Universal Library of Innovative Research and Studies Research Article

ISSN: 3065-0003 | Volume 2, Issue 2

Open Access | PP: 22-27

DOI: https://doi.org/10.70315/uloap.ulirs.2025.0202003

Methods for Minimizing Request Processing Latency in Microservices Architecture

Artem Iurchenko

Senior Software Engineer at Dexian, Atlanta, USA.

Abstract

The article examines the issue of reducing request processing latency in a microservices architecture deployed in a cloud environment. Key factors contributing to increased latency are identified, including frequent inter-service communication, uneven distribution of computational resources, and the lack of a comprehensive fault-tolerance strategy. A scheduling strategy based on a modified particle swarm optimization (PSO) algorithm and an extended Round Robin (RR) algorithm is proposed. The modified PSO accounts for the microservices call graph and the physical proximity of nodes within predefined threshold constraints, while the RR algorithm ensures balanced load distribution and eliminates single points of failure. The effectiveness of the approach was evaluated using datasets (traces) from Alibaba and Google, reflecting real-world microservices operation scenarios. Experimental results demonstrated a reduction in network traffic (up to 35%), a decrease in latency (by 80% or more in static scenarios), and a more uniform resource utilization (reducing standard deviation by 40–50%). To apply the described methodology (PSO+RR), it should be integrated with an orchestration system (Kubernetes), achieving a systematic improvement in both performance and fault tolerance. The findings presented in this article are intended for system architects, developers, and DevOps engineers seeking to optimize microservices system performance by minimizing request processing latency. The results can be incorporated into existing DevOps practices and applied in large data centers.

Keywords: *Microservices, Cloud Computing, Latency, Particle Swarm Optimization, Scheduling, Load Balancing, Fault Tolerance, Kubernetes.*

INTRODUCTION

The development of cloud computing and microservices architecture has fundamentally transformed the approach to designing and operating high-load applications. Microservices, where each service is represented as a small, loosely coupled container, have simplified the scaling of system components, enabled rapid modifications, and ensured high availability. Cloud service providers such as Google Cloud, Amazon ECS, and IBM Cloud offer infrastructure that supports the continuous operation of numerous microservices [1, 4].

However, microservices architecture also presents significant challenges. First, frequent network interactions between microservices increase the risk of network congestion and delays in request processing. Second, monitoring and resource allocation become more complex due to dynamic container migration and scaling, as well as fluctuating Central Processing Unit (CPU) and memory demands over time [1, 4]. These factors necessitate the development of specialized scheduling and load-balancing strategies to reduce latency while maintaining high service availability.

Several studies have focused on reducing network overhead to improve microservices distribution. For example, Alibaba's experience, described in an online resource [1], demonstrates how system scalability and fault tolerance can be enhanced by decomposing monolithic applications into independent services, allowing efficient management of high workloads. Similarly, Balalaie A., Heydarnoori A., and Jamshidi P. [2] highlight the importance of integrating DevOps methodologies for successful cloud migration, ensuring flexibility and rapid adaptation to changing operational conditions. A comprehensive review of principles, patterns, and migration challenges presented by Velepucha V. and Flores P. [5] further systematizes knowledge in application decomposition, identifying key challenges in transitioning from monolithic to microservices-based systems.

Alelyani A., Datta A., and Hassan G. M. [3] propose a dynamic microservices distribution strategy aimed at improving fault tolerance and reducing network traffic, which in turn minimizes request processing delays. Meanwhile, Li X. et al. [7] introduce a topology-aware framework that considers

Citation: Artem Iurchenko, "Methods for Minimizing Request Processing Latency in Microservices Architecture", Universal Library of Innovative Research and Studies, 2025; 2(2): 22-27. DOI: https://doi.org/10.70315/uloap.ulirs.2025.0202003.

service distribution within cloud infrastructure to optimize inter-service communication. The practical implementation of these methods is confirmed by the use of tools such as Kubernetes Scheduler [6] for computational task scheduling and Cloud Trace [4] for latency monitoring, enabling detailed analysis and real-time system management.

The study by Alelyani A., Datta A., and Hassan G. M. [8] focuses on optimizing cloud computing performance through microservices scheduling strategies. The proposed approach is based on analyzing service characteristics and utilizing intelligent scheduling algorithms to respond to changing cloud workload conditions.

Despite these advancements, a research gap remains in the existing literature: modern studies either concentrate on macro-level architectural changes or rely on static resource allocation methods without sufficiently integrating dynamic monitoring systems with adaptive real-time scheduling algorithms. This limitation reduces the effectiveness of developed methods under rapidly changing load and network conditions, which are characteristic of modern cloud infrastructures.

The objective of this study is to develop and justify a new methodology for dynamic adaptive scheduling aimed at minimizing request processing latency in microservices architecture within cloud systems.

The scientific novelty lies in integrating topology-aware computational task distribution with real-time monitoring systems, ensuring optimal resource utilization and improving system fault tolerance.

The research hypothesis suggests that a combined approach, incorporating dynamic network state evaluation and adaptive task distribution, significantly reduces latency compared to traditional static scheduling methods.

To test this hypothesis and achieve the research objective, the following methodological approach is employed: data collection, mathematical modeling and algorithm development, integration with containerization platforms, and experimental validation.

RESEARCH RESULTS

Microservices architecture offers extensive scalability and update flexibility since each service is deployed in an independent container and can be maintained and updated separately. However, as interactions between microservices become more complex, the risk of increased network latency and excessive resource consumption grows. In recent years, several approaches have been proposed to mitigate these overhead costs, including:

1. Optimization of deployment topology. Approaches that account for the microservices dependency graph (Call Graph) aim to place interdependent services on physically close nodes, reducing the number of inter-node transitions and, consequently, network latency. For instance, within the MOTAS concept, the microservices graph is divided into hierarchical clusters to confine high-intensity traffic within specific node groups.
2. Utilization of orchestration tools (Kubernetes, Docker Swarm) with extended plugins. Standard Kubernetes mechanisms, such as Deployment and Service, simplify the management of microservices' lifecycle but do not always consider network proximity. To address this, scheduler extenders and custom controllers are proposed, which analyze physical machine load, the placement of "neighboring" microservices, network metrics, and potential Service Level Agreement (SLA) constraints [6].
3. Heuristic and intelligent scheduling algorithms (PSO, genetic algorithms). Research indicates that particle swarm optimization methods, when enhanced, can accommodate both the distributed nature of microservices and real-time resource monitoring metrics. A crucial aspect is the correct selection of a fitness function, incorporating network metrics (latency, bandwidth) and fault tolerance criteria.

Thus, there is a trend toward comprehensive approaches combining network interaction analysis, dynamic resource monitoring, and load balancing, highlighting the need for continued research in this area [1, 4, 6].

A key feature of microservices architecture is its reliance on frequent service-to-service communication. Data exchange within cloud data centers can introduce significant overhead, particularly when tightly coupled microservices are distributed across different modules.

- Network topology. Most modern data centers use a Fat-tree (or Clos) architecture, theoretically providing high bandwidth through multiple paths between nodes. However, under practical loads, network switches at the aggregation or core levels become bottlenecks, limiting network bandwidth and increasing latency due to traffic congestion [1].
- Microservices interaction speed. Remote Procedure Call (RPC) and RESTful call patterns require constant data exchange, causing the total number of transmitted packets to grow exponentially as the number of microservices increases [4].
- Fault tolerance level. To improve reliability and availability, microservices are often replicated, generating additional network traffic between replicas and source services [7].

To systematize the existing approaches in latency reduction, load balancing, and fault tolerance, Table 1 presents a comparative analysis of different strategies.

Table 1. Comparison of approaches to reducing latency and improving resource allocation efficiency [4-7].

Approach	Key Concept	Advantages	Limitations
Hierarchical partitioning mystery of time and space(MOTAS)	Dividing microservices into groups within a shared dependency graph	Reduces inter-group traffic, clear management structure	Does not account for dynamic load changes, requires static graph analysis
Scheduler extender in Kubernetes	Extending the standard scheduler with additional plugins	Flexible integration with Kubernetes, considers network metrics and resource utilization	Requires real-time monitoring setup, increases management complexity
Modified PSO (particle swarm optimization)	Heuristic search for optimal service placement across nodes	Fast convergence in large clusters, ability to consider multiple criteria simultaneously	May get stuck in local optima, requires parameter tuning (velocity, inertia weight, etc.)
Round-robin with resource awareness	Load distribution across servers in a cyclic manner with thresholds	Simple implementation, effective under minor load fluctuations	May not consider network latency factors and complex microservices dependencies
Replication with fault tolerance and SLA considerations	Duplicating critical microservices across multiple nodes	Eliminates single points of failure, enhances overall availability	Increases total traffic, complicates management, requires an advanced node selection strategy

As shown in Table 1, several approaches offer strong advantages, such as latency reduction (e.g., MOTAS in static graph analysis) and improved availability (replication). However, none of the listed methods independently solve the combined challenge of minimizing network latency, balancing resources, and ensuring high fault tolerance. This confirms the need for further research aimed at developing a comprehensive strategy that integrates heuristic algorithms (e.g., modified PSO), distributed scheduling mechanisms (e.g., round-robin with dynamic thresholds), and orchestration tools at the Kubernetes level.

Thus, the next section will introduce an original concept that combines microservices replication with load and network topology awareness. This approach aims to simultaneously reduce network latency and minimize single points of failure, which is particularly relevant for large-scale cloud systems.

Proposed Methods and Scheduling Strategy for Reducing Latency

The particle swarm optimization algorithm was originally proposed for solving continuous optimization problems. In its classical form, PSO models particles "swarming" in a multidimensional solution space, where each particle stores its current position and velocity while considering the best solutions found both individually and by the swarm. In microservices scheduling, this approach is applied to determine the "optimal" placement of services across physical machines (Particles → Microservices deployment options), thereby reducing latency and improving resource utilization [2, 3].

However, classical PSO does not account for:

1. The microservices dependency graph (Call Graph), which determines which services interact frequently.
2. Fault tolerance constraints that require replication of critical microservices across different nodes [7].
3. The risk of network congestion when a large number of interdependent microservices are deployed at opposite ends of a data center [1].

Key PSO modifications in the proposed approach:

1. Dependency clustering. Before PSO execution, a subset of microservices closely related to the target microservice (e.g., a subservice handling requests) is formed. This "localizes" the search space and reduces the probability of particles unnecessarily jumping to nodes that are inefficient in terms of network latency [3].
2. Fault tolerance awareness. During microservice replication or migration, the algorithm verifies whether critical services are already present on a given physical machine (PM). This prevents the "single point of failure" scenario in the event of PM failure.
3. Additional component in the fitness function. The classical fitness function considers "personal" and "global" optima. A network cost factor is introduced: the higher the number of inter-node "hops" (or the greater the total latency) in interactions with dependent microservices, the lower the fitness score. If a microservice can be placed closer to its dependencies without exceeding CPU/memory utilization thresholds, the solution's overall fitness improves.
4. Velocity clamping. In classical PSO, particle velocity can increase rapidly, causing particles to "overshoot" optimal solutions. A velocity cap is introduced to ensure slower movement, allowing the swarm to explore the vicinity of potential optimal solutions more thoroughly. This technique is often used in solving discrete or mixed scheduling problems.

To illustrate the key differences between classical PSO and the modified PSO in the context of microservices scheduling, Table 2 presents a comparative analysis.

Table 2. Comparison of Classic PSO and Modified PSO for Microservices [3, 5, 6].

Criterion	Classic PSO	Modified PSO
Dependencygraph awareness	Not considered. Particles search uniformly across the entire solution space	A subset of particles is directed toward nodes close to dependent microservices (Call Graph analysis), reducing network "hops"
Fault tolerance	Not explicitly considered. Placement may be "random," leading to a single point of failure	Critical service presence on a PM is verified; duplication of critical microservices on the same node is avoided
Fitness function	Considers only "personal" and "global" optima (e.g., minimizing total resource costs)	Includes a network latency component and a load threshold factor to prevent SLA violations
Velocity control	Particle velocity can increase without strong constraints	A velocity cap is introduced to allow more thorough exploration of solution areas
Scalability	Good, but prone to local optima in complex spaces	Enhanced scalability and reduced risk of local optima by considering network and resource profiles of the cloud cluster

Thus, when properly implemented and configured (adjusting inertia weight, maximum velocity, load thresholds, etc.), the modified PSO demonstrates improved results by simultaneously considering network and resource constraints.

After determining an "acceptable" set of physical machines for microservices (the output of the modified PSO), it is necessary to evenly distribute replicas and new containers across servers [7]. For this purpose, the Round Robin (RR) algorithm is used, enhanced with the following modifications:

1. Dynamic CPU/memory utilization threshold. If a node (PM) already exceeds a predefined threshold (e.g., 70% CPU and 75% memory), RR "skips" this node and proceeds to the next one in the cycle without attempting to allocate another container there. Load data is collected from real-time monitoring systems.
2. Prioritization of nodes closer to dependent microservices. From the set of candidates proposed by PSO, the nearest available node is selected (provided that its resource utilization threshold is not exceeded). If all "close" nodes are overloaded, the RR algorithm moves to the "next tier" of nodes, and so on.
3. Replication awareness. If it is necessary to deploy *n* replicas of a microservice, the RR algorithm distributes them cyclically across eligible nodes to prevent excessive concentration of replicas on a single PM [1].

This combination of PSO and RR prevents situations where a service is "ideally" placed in terms of latency reduction but ultimately overloads a node, leading to the risk of SLA violations. Additionally, it supports more flexible load balancing, which is crucial for systems with a large number of microservices and unpredictable load spikes [4].

To ensure the practical applicability of the described methodology (PSO+RR), it must be integrated with an orchestration system. The most widely used platform is Kubernetes, which already provides:

- Containerization (Pods, Deployments, Services);
- Scaling (ReplicaSet, Horizontal Pod Autoscaler);
- Scheduler extensions (Scheduler Extender, Plugins).

Below, Figure 1 illustrates the stages of integrating the PSO and RR combination.

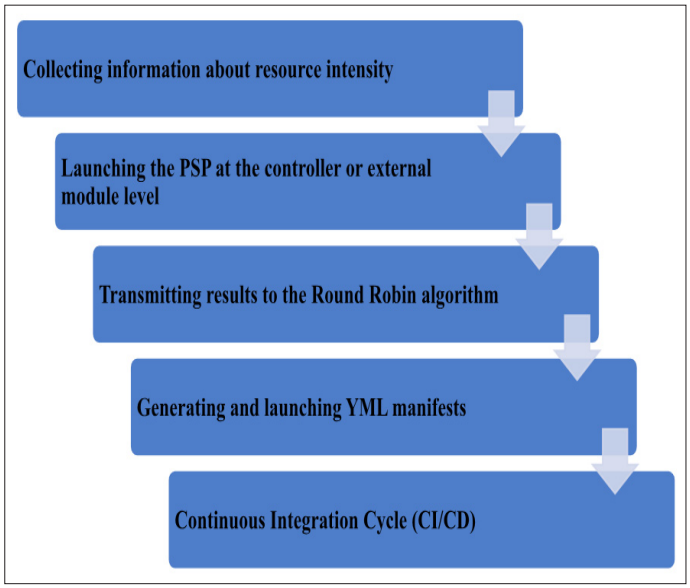

Fig.1. The stages of integration of the combination of PSO and RR [6].

Experimental Validation and Results

This section describes the methodology for experimentally validating the proposed scheduling strategy, the datasets used for simulation, and key results along with their interpretation. To simulate real-world microservices deployment scenarios, two primary datasets were used:

1. Alibaba Trace. Published by Alibaba Inc. [1], this dataset reflects the operation of a large-scale cluster utilizing a microservices architecture and includes the following types of data:

- Identifiers of virtual machines (VMs) and containers, their timestamps, and resource usage parameters (CPU, memory);

- Call Graphs indicating dependencies between microservices;
- Network traffic intensity between data center nodes (Fat-tree topology).

2. Google Trace. Provided by Google Cloud Docs [4] within the Borg system. It covers distributed computing tasks and CPU/memory usage information. Unlike the Alibaba dataset, it does not include data on microservices dependencies; therefore, dependencies were artificially generated based on probabilistic interaction distributions. This dataset allows the evaluation of the strategy's behavior under more dynamic loads and a high frequency of microservice restarts.

In all experiments, it was assumed that the Kubernetes orchestration system managed clusters consisting of several hundred physical machines (PMs) distributed in a Fat-tree topology. To validate the proposed scheduling strategy (PSO+RR), two types of scenarios were considered. For a comprehensive illustration, Table 3 presents a comparative analysis of three approaches: (1) classical Kubernetes, (2) PSO with an adaptive threshold (without considering network proximity), and (3) the PSO+RR strategy.

Table 3. Comparative analysis by three metrics. (compiled independently based on [1, 4, 6, 7, 8]).

Metric	Classical Kubernetes Scheduler	PSO with Adaptive Threshold	PSO+RR (Proposed Strategy)
Reduction in Network Traffic (relative to the initial state), %	10–12%	18–21%	33–36%
Average Latency Reduction, %	30–35%	55–60%	81–84%
Standard Deviation of CPU Utilization	25–30 (arbitrary units)	18–22	12–14

As seen in Table 3, the classical Kubernetes scheduler does not account for network proximity, leading to only moderate performance improvements. PSO with an adaptive threshold primarily focuses on resource management but does not sufficiently optimize network communication. The proposed strategy (PSO+RR) achieves the best reduction in network traffic and latency while also achieving more balanced CPU/memory utilization.

The experiments demonstrate that the modified scheduling strategy, which considers the microservice dependency graph, resource thresholds, and load balancing, yields higher performance across all three metrics.

Ultimately, the results confirm the hypothesis that the combined use of modified PSO (which considers the network structure) and adaptive Round Robin (which accounts for resource thresholds) allows for a comprehensive minimization of latency, balanced load distribution, and enhanced fault tolerance in microservice environments.

CONCLUSION

The conducted research has demonstrated the effectiveness of a comprehensive microservice scheduling strategy that combines a modified particle swarm optimization (PSO) algorithm with an enhanced load distribution mechanism (Round Robin) while incorporating network topology proximity principles. This approach addresses several key challenges:

1. Reduction of network latency by placing closely related microservices on physically adjacent nodes and minimizing inter-node transitions.
2. Load balancing based on dynamic resource utilization thresholds (CPU, memory), preventing localized overloads.
3. Increased fault tolerance by preventing the co-location of critical services on the same node and ensuring even replication distribution.

Experiments conducted on Alibaba and Google datasets showed that the proposed strategy outperforms the classical Kubernetes scheduler and other baseline heuristic methods. A significant reduction in overall network traffic was observed (up to 33–36% in static scenarios), with latency decreasing by more than 80% in some cases. At the same time, resource utilization levels increased, and the standard deviation of load distribution was nearly halved, indicating more efficient resource usage and reduced risk of SLA violations.

Thus, the proposed approach is highly relevant for large-scale cloud clusters where minimizing network latency, improving system availability, and optimizing hardware utilization are critical requirements. Future extensions of this methodology may include the integration of machine learning algorithms (such as Q-learning or other reinforcement learning techniques) for predictive microservice scheduling based on historical load data and evolving service dependency structures.

REFERENCES

1. Microservice Architecture of Alibaba. [Electronic resource] Access mode: https://www.abhishek-tiwari.com/microservice-architecture-of-alibaba / (date of access: 02/03/2025).
2. Balalaie A., Heydarnoori A., Jamshidi P. Microservices architecture enables devops: Migration to a cloud-native architecture //Ieee Software. – 2016. – Vol. 33 (3). – pp. 42-52.
3. Alelyani A., Datta A., Hassan G. M. Optimizing Cloud Performance: A Microservice Scheduling Strategy for Enhanced Fault-Tolerance, Reduced Network Traffic, and Lower Latency //IEEE Access. – 2024.- Vol.12. – pp. 35135–35153.

4. Cloud Trace overview . [Electronic resource] Access mode: https://cloud .google.com/trace/docs/overview (date of request: 02/03/2025).
5. Velepucha V., Flores P. A survey on microservices architecture: Principles, patterns and migration challenges //IEEE Access. – 2023. – Vol.11. – pp. 88339-88358/
6. Kubernetes Scheduler. [Electronic resource] Access mode: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler /(date of access: 02/03/2025).
7. Li X. et al. Topology-aware scheduling framework for microservice applications in cloud //IEEE Transactions on Parallel and Distributed Systems. – 2023. – Vol. 34 (5). – pp. 1635-1649.
8. Alelyani A., Datta A., Hassan G. M. Optimizing Cloud Performance: A Microservice Scheduling Strategy for Enhanced Fault-Tolerance, Reduced Network Traffic, and Lower Latency //IEEE Access. – 2024. – pp.1-10.