathrm{cde}}(X)$, defined for deployment scheme $X \in \mathrm{CSps} \subset \mathbb{R}^{S\times M}$, quantifies communication latency as follows:

$$\mathcal{F}_{\mathrm{cde}}(X) = \sum_{i=1}^{S}\sum_{j=1}^{M}\left\{\left(\frac{x_{ij}}{ExC_{\mathrm{Con}_i}} \cdot \sum_{k\in A_j}^{M}\left(Ds_{jk}\cdot \mathrm{SiCon}_i\right)\right) + \sum_{i'\neq i}^{S} f_{ii'} \cdot \frac{x_{ij}}{ExC_{\mathrm{Con}_i}} \cdot \left(\sum_{k\in A_{i'}}^{M}\left(Dd_{jk}\cdot Ds_{jk}\cdot v_{jk}\right)\right)\right\}, \tag{4}$$

where $A_j = \{k \neq j \mid NS_k \in \mathrm{alloc}(ms_k)\}$, and $A_{i'} = \{k \mid NS_k \in \mathrm{alloc}(ms_{i'})\}$.

3.3.2. Container deployment density evaluation function

Optimizing container deployment increases aggregation, leverages resources more effectively, and reduces dispersion. The extended call dependency matrix $\sum F' \in \mathbb{R}^{S\times S}$ covers every microservice in SF. If $f_{ij} = 1$, the directed edge weight becomes W_{ij}, and this value reflects the closeness of calls between the two microservices. These weights form the call dependency weight matrix $W \in \mathbb{R}^{S\times S}$.

In cloud computing environments, deploying microservice containers from SFC applications on the same server node or within the same data center (CDC) aligns supply and demand more cost-effectively. This practice reduces data transmission costs between services and curbs network resource consumption. Centralized deployment boosts resource utilization on specific nodes. We measure edge distances between microservices to evaluate container aggregation and quantify microservice concentration. We define two scenarios to measure node edge distance. If $f_{ii} = 1$ and $\mathrm{alloc}(ms_i) \cap \mathrm{alloc}(ms_i) = NS_k$, then two microservices with a direct call dependency share the same server node NS_k using containers, and the edge distance is W_{ii}. If $f_{ii} = 1$ and $\mathrm{alloc}(ms_i) \cap \mathrm{alloc}(ms_{i'}) = \emptyset$, we define the edge distance as follows:

$$D_k = \frac{x_{ij}}{\prod^{S}_{\substack{k=1\\k\neq j}\\ NS_k\in \mathrm{alloc}(\{ms_i\})'}(f_{ii'}\cdot Ds_{jk})} W_{ii'}. \tag{5}$$

Eq. (5) shows that a larger edge distance in a deployment scheme signifies higher container deployment density. We use the reciprocal of the sum of container edge distances as a quantification metric; smaller

values indicate better performance. We define the deployment intensity evaluation function for a deployment scheme, denoted as $\mathcal{F}_{\text{dei}}(X)$, as follows:

$$\mathcal{F}_{dei}(X) = 1/\sum_{i=1}^{S} f_{ii'}\left(\left(\sum_{j=1}^{M}\frac{x_{ij}}{A(i,j)}W_{ii'}\right)+W_{ii'}\right), \tag{6}$$

where $A(i,j)=\prod^{S}_{\substack{k=1\wedge k\neq j\\ \wedge Ns_k\in \text{alloc}\left(ms_{i'}\right)}}\left(f_{ii'}\cdot Ds_{jk}\right)$.

3.3.3. Cluster resource utilization evaluation function

Appropriate allocation and scheduling of computational resources, including CPU and memory, increase efficiency, reduce waste, and optimize system performance. The average utilization rates of these resources quantify the resource usage of a deployment scheme X in the cluster. For optimization objectives, we define the evaluation functions $\mathcal{F}_{UCPU}(X)$ and $\mathcal{F}_{URAM}(X)$ as follows:

$$\mathcal{F}_{UCPU}(X)=\frac{\sum_{i=1}^{S}\frac{Ur_{ms_i}\cdot v_k}{ExC_{Con_i}}\cdot r_i^{CPU}\cdot x_{ij}}{\sum_{j=1}^{M} rs_j^{CPU}}, \tag{7}$$

$$\mathcal{F}_{URAM}(X)=\frac{\sum_{i=1}^{S}\frac{Ur_{ms_i}\cdot v_k}{ExC_{Con_i}}\cdot r_i^{RAM}\cdot x_{ij}}{\sum_{j=1}^{M} rs_j^{RAM}}. \tag{8}$$

3.3.4. Microservice failure rate evaluation function

Optimizing deployment strategies and enhancing fault-tolerance mechanisms lowers the failure probability in microservice operations and boosts reliability and stability. Common causes of microservice failures include (1) errors in execution containers and (2) failures in the server nodes hosting these containers. We denote the error probability of container instance Con_i by ρ_{Con_i} and the failure rate of server node NS_p by ρ_{NS_p}. Let Con_i be the execution container instance for microservice ms_i with $NS_p\in\text{alloc}(ms_i)$. Then

$$\rho_{\text{Con}_i}=\frac{v_i^T}{ExC_{\text{Con}_i}}\cdot\rho_{NS_p}.$$

We define the failure rate evaluation function for a deployment plan X, denoted by $\mathcal{F}_{\text{ineff}}(X)$, as follows:

$$\mathcal{F}_{\text{ineff}}(X)=\sum_{i=1}^{S}\sum_{j=1}^{M}\left(\rho_{\text{Con}_i}+\rho_{NS_j}\right)\cdot x_{ij} \tag{9}$$

Substituting ρ_{Con_i} into Eq. (9) yields:

$$\mathcal{F}_{\text{ineff}}(X)=\sum_{i=1}^{S}\sum_{j=1}^{M}\left(\left(\sum_{A_i}^{M}\frac{v_i^T}{ExC_{\text{Con}_i}}\cdot\rho_{NS_K}\right)+\rho_{NS_j}\right)\cdot x_{ij} \tag{10}$$

where $A_i=\{k\mid k=1 \text{ and } NS_k\in\text{alloc}(ms_i)\}$.

3.3.5. Objective function of the multi-objective awareness model

Optimizing container-based microservice deployment improves system response time, resource utilization, and reduces service failure rates. We exclude $\mathcal{F}_{UCPU}(X)$ and $\mathcal{F}_{URAM}(X)$ from the objective function, as they are influenced by container density. The final model's objective function and constraints are defined as follows:

$$F_{\min}(X)=\min_{ob\in\{cde,dei,ineff\}}\sum F_{ob}(X) \tag{11}$$

Subject to:

$$\sum_{i=1}^{S}(r_i^{\text{CPU}}\cdot x_{ij})\le Rs_j^{\text{CPU}}, NS_j\in\text{alloc}(ms_i) \tag{11-1}$$

$$\sum_{i=1}^{S}(r_i^{\text{RAM}}\cdot x_{ij})\le Rs_j^{\text{RAM}}, NS_j\in\text{alloc}(ms_i) \tag{11-2}$$

$$\sum_{i=1}^{S}(Ur_{ms_i}\cdot v_i)\cdot x_{ij}\le v_i^T, NS_j\in\text{alloc}(ms_i) \tag{11-3}$$

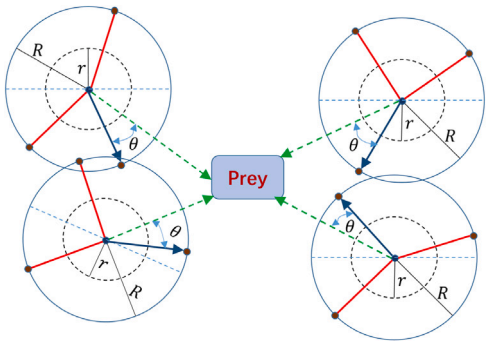

Fig. 2. Illustration of position update mechanisms for individuals in a sand cat swarm during a predation event.

$$\sum_{i=1}^{S} x_{ij}=1, NS_j\in\text{alloc}(ms_i) \tag{11-4}$$

$$\sum_{j=1}^{M} x_{ij}\ge 1, NS_j\in\text{alloc}(ms_i) \tag{11-5}$$

Constraints (11-1) and (11-2) limit CPU and RAM usage by microservices on each server node to prevent exceeding capacity. Constraint (11-3) ensures stable execution containers during load balancing. Constraint (11-4) guarantees that microservice instances on the same server node remain distinct. Constraint (11-5) facilitates hosting of multiple microservices on a single server node.

These constraints necessitate a container-based microservice deployment plan, constituting a multi-objective optimization problem. Section 2 reviews metaheuristic and population-based optimization algorithms that address these tasks and efficiently identify Pareto-optimal solutions. We propose a new hybrid strategy-enhanced multi-objective sand cat swarm optimization algorithm, which quickly identifies multi-objective-aware deployment schemes with a problem size of $\mathbb{R}^{S\times M}$ (i.e., S microservices and M server nodes), denoted as $\text{CSps}=[\text{X}_1,\text{X}_2,\ldots,\text{X}_S]^T$.

4. Multi-objective awareness solution-hybrid strategy enhanced multi-objective sand cat swarm optimization algorithm

4.1. Basic SCSO algorithm

SCSO, introduced by Seyyedabbasi et al. is a novel swarm intelligence optimization algorithm inspired by the predation behavior of sand cats [27]. This algorithm effectively mimics the hunting behaviors of sand cats, showcasing significant optimization performance. Fig. 2 illustrates the position-update mechanism among individuals in the sand cat swarm during predation.

The fundamental control parameters of the basic SCSO algorithm and their respective updating formulas are as follows:

$$\overrightarrow{r_G}=S_M-\left(\frac{S_M-\text{iter}_c}{\text{iter}_{\text{Max}}}\right), \tag{12}$$

$$\vec{R}=2\times\overrightarrow{r_G}\times\text{rand}(0,1)-\overrightarrow{r_G}, \tag{13}$$

$$\vec{r}=\overrightarrow{r_G}\times\text{rand}(0,1), \tag{14}$$

where S_M mimics the auditory sensitivity of sand cats, typically set at $S_M=2.0$; $iter_c$ denotes the current iteration number; and $iter_{Max}$ represents the predetermined maximum number of iterations, usually set to 200 or 400. The parameter $\overrightarrow{r_G}$ emulates the sensing range of sand cats during their prey hunting, which linearly decreases from 2 to 0 as the iteration number $iter_c$ increases. The control parameter $\vec{R}$ is utilized during the hunting or attacking phase. The variable $\vec{r}$ indicates the sensitivity range of the sand cats.

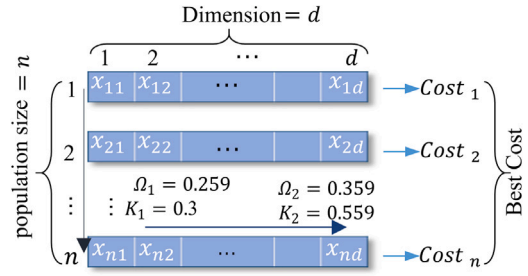

Fig. 3. Example of initialized population and process solution matrix.

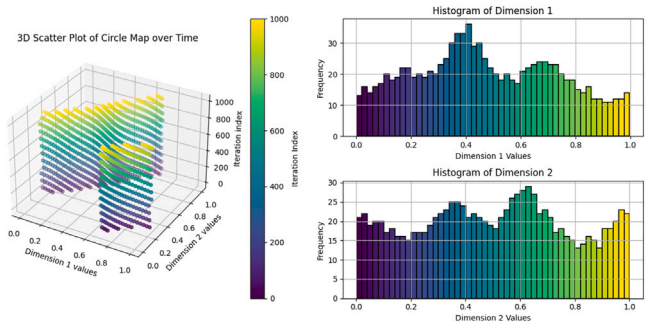

Fig. 4. The distribution of the Circle chaos mapping.

The predation behavior consists of two phases: hunting prey and attacking. The corresponding mathematical models for these phases are defined as follows:

(a) Mathematical model for the hunting prey phase:

$$\overrightarrow{\mathrm{Pos}}_c^{(t+1)} = \vec{r} \cdot \left(\overrightarrow{\mathrm{Pos}}_{bc}^{(t)} - \mathrm{rand}(0,1) \cdot \overrightarrow{\mathrm{Pos}}_c^{(t)} \right), \tag{15}$$

where $\overrightarrow{\mathrm{Pos}}_c^{(t)}$ represents the current position of the individual within the population during this iteration; $\overrightarrow{\mathrm{Pos}}_{bc}^{(t)}$ denotes the position of the best candidate individual within the population. Eq. (15) reflects the iterative formula for the searching behavior, with $\overrightarrow{\mathrm{Pos}}_c^{(t+1)}$ indicating the updated position of the individual. The sensitivity $\vec{r}$ ensures convergence effectiveness during the prey-hunting phase.

(b) Mathematical model for the attacking prey phase:

$$\overrightarrow{P_r} = \left| \mathrm{rand}(0,1) \cdot \overrightarrow{\mathrm{Pos}_{bb}}^{(t)} - \overrightarrow{\mathrm{Pos}}_c^{(t)} \right|, \tag{16}$$

$$\overrightarrow{\mathrm{Pos}}_c^{(t+1)} = \overrightarrow{\mathrm{Pos}}_b^{(t)} - \vec{r} \cdot \overrightarrow{P_r} \cdot \cos(\theta), \tag{17}$$

where $\overrightarrow{P_r}$ represents a random individual located near the combination of the optimal individual $\overrightarrow{Pos}_b^{(t)}$ (i.e., the optimal solution) and the current position $\overrightarrow{Pos}_c^{(t)}$; $\theta \in [0°, 360°]$ represents the directional angle of movement.

Combining Eqs. (15) to (17), based on the stage control parameter $\vec{R}$, the strategy for updating positions in the next round, selecting either the hunting or attacking phase's mathematical model, can be defined as follows:

$$X^{(t+1)} = \begin{cases} \overrightarrow{Pos}_b^{(t)} - \vec{r} \cdot \overrightarrow{P_r} \cdot \cos(\theta), |\vec{R}| \le 1 \\ \vec{r} \cdot \left(\overrightarrow{Pos}_{bc}^{(t)} - \mathrm{rand}(0,1) \cdot \overrightarrow{P_c}(t) \right), |\vec{R}| > 1 \end{cases} \tag{18}$$

where $X^{(t+1)} \in \mathrm{CSps}$ represents the solution for iteration $t+1$.

4.2. Hybrid strategy enhanced MASCSO-HS algorithm

4.2.1. Optimizing population diversity

To maximize the diversity of the initial population, we carefully choose chaotic sequences with high positive Lyapunov exponents. It optimizes population diversity with the Circle map, a low-dimensional chaotic system. Fig. 3 illustrates an example of the initialized population and a solution matrix during the solving process.

In the population initialization process, chaotic mappings with different coefficients are applied to each individual and across various dimensions within each individual, as shown in Fig. 4. The population initialization is strategically defined through the implementation of a dual-circle chaotic mapping:

$$x_{i,1} = x_{i-1,1} + \Omega_1 - \frac{\mathrm{K}_1}{2\pi} \sin\left(2\pi \cdot x_{i-1,1}\right), \quad \forall i \in \mathrm{population}(2,n), \tag{19}$$

$$x_{i,j} = x_{i,j-1} + \Omega_2 - \frac{\mathrm{K}_2}{2\pi} \sin\left(2\pi \cdot x_{i,j-1}\right), \quad \forall i \in \mathrm{population}(1,n), \forall j \in \mathrm{dimension}(2,d), \tag{20}$$

where d represents the dimension of the solution, and n, denotes the size of the population, that is, the number of individuals. Ω_1, K_1, Ω_2, K_2 are the coefficients for the Circle chaotic mapping applied to the population and the individuals, respectively, with $x_{1,1} = \mathrm{rand}(0,1)$. Fig. 4 shows the frequency histogram of the scaled sequence values and the scatter plot of the population distribution.

In Fig. 4, the population distribution of the Circle chaos mapping initialization sequence is uniform, showing greater diversity compared to the basic SCSO algorithm. This is achieved through the initialization of dual circle chaos mapping.

4.2.2. Enhancing global search performance

The spiral search strategy enhances exploration efficiency in metaheuristic algorithms by simulating spiral motion, thus avoiding local optima and increasing the probability of finding global optima. The parameters relevant to the spiral search strategy are defined as follows:

$$\overrightarrow{l^{(t)}} = \rho \cdot \left(\overrightarrow{\mathrm{Pos}}_{bc}^{(t)} - \overrightarrow{\mathrm{Pos}}_b^{(t)} \right), \tag{21}$$

$$z^{(t)} = e^{\kappa \cdot \cos\left(2\pi \cdot (1 - t/\mathrm{iter}_{\mathrm{Max}})\right)}, \tag{22}$$

where $\overrightarrow{l^{(t)}}$ denotes the differential operation between the position of an individual, $\overrightarrow{\mathrm{Pos}}_{bc}^{(t)}$, and the best individual's position, $\overrightarrow{\mathrm{Pos}}_b^{(t)}$, scaled by a factor $\rho \in [0.1, 4]$. The function $z^{(t)}$ combines a sinusoidal function with an exponential factor, facilitating exploration of new regions while revisiting previous ones in search of the global optimum. The parameter κ controls the amplitude of the sinusoidal wave, with $\kappa \in [3, 5]$.

Integrating the spiral search strategy into the prey-hunting phase of the basic SCSO algorithm results in the following mathematical model:

$$\overrightarrow{\mathrm{Pos}}_c^{(t+1)} = e^{z^{(t)} \cdot \overrightarrow{l^{(t)}}} \cdot \sin\left(\frac{1}{3}\pi \cdot \overrightarrow{l^{(t)}}\right) \cdot \vec{r} \cdot \left(\overrightarrow{\mathrm{Pos}}_{bc}^{(t)} - \mathrm{rand}(0,1) \cdot \overrightarrow{\mathrm{Pos}}_c^{(t)} \right). \tag{23}$$

The inclusion of the spiral search strategy in the MASCSO-HS algorithm improves its ability to explore unknown regions, escape local optima, and enhance overall global search performance compared to the basic SCSO algorithm.

4.2.3. Optimization of boundary convergence performance

During each iteration, some individuals may end up marginalized or in precarious states. Integrating a vigilance mechanism allows individuals to transition from "dangerous" to "safe" positions, facilitating random movements within the group. This process uses a sparrow vigilance mechanism to enhance boundary convergence performance.

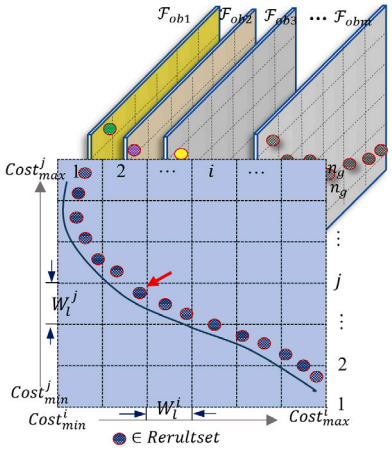

Fig. 5. The multi-objective function adaptive hypergrid Pareto front storage.

The mathematical model governing this process is defined as follows:

$$\overrightarrow{\mathrm{Pos}}_{rs}^{(t+1)} = \begin{cases} \overrightarrow{\mathrm{Pos}}_{b}^{(t)} + \mathrm{rand}(0,1)\cdot q \cdot \left|\overrightarrow{\mathrm{Pos}}_{rs}^{(t)} - \overrightarrow{\mathrm{Pos}}_{b}^{(t)}\right| \\ \quad \text{if } \mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{rs}^{(t)}\right) > \mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{b}^{(t)}\right), \\ \overrightarrow{\mathrm{Pos}}_{rs}^{(t)} + \tau \cdot \left(\dfrac{\left|\overrightarrow{\mathrm{Pos}}_{rs}^{(t)} - \overrightarrow{\mathrm{Pos}}_{ws}^{(t)}\right|}{\left(\mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{rs}^{(t)}\right) - \mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{ws}^{(t)}\right) + \varepsilon\right)} \right) \\ \quad \text{if } \mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{rs}^{(t)}\right) = \mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{b}^{(t)}\right), \end{cases} \tag{24}$$

where $\overrightarrow{\mathrm{Pos}}_{rs}^{(t)}$ represents the individuals randomly selected with a probability p_s, where $rs \in \mathrm{select}\left(p_s \cdot n\right)$, and $p_s \in [0.2, 0.4]$. The function $\mathcal{F}_{\min}\left(X^{(t)}\right) \in \mathbb{R}^d$ denotes the objective function values of all individuals at the tth iteration, with $\mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{ws}^{(t)}\right) = \max\left(\mathcal{F}_{\min}\left(X^{(t)}\right)\right)$. The step length control parameter $q \in \mathbb{R}^d$, with $q_i \sim N(0,1)$, and the direction and step length control parameter $\tau \in [-1, 1]$, where $\varepsilon = 1 \times 10^{-50}$.

Eq. (24) describes a scenario where a randomly selected individual is in a marginal "dangerous" position when $\mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{rs}^{(t)}\right) > \mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{b}^{(t)}\right)$. In this case, the individual moves toward the best position to avoid danger. Conversely, when randomly selected individuals need to converge to avoid potential danger, specifically when $\mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{rs}^{(t)}\right) = \mathcal{F}_{\min}\left(\overrightarrow{\mathrm{Pos}}_{b}^{(t)}\right)$, they wander toward the vicinity of other individuals to mitigate risk.

4.2.4. Multi-objective adaptive hypergrid pareto front storage strategy

Adaptive grid strategies, as detailed in [28,29], utilize a hypergrid dimension value, denoted as $n_g \in [10, 30]$. The value of n_g affects the storage density of the Pareto solution set. A hypergrid is constructed for each objective function, with the grid dimension corresponding to the number of objective functions, $|\mathcal{F}_{ob}|$.

The storage strategy for each dimension of the hypergrid involves defining the maximum and minimum values of the objective function for the current non-dominated solution set as $\mathrm{Cost}_{\max}^{\mathcal{F}_{ob}}$ and $\mathrm{Cost}_{\min}^{\mathcal{F}_{ob}}$, respectively. The hypergrid expansion amount, E_l, and the width of each grid segment, W_l, are given by:

$$\mathrm{Cost}_{\max}^{\mathcal{F}_{ob}} = \max_{|\mathcal{F}_{ob}|}\left(\mathrm{Cost}^{\mathcal{F}_{ob}}\left(\mathrm{ResulSet}^{t}\right)\right), \tag{25}$$

$$\mathrm{Cost}_{\min}^{\mathcal{F}_{ob}} = \min_{|\mathcal{F}_{ob}|}\left(\mathrm{Cost}^{\mathcal{F}_{ob}}\left(\mathrm{ResulSet}^{t}\right)\right), \tag{26}$$

$$E_l = \eta \cdot \left(\mathrm{Cost}_{\max}^{\mathcal{F}_{ob}} - \mathrm{Cost}_{\min}^{\mathcal{F}_{ob}}\right), \tag{27}$$

$$W_l = \left(\left(\mathrm{Cost}_{\max}^{\mathcal{F}_{ob}} + E_l\right) - \left(\mathrm{Cost}_{\min}^{\mathcal{F}_{ob}} - E_l\right)\right) / \left(n_g - 1\right). \tag{28}$$

Fig. 5 illustrates the adaptive hypergrid storage for a Pareto front solution set with $|\mathcal{F}_{ob}| = m$, where each non-dominated solution is indexed to its grid number based on its values, allowing identification of the grid with the least solutions, optimal for the next iteration, as shown by the red arrows in Fig. 5. Given the independence of non-dominated solutions, their positions in the hypergrid may cluster or disperse, with the strategy for selecting optimal individuals based on density accumulation and a threshold $\lambda = 1/3$. If $|\mathrm{ResulSet}^t| > \lambda \cdot n$, individuals in sparse hypergrids are selected; otherwise, those in dense hypergrids are chosen. When $\lambda = 1$, dense hypergrids are prioritized, while $\lambda = 0$ favors sparse hypergrids. During this iterative search, new solutions $newP^{t+1}$ are integrated into the hypergrid to form the new solution set $\mathrm{ResultSet}^{t+1}$, determined by their positions relative to existing solutions.

(1) If the new solution $newP^{t+1}$ dominates any existing solution $oldP^{t}$, the dominated solutions are removed, and the new solution is added. The updated solution set is defined as $ResulSet^{t+1} = \left(ResulSet^{t} - oldP^{t}\right) \cup newP^{t+1}$.

(2) If $|ResulSet^{t+1}| > n_g$, then execute select $\left(\mathrm{maxcount}\left(\mathrm{GridIndex}\left(\mathrm{ResulSet}^{t}\right)\right)\right)$ to remove a solution from the most densely mapped hypergrid.

(3) If $|ResulSet^{t+1}| < n_g$, the new solutions are directly added, updating the hypergrid's boundaries for the new iteration defined as:

$$\mathrm{Lower} = \mathrm{Cost}_{\min}^{\mathcal{F}_{ob}} - E_l, \tag{29}$$

$$\mathrm{Upper} = \mathrm{Cost}_{\max}^{\mathcal{F}_{ob}} + E_l, \tag{30}$$

$$\mathrm{GIndex}(X) = \left\lceil \frac{\mathrm{Cost}^{\mathcal{F}_{ob}}(X)}{W_l} \right\rceil \quad \forall X \in \mathrm{ResulSet}^{t+1}. \tag{31}$$

Eqs. (29) and (30) define the lower and upper bounds for the new hypergrid, while Eq. (31) refers to the mapping indices of individual solutions in the new set. The adaptive hypergrid storage strategy is a meta-heuristic approach designed to preserve elitism, with its size and shape adapting based on solution values to maintain diversity and leverage high-quality solutions in uncovering superior results.

This enables the use of high-quality solutions to uncover superior solutions.

4.3. Implementation and testing validation of MASCSO-HS algorithm

4.3.1. Implementation of MASCSO-HS algorithm

The algorithm integrates population diversity optimization, a spiral search strategy, a sparrow vigilance mechanism, and a multi-objective adaptive hypergrid storage strategy. The pseudocode for this algorithm is presented as Algorithm 1.

The original SCSO algorithm can be approximated as $O(n \cdot m)$, where n is the population size and m is the number of iterations. In our proposed MASCSO-HS, the integration of additional strategies for enhanced exploration and exploitation adds overhead. The complexity of MASCSO-HS can be represented as $O(n \cdot m + k)$, where k accounts for the additional computations introduced by the hybrid strategies, such as the spiral search strategy and the vigilance mechanism. Therefore, although MASCSO-HS has a slightly higher computational complexity, its superior performance in terms of convergence speed and solution quality positions it as a more effective approach in multi-objective optimization.

4.3.2. Validation and analysis of algorithm effectiveness

To validate the performance of the based algorithm of MASCSO-HS (referred to as SCHS) against the original SCSO [27] algorithm and compare it with other recent swarm intelligence algorithms, such as HHO, YDSE, WOA, and COA, we utilized both the CEC 2019 and CEC 2009 benchmark functions to comprehensively analyze SCHS's capabilities across different optimization scenarios. The experiments were

Table 1
Rank-sum test results for MASCSO-HS compared to other optimization algorithms on CEC 2019 functions.

Function	DBO	POA	SCSO	HHO	SABO	GWO	YDSE	WOA	PSO	COA
F1	6.25E−10	1	0.000313	1	1.21E−12	1.21E−12	1.21E−12	1.21E−12	1.21E−12	1
F2	4.55E−09	2.43E−05	2.81E−05	1.72E−12	3.02E−11	3.02E−11	3.02E−11	3.02E−11	3.02E−11	1.21E−12
F3	0.001003	0.055542	0.043581	0.000903	1.29E−06	0.061448	8.15E−11	0.000189	0.011227	3.59E−05
F4	0.000399	0.118812	0.090486	0.589446	0.133449	1.61E−10	1.96E−10	0.911708	3.33E−11	8.29E−06
F5	0.011228	3.69E−11	3.02E−11	3.02E−11	3.02E−11	5.49E−11	3.02E−11	3.02E−11	1	3.02E−11
F6	0.283778	0.864994	0.371077	9.51E−06	0.055546	1.43E−08	0.006972	9.06E−08	1.33E−10	8.15E−11
F7	0.12967	1.11E−06	0.074827	0.958731	8.15E−11	3.57E−06	0.002891	0.318304	2.49E−06	1.21E−10
F8	0.911709	0.000952	0.318304	0.000168	0.000141	9.21E−05	0.1809	0.000691	0.000117	1.11E−06
F9	0.10547	0.000189	0.258051	0.051877	0.559231	1.73E−06	0.318304	0.046756	9.83E−08	3.02E−11
F10	6.7E−11	4.69E−08	4.12E−06	1.16E−07	3.34E−11	3.02E−11	3.34E−11	1.21E−10	1.29E−09	5.49E−11

Algorithm 1 MASCSO-HS Algorithm

Input $F_{\min}(X)$, Dim;
Initialize $n, n_g, \alpha, \beta, p_s, S_M, \rho, \kappa, \eta$;
Initialize the population $X_i (i = 1, 2, \ldots, n)$ with Eq. (19) and Eq. (20);

for each X_i **do**
  ConstrainsTest X_i;
  CalculateCost X_i;
end for
DetermineDomination X^0;
CreateHyperGrid X^0; Eq.(25) to Eq.(30);
Save $X^0 \rightarrow$ ResulSet^0;
while $t <$ iter$_{\text{Max}}$ **do**
  for each search agent **do**
    Update the position by Eqs. (12)–(14) and Eqs. (21)–(23);
    Update the position with p_s and Eq. (24);
  end for
  DetermineDomination(X^t);
  CreateHyperGrid(X^t);
  Save ($X^t \rightarrow$ ResulSet^t);
  $t \leftarrow t + 1$;
end while
return $ResulSet^t$;

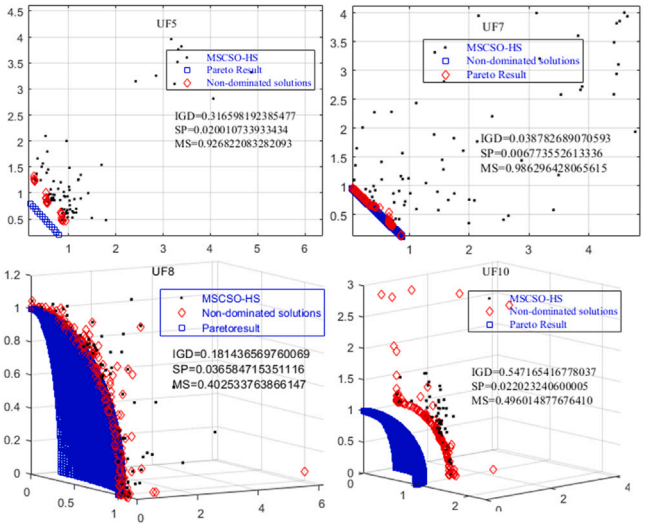

Fig. 6. Non-dominated Pareto front solution sets obtained by MASCSO-HS on UF5, UF7, UF8, and UF10.

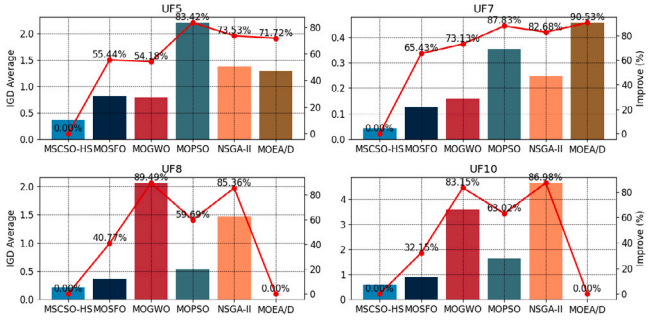

Fig. 7. Histograms of IGD values for various algorithms and the performance improvement line chart for MASCSO-HS on UF5, UF7, UF8, and UF10.

conducted using the publicly available CEC 2019 test suite, which primarily assesses single-objective performance, alongside the CEC 2009 test suite for multi-objective evaluation.

Based on the rank-sum test results presented in Table 1, the SCHS algorithm shows superior performance across various CEC 2019 functions, especially in accuracy, stability, and convergence speed. For example, SCHS outperformed SCSO in functions F1 and F2, achieving error metrics of 6.25×10^{-10} and 4.55×10^{-09} respectively. This is substantiated by a comprehensive analysis of optimal values, averages, and standard deviations, highlighting SCHS's rapid convergence to high-quality solutions and its strong adaptability in dynamic environments, making it well-suited for complex multi-objective optimization problems.

We verified the effectiveness of the MASCSO-HS algorithm in multi-objective optimization using the 10-function multi-objective optimization test benchmark set CEC 2009

[30]. Performance evaluation metrics such as Inverted Generational Distance (IGD) [28,31], Spacing (SP) [28,29], and Maximum Spread (MS) [29] were used as the assessment criteria. The detailed definitions of evaluation metrics can be found in Table 2.

The experimental data on the CEC 2009 Function shows that the MASCSO-HS algorithm has significant advantages over other algorithms. The comparison graphs of the non-dominated Pareto front solution sets obtained by MASCSO-HS and the true solution sets for test functions UF5, UF7, UF8, and UF10 are shown in Fig. 6. The bar graphs for the values and the improvement line charts for UF5, UF7, UF8, and UF10 are also presented in Fig. 7. The values achieved the best results in 9 out of the 10 test functions of CEC 2009, corresponding to a 90% success rate, despite the poorest performance on UF3. Particularly on UF5, UF7, UF8, and UF10, compared to other algorithms, the improvements average 67.66%, 79.92%, 68.82%, and 66.32%, respectively. Since the MOEA/D algorithm is unable to solve UF8 and UF10, the improvements for these functions are reported as 0% in Fig. 7.

The experimental data suggest that MASCSO-HS outperforms other algorithms with faster convergence and solution speed. This paper has validated the effectiveness and superiority of MASCSO-HS compared to other recent and typical algorithms. Our ongoing research and experimentation will continue to test further and fine-tune the algorithm's performance.

Table 2
Control parameters of meta-heuristics.

Parameter	Description	Setting value
iter $_{Max}$	Number of iterations	200
n	Population Size	100
Ω_1, K_1	Coefficient of Circle_1 chaos	0.259/0.3
Ω_2, K_2	Coefficient of Circle_2 chaos	0.359/0.559
p_s	Probability of vigilance and random walk	[0.2, 0.4]
n_g	Size of hypergrid per dimension	[10, 30]
η	Grid expansion coefficient	[0.05, 0.15]
λ	Pareto solution stacking density threshold	1/3
ρ	Step size scaling factor	[0.1, 4]

4.4. MASCSO-HS's performance evaluation

Performance evaluation metrics such as Inverted Generational Distance (IGD) [28,31], Spacing (SP) [28,29], and Maximum Spread (MS) [29] were used as the assessment criteria. The IGD formulas for these evaluation metrics are defined as follows:

$$IGD = \frac{\sqrt{\sum_{i=1}^{n} d_i^2}}{n}, \tag{32}$$

where n represents the size of the true Pareto solution set, and d_i is the Euclidean distance between the ith solution in the true Pareto solution set and the nearest solution found by the algorithm. The Inverted Generational Distance (IGD) is thus calculated as the Euclidean distance between each solution in the true Pareto solution set and the obtained solution set. A smaller IGD value indicates that the obtained solution set is closer to the true one.

Note: Due to space limitations, the detailed formulations for assessing the uniformity (Eq. (33)) and coverage (Eq. (34)) of the Pareto solution set have been omitted from the main text. Further details can be requested from the corresponding authors.

$$SP = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(\bar{d} - d_i\right)^2}, \tag{33}$$

$$MS = \sqrt{\sum_{i=1}^{o}\max\left(d\left(a_i - b_i\right)\right)} \tag{34}$$

The detailed configuration information for the parameters related to the MASCSO-HS algorithm used in the experiments is presented in Table 2. The experimental setup includes Processor: 12th Gen Intel(R) Core(TM) i7-12700 at 2.10 GHz, Memory: 32.0 GB RAM; utilizing MATLAB R2023b on the Windows 10 operating system. Each algorithm was executed 20 times, and the average values, standard deviation (SD), and the worst and best values of each metric (IGD, SP, and MS) were subsequently calculated and recorded. Table 2 shows the control parameters of meta-heuristics derived from initial experience with the algorithm and optimized through testing experiments.

According to the configurations of related algorithms described in Ref. [32], a comparison was conducted with the latest and most classical multi-objective optimization algorithms: MOSFO, MOPSO, NSGA-II, MOGWO, and MOEA/D. The performance metrics of these algorithms on the CEC 2009 test benchmark set, along with the experimental data statistics of the MASCSO-HS algorithm in IGD results, are presented in Table 3 (The MS and SP results can be provided upon request, if necessary.).

From the comprehensive experimental data assessed, it can be concluded that MASCSO-HS performs the best among the comparison algorithms, demonstrating effective convergence and a faster solution speed than other algorithms.

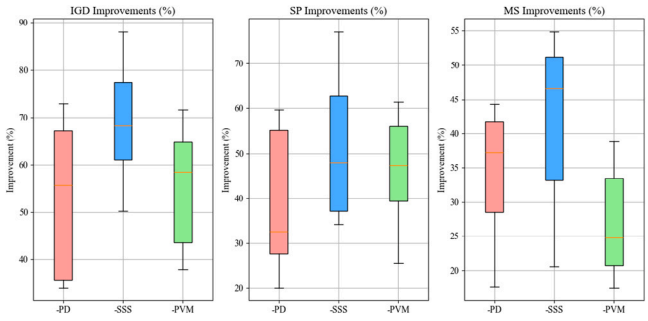

Fig. 8. The Improvements of each module in Ablation Studies on CEC2009 Test Functions.

4.5. Ablation study on the impact of optimization strategies in MASCSO-HS

We systematically remove one optimization strategy at a time: Population Diversity, Spiral Search Strategy, and Pigeon Vigilance Mechanism, denoting these modified versions as MASCSO-HS-PD, MASCSO-HS-SSS, and MASCSO-HS-PVM, respectively.

According to the tests, each optimization strategy module significantly contributes to the algorithm's performance. As shown in Fig. 8, average values after 20 runs for each test function reveal improvements in IGD, SP, and MS metrics, indicating the critical contribution of these modules to overall algorithm efficacy.

5. Multi-objective-aware container-based microservice deployment solution

The MSCSO-HS algorithm optimizes container-based microservice deployment with multiple objectives. It aims to identify the optimal deployment solution for container-based microservices in data centers, considering various optimization objectives and deployment constraints.

5.1. Definition of population encoding scheme

According to Eqs. (2) and (3), a container-based microservice deployment solution CSps $\in \mathbb{R}^{S\times M}$ can be represented as:

$$\left[\mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{X}_S\right]^T = \begin{bmatrix} f_{CSps}\left(ms_1\right) \\ f_{CSps}\left(ms_2\right) \\ \cdots \\ f_{CSps}\left(ms_S\right) \end{bmatrix}, \tag{35}$$

where $\mathrm{X}_i(1 \le i \le S)$ represents the deployment of ms_i on M server nodes, and the solution function $f_{CSps}\left(ms_i\right) = \left(x_{i1}, x_{i1}, \ldots, x_{iM}\right)$. According to Eqs. (2), (3), and (35), the deployment solution is an $S \times M$ matrix, and $1 \le \sum_{j=1}^{M} x_{ij}\left(x_{ij} \in \{0,1\}\right)$, indicating that each ms_i is deployed on at least one server node. The population individual X_{MS} in the MSCSO-HS algorithm is designed as follows:

$$\mathrm{X}_{MS} = \left[y_1, y_2, \ldots, y_S\right], \tag{36}$$

where $\mathrm{y}_i(1 \le i \le S) = \left[\mathrm{y}_{i1}, \mathrm{y}_{i2}, \ldots \mathrm{y}_{iM}\right], \mathrm{y}_{ij}(1 \le j \le M) = \{0,1\}$. First, each server node is assigned a number-position encoding; that is, for a server node with number i, if the corresponding ith position is 1, it indicates the deployment of the corresponding microservice; if it is 0, the microservice is not deployed. Thus, in the MASCSO-HS algorithm, the dimension of the population individual is $d = S$, and each dimension is an M-bit binary encoding.

Table 3
IGD results of algorithms in the CEC2009 test functions.

Algorithm	UF	Average	SD	Worst	Best	UF	Average	SD	Worst	Best
MSCSO-HS	1	**0.05713**	**0.00961**	**0.06935**	0.04188	2	**0.02990**	**0.00220**	**0.03310**	0.02531
MOSFO		0.06078	0.01479	0.09472	**0.03838**		0.03061	0.00796	0.04438	**0.01824**
MOGWO		0.11442	0.01954	0.15774	0.08023		0.05825	0.00739	0.07322	0.04980
MOPSO		0.13700	0.04407	0.22786	0.08990		0.06040	0.02762	0.13051	0.03699
NSGA-II		0.18640	0.01911	0.22631	0.15440		0.04492	0.00772	0.05922	0.03284
MOEA/D		0.18710	0.05070	0.24640	0.12650		0.12230	0.01070	0.14370	0.10490
MSCSO-HS	3	0.38587	0.07175	0.63280	0.27049	4	**0.04949**	0.00161	**0.05228**	0.04631
MOSFO		0.30180	0.08028	0.49793	0.16237		0.05265	0.00478	0.06029	**0.04478**
MOGWO		**0.25569**	0.08070	0.36786	**0.12950**		0.05867	**0.00048**	0.05936	0.05797
MOPSO		0.31399	0.04473	0.37773	0.25648		0.13504	0.00739	0.15189	0.12733
NSGA-II		0.27400	**0.03691**	**0.33351**	0.21761		0.09661	0.01073	0.11812	0.07612
MOEA/D		0.28865	0.01592	0.31294	0.26342		0.06810	0.00210	0.07040	0.06470
MSCSO-HS	5	**0.36522**	0.13793	**0.60985**	**0.17367**	6	**0.21702**	0.07534	**0.36620**	**0.11534**
MOSFO		0.81965	0.28927	1.56396	0.47421		0.25152	0.27570	0.90277	0.13518
MOGWO		0.79707	0.37857	1.73857	0.46795		0.27937	0.10448	0.55036	0.19338
MOPSO		2.20237	0.55304	3.03836	1.46479		0.64752	0.26612	1.24281	0.37933
NSGA-II		1.37961	0.22912	2.1275	1.19324		0.51132	0.13572	0.80123	0.28420
MOEA/D		1.29145	**0.13489**	1.46746	1.12306		0.68812	**0.05533**	0.74011	0.55235
MSCSO-HS	7	**0.04309**	**0.00978**	**0.06380**	0.02691	8	**0.21636**	**0.03531**	**0.31953**	**0.17225**
MOSFO		0.12464	0.01015	0.15974	**0.01903**		0.36527	0.10864	0.59228	0.24561
MOGWO		0.16036	0.13911	0.40142	0.06275		2.05777	1.14552	3.87888	0.46131
MOPSO		0.35395	0.20442	0.61512	0.05402		0.53671	0.18257	0.79637	0.24530
NSGA-II		0.24872	0.09733	0.47301	0.04832		1.47756	0.37454	2.51525	1.10274
MOEA/D		0.45520	0.18980	0.67700	0.02900		-	-	-	-
MSCSO-HS	9	**0.18300**	**0.03327**	**0.22633**	0.12977	10	**0.60551**	**0.17034**	**0.92489**	**0.29982**
MOSFO		0.18727	0.10286	0.35615	**0.11626**		0.89247	0.23098	1.96783	0.33503
MOGWO		0.19174	0.09250	0.44794	0.12910		3.59453	3.48829	12.9564	1.04314
MOPSO		0.48850	0.14449	0.72210	0.33355		1.63719	0.29879	2.16220	1.22008
NSGA-II		0.24162	0.15545	0.38176	0.16453		4.64931	1.10352	6.71283	2.56657
MOEA/D		-	-	-	-		-	-	-	-

5.2. The fitness function for the deployment model

According to Section 2, the MASCSO-HS algorithm deploys multiple container instances of microservices onto suitable server nodes to obtain an optimal or sub-optimal deployment solution. To choose the best deployment solution from the non-dominated Pareto front, we process each objective function using max–min normalization:

$$\mathcal{F}_{ob}^{\text{norm}}(x) = \frac{\mathcal{F}_{ob}(x) - \min_{x\in \mathrm{X}_{MS}} \mathcal{F}_{ob}(x)}{\max_{x\in \mathrm{X}_{MS}} \mathcal{F}_{ob}(x) - \min_{x\in \mathrm{X}_{MS}} \mathcal{F}_{ob}(x)}, \quad ob \in \{cde,\ \text{dei, ineff}\ \}, \tag{37}$$

where $x \in \mathrm{X}_{MS}$ is a deployment solution obtained by the algorithm. $\max_{x\in \mathrm{X}_{MS}} \mathcal{F}_{ob}(x)$ and $\min_{x\in \mathrm{X}_{MS}} \mathcal{F}_{ob}(x)$ represent the maximum and minimum values of the corresponding objective function in the obtained non-dominated Pareto front, respectively. By normalizing each objective function according to Eq. (37), we can uniformly map the impact of each objective function to the domain $[0, 1]$, ensuring fairness in their influence on the outcome.

5.3. MASCSO algorithm

MASCSO is a swarm intelligence-based multi-objective aware microservice deployment optimization algorithm. The algorithm is built upon the MSCSO-HS. The deployment model's fitness function selects the optimal container-based microservice deployment solution from the non-dominated Pareto solutions in the adaptive hypergrid. Algorithm 2 presents the pseudocode of the MASCSO algorithm.

Algorithm 2 MASCSO Algorithm

```
Input SF, MS, NS;
Initialize X_MS = [y_1, y_2, ..., y_S];
Initialize MASCSO − HS(X^0_MS);
while t < iter_Max do
  ResultSet^t = MASCSO − HS(X^t_MS);
  for each X_MS in ResultSet^t do
    for each ⊋ in {cde, dei, ineff} do
      F^norm_ob(x) = (F_ob(x) − min_{x∈X_MS} F_ob(x)) / (max_{x∈X_MS} F_ob(x) − min_{x∈X_MS} F_ob(x));
    end for
    if F_min(X_MS) < resulmin then
      resulmin = F_min(X_MS);
      resultMS = X_MS;
    end if
  end for
  t = t + 1;
end while
Return resultMS;
```

6. Experiments and discussions

We compare the MASCSO algorithm with other state-of-the-art and classical algorithms to assess its effectiveness in container-based microservice deployment. In the experiments, we use Alibaba Cloud's V2018 [33] dataset as a benchmark to generate microservice sets based on functional chains as experimental data.

6.1. Dataset and environment description

Based on the invocation dependency relationships among Job, Task, and Instance in the V2018 [33] dataset, we generate a dataset with 3 SFCs in the SF set, containing a total of 18 microservices and 22 invocation dependency edges. The corresponding SF sets are

$$\begin{aligned} SF_1 &= \{ms_1, ms_2, ms_3, ms_7, ms_{11}\}, \\ SF_2 &= \{ms_2, ms_3, ms_4, ms_5, ms_6, ms_7, ms_8, ms_9, ms_{11}\}, \\ SF_3 &= \{ms_2, ms_7, ms_{10}, ms_{12}, ms_{13}, ms_{14}, ms_{15}, ms_{16}, \\ &\quad ms_{17}, ms_{18}\} \end{aligned} \tag{38}$$

The specific data configurations for all MS sets are shown in Tables 4–5.

The corresponding microservice invocation dependency DAG is shown in Fig. 9 (see Table 6).

Table 4
The information of each $ms_i(i = 1, 2, \ldots, 18)$ in MS set.

MS set	ms1	ms2	ms3	ms4	ms5	ms6	ms7	ms8	ms9	ms10	ms11	ms12	ms13	ms14	ms15	ms16	ms17	ms18
r_k^{CPU}	3.8	1.5	4.2	3.1	5	3.2	5.6	50	4.5	3.2	3.1	6.3	7	6.5	18	20	6.3	6.2
r_k^{RAM}	1.5	13.6	5.7	6.5	7	5.1	1.6	55	4.3	2.5	4.3	4.5	6.8	7.5	50	50	3.5	2.1
v_k	30	55	45	28	60	30	100	10	45	55	60	70	35	35	20	20	70	20
v_k^T	10	11	9	7	10	6	10	2.5	9	5	10	7	6	5	2.5	6	8	7
$SiCon_i$	64.1	48.3	180.5	41.2	80.6	50	22.8	55.4	88.3	65	37.6	63.2	56.1	32.7	52.9	65.8	52.5	68.3

Table 5
The edge information of $ms_i(i = 1, 2, \ldots, 18)$ call dependency DAG graph.

$Edge$	W_{ii}	v_{ii}	$Dd_{ii'}$	$Edge$	W_{ii}	v_{ii}	$Dd_{ii'}$
$(0, ms_1)$	1	45	0	(ms_5, ms_6)	2	30	6.0
(ms_1, ms_2)	2	50	4.1	(ms_6, ms_9)	2	25	5.1
(ms_1, ms_7)	2	20	1.2	(ms_{10}, ms_2)	2	15	4.2
(ms_2, ms_3)	1	10	5.0	$(0, ms_{10})$	1	25	0
(ms_2, ms_{11})	1	6	4.2	(ms_{10}, ms_{12})	2	55	4.6
(ms_2, ms_7)	1	13	2.3	(ms_{12}, ms_{13})	2	35	2.3
$(0, ms_3)$	1	30	0	(ms_{12}, ms_{14})	2	20	4.3
(ms_3, ms_4)	2	80	6.7	(ms_{14}, ms_{15})	2	30	2.4
(ms_3, ms_{11})	1	12	6.0	(ms_{15}, ms_{16})	3	40	82
(ms_4, ms_5)	2	90	8.0	(ms_{13}, ms_{17})	2	35	2.5
(ms_5, ms_8)	3	50	80	(ms_{17}, ms_{18})	2	45	5.3

Table 6
Configuration description of physical server nodes NS_p.

Parameter	Description	Values
M	Number of Server Nodes	120/240
Rs_p^{CPU}	Range of CPU Resource Capacity	{200.0, 400.0, 800.0}
Rs_p^{RAM}	Range of RAM Resource Capacity	{200.0, 400.0, 800.0}
Ds_{ij}	Inter-node Network Distance	{1.0, 4.0, 8.0}
ρ_{NS_p}	Node Failure Rate	(0.1%, 3.0%)

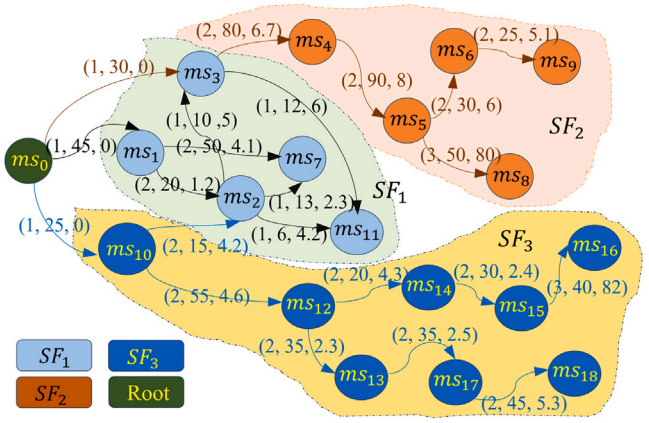

Fig. 9. Microservice invocation dependency DAG.

In Table 5, $\mathrm{Ds}_{ij} = \left\|\mathrm{Pos}_i - \mathrm{Pos}_j\right\| \in \{1.0, 4.0, 8.0\}$. When $\left\|\mathrm{Pos}_i - \mathrm{Pos}_j\right\| = 0$, i.e., $i = j$, it represents the same node and is set to 1.0. When $i \neq j$ but $\left\|\mathrm{Pos}_i - \mathrm{Pos}_j\right\| \leq 4$, it indicates that the two different nodes are in the same CDC, and the value is set to 4.0. When $4.0 < \left\|\mathrm{Pos}_i - \mathrm{Pos}_j\right\|$, it means that the two nodes are in different CDCs, and the value is set to 8.0. Table 2 shows the experimental environment and the configuration of the MASCSO (MASCSO-HS) algorithm.

Experimental description: To test the effectiveness of MASCSO under different server cluster scales and user request pressures, the experiments simulate multi-objective aware microservice deployment under two server cluster scales, as shown in Table 6, M = 120 and M = 240, with 6 types of user request numbers $Ur_{ms_k} = \{\times 1.0, \times 2.0, \times 3.0, \times 4.0, \times 5.0, \times 6.0\}$, i.e., Experiment 1(M = 120) and Experiment 2($M = 240$). In the multi-objective awareness, the weight coefficients of the objective functions are set as follows:

$$\{w_{cde}, w_{\mathrm{dei}}, w_{\mathrm{ineff}}\} = \left\{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right\}.$$

6.2. The deployment solution

Microservice deployment can be categorized into static and dynamic scenarios. Static deployment involves predefining the deployment strategy before system startup, making it suitable for stable load conditions. In contrast, dynamic deployment adjusts the deployment strategy based on real-time operational states, which is ideal for fluctuating loads. The deployment solution primarily focuses on quasi-dynamic microservice deployment, aiming to optimize the initial deployment strategy to accommodate moderate load variations and node failures while ensuring service quality and enhancing resource utilization.

Each normalized objective function is assigned a weight coefficient $w_{ob}(ob \in \{cde, \mathrm{dei}, \mathrm{ineff}\})$ to represent the importance of the corresponding objective function in the overall objective. Combining Eqs. (2), (36), and (37), the deployment model for solving the deployment solution can be defined as:

$$\mathrm{Find}\ (\mathrm{X}_{MS}) = [y_1, y_2, \ldots, y_S] \tag{39}$$

Subject to:

$$\begin{aligned} \mathcal{F}_{\min}(X) &= \min_{ob \in \{cde,\ \mathrm{dei,ineff}\}} \sum w_{ob} \cdot \mathcal{F}_{ob}^{\mathrm{norm}}(X), \qquad w_{ob} \geq 0, \\ &\sum_{ob \in \{cde,\ \mathrm{dei,ineff}\}} w_{ob} = 1. \end{aligned} \tag{39-1}$$

The experimental configuration validates the practical applicability of the MASCSO algorithm. All simulations focus on the configurations indicated by the Pareto front to ascertain their deployability in real cluster environments. We utilized the Alibaba Cloud V2018 dataset to generate microservice sets, with the deployment scheme based on the invocation dependencies among Job, Task, and Instance. Performance parameters for multiple server nodes were defined, including CPU and RAM capacities, inter-node network distances, and node failure rates, all adhering to current cloud computing standards.

The configurations generated by the MASCSO algorithm meet the needs of various server cluster scales (*e.g.*, $M = 120$ and $M = 240$). By optimizing the fitness functions of the deployment model, we ensure that the microservice solutions satisfy multiple optimization objectives and are feasible for implementation in real cloud environments. This validates that our optimization method provides actionable solutions for real-world microservice architectures, optimizing resource utilization and achieving efficient management.

6.3. Experimental results data analysis and comparison

Using the aforementioned test dataset and simulation experiment settings, 20 experiments are conducted for each scenario using the algorithms MASCSO, AF-CSDS [19], APSO-TSDS [18], MSG-NSGA-III [34], ACO-MCMS [17], and GA-NSGA-II [35]. The obtained experimental result data are statistically analyzed and compared based on 5 evaluation indicators: communication delay $\mathcal{F}_{\mathrm{cde}}(X)$, container deployment density $\mathcal{F}_{\mathrm{dei}}(X)$, microservice failure rate $\mathcal{F}_{\mathrm{ineff}}(X)$, cluster resource utilization $\mathcal{F}_{UCPU}(X)$, and $\mathcal{F}_{\mathrm{URAM}}(X)$.

Table 7
$\mathcal{F}_{cde}(X)$ in Experiment 1 (M = 120).

Algorithm	X1.0	X2.0	X3.0	X4.0	X5.0	X6.0
MASCSO	**38.2648**	**84.5876**	**129.1326**	**178.2415**	**229.8645**	**286.1429**
AF-CSDS	41.24635	91.6312	142.2383	197.2654	**257.5917**	308.7243
APSO-TSDS	43.6725	103.0629	168.4009	229.8420	302.4544	357.3990
MSG-NSGA-III	63.9962	126.8161	189.5266	251.5483	317.1417	387.6211
ACO-MCMS	48.7573	119.4410	184.3963	249.8399	320.9509	376.8277
GA-NSGA-II	50.9100	119.0349	185.6425	260.1502	315.7780	378.1257

Table 8
$\mathcal{F}_{cde}(X)$ in experiment 2 (M = 240).

Algorithm	X1.0	X2.0	X3.0	X4.0	X5.0	X6.0
MASCSO	39.9094	81.3603	129.3527	180.8494	226.8018	284.9376
AF-CSDS	44.4600	93.0183	144.3533	195.1785	256.7523	309.3256
APSO-TSDS	48.3544	104.4719	171.53029	242.2687	302.5902	359.3504
MSG-NSGA-III	70.1917	122.3831	194.9334	250.7215	322.8488	387.5925
ACO-MCMS	49.2449	122.1727	185.3435	252.6451	313.3203	384.0997
GA-NSGA-II	57.7841	124.7605	185.8094	259.6519	319.4655	388.6816

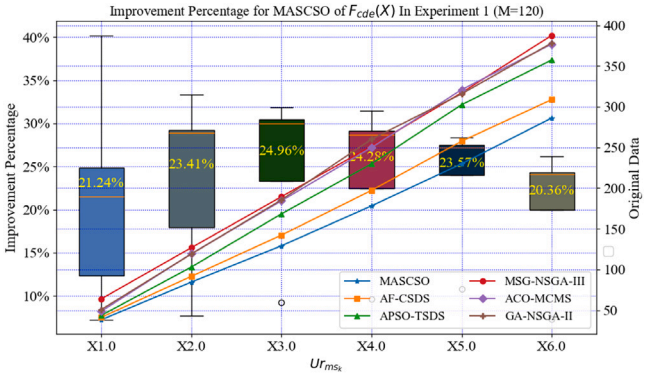

Fig. 10. $\mathcal{F}_{cde}(X)$ Results and improvement by MASCSO (M = 120).

6.3.1. Microservice communication delay results $\mathcal{F}_{cde}(X)$

Based on the deployment schemes obtained by the algorithm search, the results of Experiment 1 (M = 120) and Experiment 2(M = 120) are collected according to the microservice communication delay evaluation function $\mathcal{F}_{\text{cde}}(X)$, as shown in Table 7 and Table 8, respectively. The tables' data with a gray background and bold font represent the best results among the 6 comparison algorithms.

Lower values of $\mathcal{F}_{cde}(X)$ signify enhanced communication delay performance among deployed microservices. The data in the table demonstrate that MASCSO consistently achieves the lowest communication delays across all scenarios, outperforming the other six algorithms. Additionally, further analysis confirms that MASCSO excels in communication delay performance in both experiments, as illustrated in Figs. 10 and 11.

Figs. 10 and 11 show that MASCSO achieves an overall improvement of more than 23% compared to the other five algorithms. The data from Tables 7 and 8 indicate that MASCSO reaches its best case in Experiment 1 ($M = 120, Ur_{ms_k} = \times 1.0$) and Experiment 2 ($M = 240, Ur_{ms_k} = \times 1.0$), outperforming MSG-NSGA-III by 40.2% and 43.1%, respectively. AF-CSDS ranks as the second-best algorithm, but MASCSO exceeds it by more than 7% in every scenario and pushes the improvement to over 12.5% in the highest case.

6.3.2. Container deployment density results $\mathcal{F}_{dei}(X)$

Optimizing microservice resource demands can increase container deployment density and enhance resource utilization. The statistical results of container deployment density for each algorithm's deployment solutions are presented in Figs. 12 and 13, and Table 9.

In Figs. 12–13, the line graphs show container deployment density results obtained by six algorithms from two experiments. The box plots illustrate MASCSO's percentage improvement over the other five algorithms across varying request counts. Fig. 14 shows the distribution of container density results achieved by each algorithm in the two experiments.

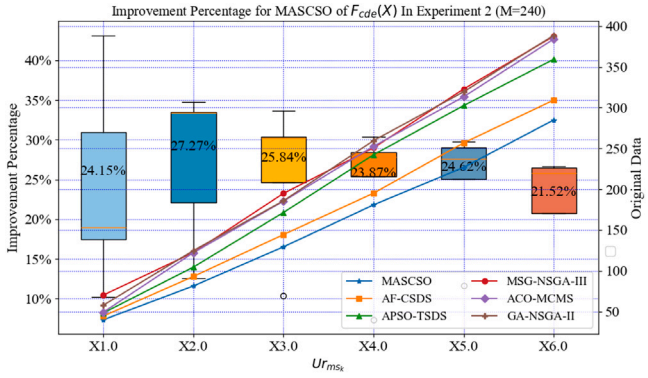

Fig. 11. $\mathcal{F}_{cde}(X)$ Results and improvement by MASCSO (M = 240).

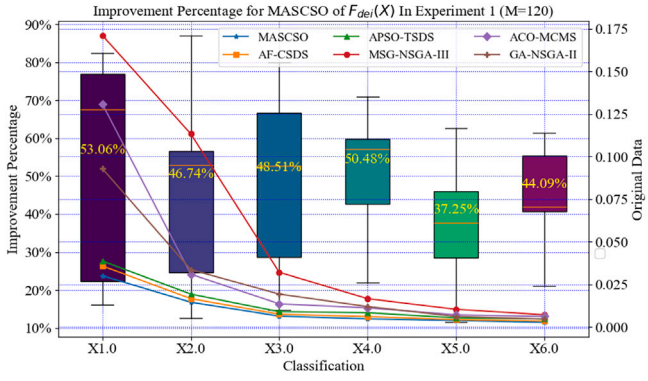

Fig. 12. $\mathcal{F}_{dei}(X)$ Results and improvement by MASCSO (M = 120).

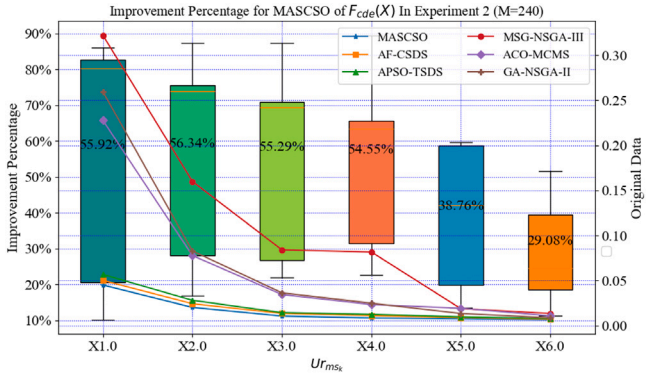

Fig. 13. $\mathcal{F}_{dei}(X)$ Results and improvement by MASCSO (M = 240).

Table 9
The range of values of $\mathcal{F}_{\text{dei}}(X)$.

Algorithm	Container deployment density range	
	Experiment1 (M = 120)	Experiment2 (M = 240)
MASCSO	**0.002805 ~ 0.030126**	**0.006531 0.045061**
AF-CSDS	0.003552 ~ 0.035881	0.007365 ~ 0.050140
APSO-TSDS	0.004839 ~ 0.038777	0.008016 ~ 0.056780
MSG-NSGA-III	0.007265 ~ 0.171087	0.013491 ~ 0.321911
ACO-MCMS	0.006286 ~ 0.130777	0.010780 ~ 0.227637
GA-NSGA-II	0.004724 ~ 0.092964	0.008647 ~ 0.259413

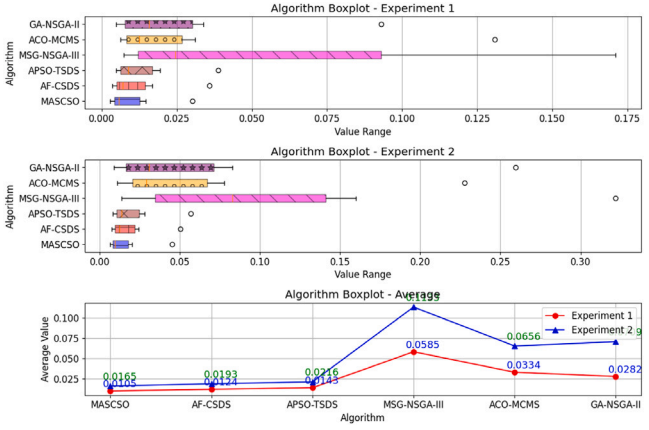

Fig. 14. Data distribution and result range of container deployment density for each algorithm.

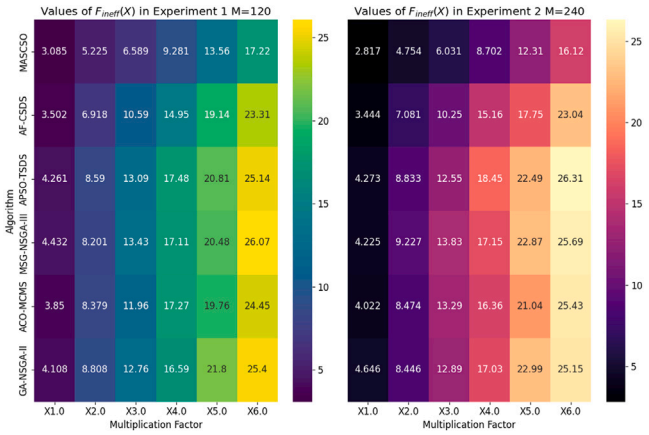

Fig. 15. Heatmap of microservice failure rate results for each algorithm.

Observations from these figures indicate that MASCSO exhibits a significant overall improvement in Experiment 2. With user requests $ureqs = \{\times 1.0, \times 2.0, \times 3.0, \times 4.0\}$, MASCSO achieves an average improvement of 50% over the other algorithms. Statistical analysis reveals that MA- SCSO attains the highest container deployment density among the six algorithms. The proposed algorithm achieves an average overall improvement of 75% compared to MSG-NSGA-III across both experiments and a 50% improvement over ACO-MCMS. Even when compared to the relatively better-performing AF-CSDS, MASCSO surpasses it by an average of more than 16%. MASCSO demonstrates a 47.51% improvement compared to all other algorithms.

6.3.3. *Microservice failure rate results* $\mathcal{F}_{ineff}(X)$

Different deployment solutions result in varying microservice invocation failure rates, which affect SFCs' performance, including service waiting time, response time, and resource utilization. Based on the calculation Eq. (10), the resulting data for the microservice failure rate $\mathcal{F}_{\text{ineff}}(X)$ of each deployment solution is presented in the heatmap of Fig. 15.

The smaller the $\mathcal{F}_{\text{ineff}}(X)$ result, the better. In Fig. 15, darker colors indicate lower microservice failure rates.

Upon observing Fig. 15, it is clear that MASCSO's results have the darkest color within each column, indicating that the $\mathcal{F}_{\text{ineff}}(X)$ values from MASCSO's solutions are the smallest and best among the six algorithms.

As shown in Fig. 16, MASCSO's overall average improvement in $\mathcal{F}_{\text{ineff}}(X)$ exceeds 38% compared to the other algorithms. Specifically,

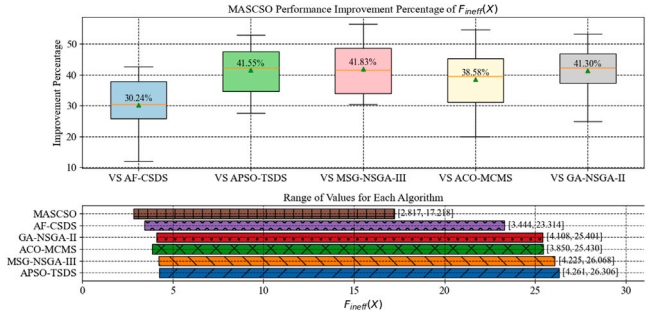

Fig. 16. MASCSO's improvement in $\mathcal{F}_{\text{ineff}}(X)$ and result ranges of each algorithm.

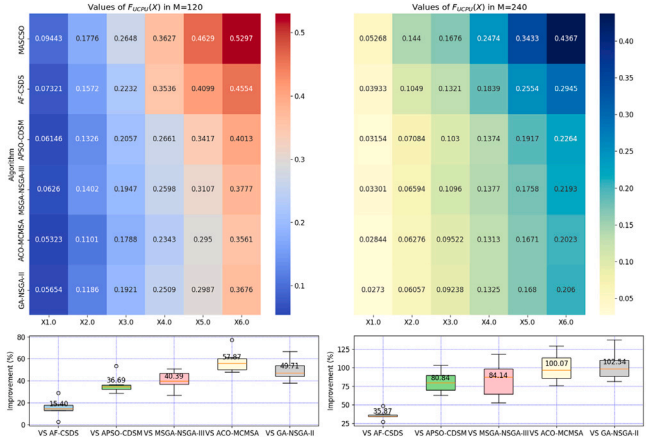

Fig. 17. Heatmap of $\mathcal{F}_{UCPU}(X)$ and improvement by MASCSO.

it achieves a 30.24% improvement over AF-CSDS and a maximum of 41.83% over MSG-NSGA-III. The distribution of the percentage improvement data indicates that the minimum improvement over AF-CSDS is still over 10%. The range of $\mathcal{F}_{\text{ineff}}(X)$ values obtained by each algorithm reveals that MASCSO's $\mathcal{F}_{\text{ineff}}(X) \in [2.817, 17.218]$.

6.3.4. *Cluster resource utilization rates* $\mathcal{F}_{UCPU}(X)$ *and* $\mathcal{F}_{URAM}(X)$

Different microservice deployment schemes have varying impacts on resource utilization rates. When deploying microservices, co-locating those with non-conflicting resource demands on the same server can improve resource utilization and system responsiveness. The goal of increasing container deployment density in the deployment process inherently involves this complementary deployment heuristic rule. By using Eqs. (7)–(8), $\mathcal{F}_{UCPU}(X)$ and $\mathcal{F}_{URAM}(X)$ are calculated to determine the CPU and RAM utilization rates in the cluster, enabling a comparison of algorithm performance in terms of resource utilization. Figs. 17 and 18 display CPU and RAM utilization heatmaps for all algorithms in various scenarios, along with box plots showing MASCSO's percentage improvement compared to other algorithms.

Higher resource utilization rates are desirable. As observed in Figs. 17–18, the microservice deployment solutions derived from the MASCSO algorithm exhibit the highest color values in the CPU and RAM utilization rate heatmaps, indicating superior utilization rates.

The CPU and RAM improvement percentage plots show that MASCSO achieves the maximum improvement over ACO-MCMSA. Specifically, MASCSO improves CPU utilization by 57.97%(M = 120) and 100.07%(M = 240), and RAM utilization by 71.23%(M = 120) and 96.56%(M = 240). The improvement over AF-CSDS is relatively smaller, with CPU utilization improvements of 15.40%(M = 120) and 35.87%(M = 240), and RAM utilization improvements of 19.23%(M = 120) and 21.94%(M = 240). To summarize the results, the CPU and RAM utilization rates obtained across all scenarios are presented in

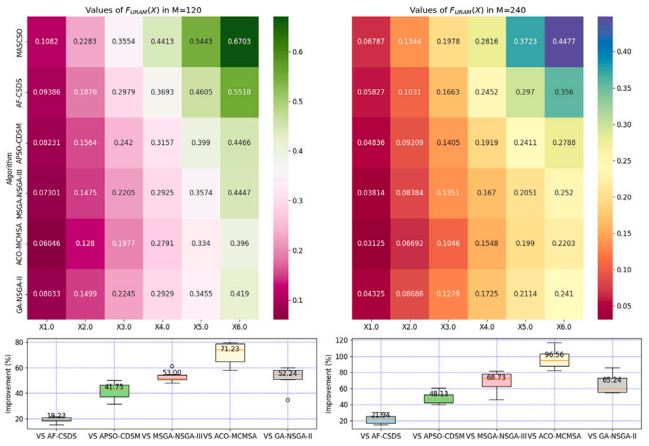

Fig. 18. Heatmap of $\mathcal{F}_{URAM}(X)$ and improvement by MASCSO.

Table 10
Resource (CPU/RAM) utilization results for all algorithms.

Algorithms	$M = 120$		$M = 240$	
	CPU	RAM	CPU	RAM
MASCSO	31.54%	39.13%	23.20%	25.03%
AF-CSDS	27.88%	32.68%	16.84%	20.43%
APSO-CDSM	23.48%	27.37%	12.68%	16.55%
MSGA-NSGAIII	22.43%	25.59%	12.36%	14.69%
ACO-MCMSA	20.46%	23.26%	11.45%	12.95%
GA-NSGAII	21.41%	25.20%	11.44%	14.71%

Table 11
Resource utilization results with difference w_{ob}.

w_{ob}	$M = 120$		$M = 240$	
	CPU	RAM	CPU	RAM
{0.4, 0.4, 0.2}	29.36%	40.62%	22.48%	25.97%
{0.5, 0.2, 0.3}	27.61%	41.23%	20.82%	26.71%
{1/3, 1/3, 1/3}	31.54%	39.13%	23.20%	25.03%
{0.2, 0.5, 0.3}	33.42%	33.84%	24.13%	22.54%

Table 10. The table displays average CPU and RAM utilization rates for each algorithm's deployment solutions under two different cluster configurations.

As shown in Table 10, the MASCSO algorithm achieves the highest CPU and RAM utilization rates among all the algorithms. Specifically, the average CPU and RAM utilization rates reach 31.54% and 39.13%(M = 120), and 23.20% and 25.03%, respectively.

The MASCSO algorithm outperforms all six algorithms across microservice communication latency, container deployment density, microservice failure rate, and cluster resource utilization metrics. The proposed multi-objective-aware container-based microservice deployment algorithm has been validated for its effectiveness and superiority.

6.3.5. Discussion on alternative multi-objective optimization techniques

To investigate the impact of the proposed MASCSO algorithm on multi-objective optimization in container-based microservices, we compared it with the Differential Group-Based Whale Optimization Algorithm (DGWO). The resource utilization results (CPU/RAM) for various weight configurations are summarized in Table 11.

The results show that varying the weights significantly affects resource utilization. For example, in the $M = 120$ scenario, the highest CPU utilization (33.42%) was achieved with the weight configuration $\{0.2, 0.5, 0.3\}$, favoring density and failure rate over latency. Conversely, in $M = 240$, the configuration $\{0.4, 0.4, 0.2\}$ yielded the lowest CPU utilization (22.48%), highlighting the importance of optimal weight assignment in achieving desired performance metrics.

In addition to MASCSO, alternative techniques such as the Weighted Fitness Function method (*e.g.*, DGWO) and distributed approaches (*e.g.*, DiCSPM) are noteworthy. DGWO dynamically adjusts weights based on current performance metrics, enhancing resource allocation under varying workloads. Meanwhile, DiCSPM allows for decentralized task distribution, improving scalability and resilience in cloud environments.

7. Conclusions and future work

Container-based microservices architecture is a major paradigm for modern application development to form microservices within the same SFC. To tackle challenges in deployment, we propose the MASCSO algorithm, which optimizes for multiple objectives. Existing models often inadequately address cloud complexities, focusing on isolated goals like resource utilization and performance, while neglecting interdependencies and communication delays among microservices. This leads to inefficiencies under fluctuating workloads. Our proposed Containerized Microservice Deployment Model offers a robust solution to improve resource scheduling and has shown superior performance in 9 out of 10 CEC 2009 benchmark tests. Additionally, the MASCSO algorithm promotes efficient resource management in real-world applications such as e-commerce, healthcare, and IoT ecosystems, enhancing user experience and operational reliability. This research can significantly influence the deployment and management of microservices in increasingly cloud-based environments.

Overall, this paper presents a robust model targeting quasi-dynamic microservice deployment scenarios. Key contributions include: (1) a novel formal model that analyzes microservice interdependencies within Service Function Chains, (2) the development of the Multi-objective Sand Cat Swarm Optimization with Hybrid Strategies (MA SCSO-HS) algorithm, and (3) extensive validation of the MASCSO algorithm, demonstrating significant improvements in resource utilization, system responsiveness, and application reliability across diverse environments. By optimizing deployment strategies, our model achieves enhancements of 23.76% in communication latency, 47.51% in deployment density, 38.70% in failure rate, 58.50% in CPU utilization, and 53.81% in RAM utilization. These contributions not only advance theoretical understanding but also provide actionable solutions for practical applications in cloud environments.

However, it is important to acknowledge some limitations of the MASCSO algorithm. While it performs well in optimizing microservice deployment, its scalability in handling very large or rapidly changing service chains may present challenges. Additionally, the algorithm's effectiveness can be influenced by specific application constraints, such as varying resource availability and the nature of interdependencies among microservices. Future work should focus on addressing these limitations to enhance the algorithm's robustness in diverse operational scenarios.

In future research, we aim to apply the MSCSO-HS algorithm to address the large-scale multi-objective cost-aware microservice optimization scheduling problem in heterogeneous cloud data centers. We will specifically focus on: (1) enhancing the scalability of MASCSO for dynamic environments by integrating adaptive resource allocation techniques, (2) developing a framework to model and mitigate resource availability variability, and (3) implementing case studies in real-world applications to validate the algorithm's effectiveness under varying interdependency conditions. By pursuing these goals, we hope to further refine our model and improve its applicability to diverse microservices architectures.

CRediT authorship contribution statement

Jiaxian Zhu: Writing – original draft, Methodology. **Weihua Bai:** Methodology, Conceptualization. **Huibing Zhang:** Investigation. **Weiwei Lin:** Validation. **Teng Zhou:** Writing – review & editing, Project administration. **Keqin Li:** Supervision.

Declaration of competing interest

The authors declare that they have no conflict of interest.

Acknowledgments

The research was supported by National Natural Science Foundation of China (No. 62267003, No. 62462021), Philosophy and Social Sciences Planning Project of Zhejiang Province (No. 25JCXK006YB), Hainan Provincial Natural Science Foundation (No. 625RC716), Guangdong Basic and Applied Basic Research Foundation (No. 2025A1515010197, No. 2025A1515010113), The Innovative Development Joint Fund of Natural Science foundation in Shandong Province (No. ZR2024LZH012), Special Fund for Guangdong Province University Key Field (No. 2023ZDZX 3041), Innovation Research Team Project of Zhaoqing University, Innovation Project of Guangdong Province University (No. 2024K QNCX023), and Innovation Project of Zhaoqing City (No. 24121215 4168613).

Data availability

Data will be made available on request.

References

[1] S. Mendes, J. Simão, L. Veiga, Oversubscribing micro-clouds with energy-aware containers scheduling, in: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, 2019, pp. 130–137.
[2] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, P. Hu, Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster, in: 2017 IEEE 36th International Performance Computing and Communications Conference, IPCCC, IEEE, 2017, pp. 1–8.
[3] I. Ahmad, M.G. AlFailakawi, A. AlMutawa, L. Alsalman, Container scheduling techniques: A survey and assessment, J. King Saud Univ- Comput. Inf. Sci. 34 (7) (2022) 3934–3947.
[4] L. Lv, Y. Zhang, Y. Li, K. Xu, D. Wang, W. Wang, M. Li, X. Cao, Q. Liang, Communication-aware container placement and reassignment in large-scale internet data centers, IEEE J. Sel. Areas Commun. 37 (3) (2019) 540–555.
[5] W. Bai, J. Zhu, S. Huang, H. Zhang, A queue waiting cost-aware control model for large scale heterogeneous cloud datacenter, IEEE Trans. Cloud Comput. 10 (2) (2020) 849–862.
[6] S. Pallewatta, V. Kostakos, R. Buyya, Placement of microservices-based iot applications in fog computing: A taxonomy and future directions, ACM Comput. Surv. 55 (14s) (2023) 1–43.
[7] S. Pallewatta, V. Kostakos, R. Buyya, QoS-aware placement of microservices-based IoT applications in Fog computing environments, Future Gener. Comput. Syst. 131 (2022) 121–136.
[8] S. Pallewatta, V. Kostakos, R. Buyya, MicroFog: A framework for scalable placement of microservices-based IoT applications in federated Fog environments, J. Syst. Softw. 209 (2024) 111910.
[9] F. Faticanti, M. Savi, F. De Pellegrini, D. Siracusa, Locality-aware deployment of application microservices for multi-domain fog computing, Comput. Commun. 203 (2023) 180–191.
[10] M. Imdoukh, I. Ahmad, M. Alfailakawi, Optimizing scheduling decisions of container management tool using many-objective genetic algorithm, Concurr. Comput.: Pr. Exp. 32 (5) (2020) e5536.
[11] O. Oleghe, Container placement and migration in edge computing: Concept and scheduling models, IEEE Access 9 (2021) 68028–68043.
[12] H. Li, Q. Zhang, Multiobjective optimization problems with complicated Pareto sets, MOEA/D and NSGA-II, IEEE Trans. Evol. Comput. 13 (2) (2008) 284–302.
[13] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.
[14] Q. Zhang, H. Li, MOEA/D: A multiobjective evolutionary algorithm based on decomposition, IEEE Trans. Evol. Comput. 11 (6) (2007) 712–731.
[15] Z. Lin, D. Wang, C. Cao, H. Xie, T. Zhou, C. Cao, GSA-KAN: A hybrid model for short-term traffic forecasting, Mathematics 13 (7) (2025) 1158.
[16] L. Bianchi, M. Dorigo, L.M. Gambardella, W.J. Gutjahr, A survey on metaheuristics for stochastic combinatorial optimization, Nat. Comput. 8 (2009) 239–287.
[17] M. Lin, J. Xi, W. Bai, J. Wu, Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud, IEEE Access 7 (2019) 83088–83100.
[18] M. Ouyang, J. Xi, W. Bai, K. Li, Band-area resource management platform and accelerated particle swarm optimization algorithm for container deployment in internet-of-things cloud, IEEE Access 10 (2022) 86844–86863.
[19] M. Ouyang, J. Xi, W. Bai, K. Li, A container deployment strategy for server clusters with different resource types, Concurr. Comput.: Pr. Exp. 35 (10) (2023) e7665.
[20] D. Liu, A. Hafid, L. Khoukhi, Workload balancing in mobile edge computing for internet of things: A population game approach, IEEE Trans. Netw. Sci. Eng. 9 (3) (2022) 1726–1739.
[21] H. Zhao, C. Zhang, An ant colony optimization algorithm with evolutionary experience-guided pheromone updating strategies for multi-objective optimization, Expert Syst. Appl. 201 (2022) 117151.
[22] K. Dubey, S.C. Sharma, A novel multi-objective CR-PSO task scheduling algorithm with deadline constraint in cloud computing, Sustain. Comput.: Inform. Syst. 32 (2021) 100605.
[23] S. Muniswamy, R. Vignesh, DSTS: A hybrid optimal and deep learning for dynamic scalable task scheduling on container cloud environment, J. Cloud Comput. 11 (1) (2022) 33.
[24] O.T.S.T. Deep, Long-term container allocation via optimized task scheduling through deep learning (OTS-DL) and high-level security, KSII Trans. Internet Inf. Syst. 17 (4) (2023) 1258–1275.
[25] T. Danino, Y. Ben-Shimol, S. Greenberg, Container allocation in cloud environment using multi-agent deep reinforcement learning, Electronics 12 (12) (2023) 2614.
[26] Y. Cheng, Z. Cao, X. Zhang, Q. Cao, D. Zhang, Multi objective dynamic task scheduling optimization algorithm based on deep reinforcement learning, J. Supercomput. 80 (5) (2024) 6917–6945.
[27] A. Seyyedabbasi, F. Kiani, Sand Cat swarm optimization: A nature-inspired algorithm to solve global optimization problems, Eng. Comput. 39 (4) (2023) 2627–2651.
[28] S. Mirjalili, S. Saremi, S.M. Mirjalili, L.d.S. Coelho, Multi-objective grey wolf optimizer: a novel algorithm for multi-criterion optimization, Expert Syst. Appl. 47 (2016) 106–119.
[29] C.A.C. Coello, G.T. Pulido, M.S. Lechuga, Handling multiple objectives with particle swarm optimization, IEEE Trans. Evol. Comput. 8 (3) (2004) 256–279.
[30] Q. Zhang, A. Zhou, S. Zhao, P.N. Suganthan, W. Liu, S. Tiwari, et al., Multiobjective Optimization Test Instances for the CEC 2009 Special Session and Competition, Technical Report 264, 2008, pp. 1–30.
[31] M.R. Sierra, C.A. Coello Coello, Improving PSO-based multi-objective optimization using crowding, mutation and ∈-dominance, in: International Conference on Evolutionary Multi-Criterion Optimization, Springer, 2005, pp. 505–519.
[32] J.L.J. Pereira, G.F. Gomes, Multi-objective sunflower optimization: A new hypercubic meta-heuristic for constrained engineering problems, Expert Syst. 40 (8) (2023) e13331.
[33] A. Corp, Alibaba cluster trace V2018, 2021, Online, URL https://github.com/alibaba/clusterdata.
[34] W. Ma, R. Wang, Y. Gu, Q. Meng, H. Huang, S. Deng, Y. Wu, Multi-objective microservice deployment optimization via a knowledge-driven evolutionary algorithm, Complex Intell. Syst. 7 (2021) 1153–1171.
[35] C. Guerrero, I. Lera, C. Juiz, Genetic algorithm for multi-objective optimization of container allocation in cloud architecture, J. Grid Comput. 16 (2018) 113–135.