

DOI: <https://doi.org/10.15276/ict.03.2026.68>

UDC 004.75:519.17

Graph-based resource optimization considering dependencies between microservices: a survey of algorithms and approaches

Yevhen I. Lysevych ¹⁾

ORCID: <https://orcid.org/0009-0004-6161-9453>; lisevich.e.i@gmail.com

¹⁾ Odesa I. I. Mechnikov National University, 2, Vsevoloda Zmiiienka St. Odesa, 65082

ABSTRACT

Modern cloud-native systems built on microservice architectures present a major challenge for resource optimization: individual services are not independent but form complex networks of dependencies where scaling, placement, and capacity decisions for one service directly affect the performance and resource consumption of others. This paper presents a comprehensive survey of graph-theoretic algorithms and approaches applicable to dependency-aware resource optimization in microservice systems. We formalize a microservice system as a weighted directed graph in which vertices represent services and infrastructure components, edges represent runtime calls and data flows, and weights encode metrics such as request rate, latency, retry rate, and network cost. We systematically analyze eleven categories of graph algorithms - including topological sorting, Critical Path Method, network flow, shortest path algorithms, centrality metrics (betweenness, PageRank), community detection (Louvain), graph partitioning (METIS library), minimum spanning trees, graph coloring, directed acyclic graph scheduling (Heterogeneous Earliest Finish Time), and strongly connected component detection - evaluating their time and space complexity, scalability, and applicability to microservice resource optimization tasks: bottleneck identification, coordinated scaling, placement optimization, and capacity planning. We present a load propagation model based on adjacency matrix inversion that enables formal calculation of downstream load amplification across the entire dependency structure, including a discussion of convergence conditions for cyclic dependency graphs. Furthermore, we review recent advances in graph neural networks combined with reinforcement learning for adaptive autoscaling, including systems Graph Attention-aware Fusion Networks, predict end-to-end response times in Graph Neural Networks, DeepScaler, and HGraphScale, which demonstrate improvements of up to nineteen percents in compute cost (Graph Attention-aware Fusion Networks), up to forty percents in violation of service-level agreement reduction (DeepScaler), and up to eighty percents in response time reduction (HGraphScale) compared to traditional approaches - noting that these results span different benchmarks and metrics. Our comparative analysis reveals that combining classical graph algorithms for structural analysis with graph neural network-based methods for dynamic adaptation is the most promising direction for production-grade, dependency-aware resource optimization.

Keywords: Microservice architecture; graph theory; resource optimization; dependency graph; critical path method; network flow; graph neural networks; autoscaling; centrality metrics; Kubernetes

For citation: Lysevych Y. I. “Graph-based resource optimization considering dependencies between microservices: a survey of algorithms and approaches”. *Informatics. Culture. Technology*. 2026; Vol. 3 No. 2(4): 878–896. DOI: <https://doi.org/10.15276/ict.03.2026.68>

The study aims to provide a systematic survey of graph-theoretic algorithms and modern graph-neural-network approaches for dependency-aware resource optimization in microservice architectures, to formalize the transmission of load across service dependencies, and to define a practical algorithmic pipeline for bottleneck identification, coordinated scaling, and placement optimization.

Methods. This survey follows a structured literature selection methodology aligned with PRISMA guidelines for scoping reviews.

Search strategy. We searched four electronic databases on March 15, 2025. The exact query strings, adapted to each database's syntax, were:

IEEE Xplore: (“microservice*” OR “micro-service*” OR “cloud-native”) AND (“graph” OR “dependency graph” OR “call graph” OR “service mesh”) AND (“resource” OR “scaling” OR “optimization” OR “scheduling”), filtered to 2015-2025.

ACM Digital Library: [[All: microservice*] OR [All: “cloud-native”]] AND [[All: graph] OR [All: “dependency graph”] OR [All: “call graph”]] AND [[All: resource] OR [All: scaling] OR [All: optimization]], filtered to 2015-2025.

Springer: (“microservice” OR “cloud-native”) AND (“graph” OR “dependency”) AND (“resource optimization” OR “autoscaling”), filtered to 2015-2025, English only, Computer Science subject.

arXiv (cs.DC, cs.SE, cs.LG): (microservice OR cloud-native) AND (graph OR dependency) AND (resource OR scaling OR optimization), filtered to 2015-2025.

Additionally, we performed backward snowballing on the reference lists of included papers and forward snowballing via Google Scholar citations of key works [1], [2], [4], [7].

Inclusion criteria: (IC1) peer-reviewed conference papers, journal articles, or preprints with at least 5 citations (or published after January 2024); (IC2) written in English; (IC3) explicitly addresses microservice or cloud-native architectures; (IC4) employs graph-theoretic or graph-neural-network methods; (IC5) targets resource optimization, scaling, scheduling, or capacity planning.

Exclusion criteria:

(EC1) workshop papers, extended abstracts, or posters without full text;

(EC2) studies addressing graph-based methods without application to resource optimization (e.g., pure fault localization or testing);

(EC3) duplicate publications (shorter conference version when a journal extension exists);

(EC4) studies not modeling inter-service dependencies.

Screening process. The initial database search yielded 707 records (IEEE Xplore: 165, ACM DL: 225, Springer: 228, arXiv: 89). After removing 143 duplicates by DOI and title matching (using Zotero reference manager), 564 unique records remained. Title and abstract screening against inclusion criteria IC1 – IC5 excluded 398 records. Full-text assessment of the remaining 166 papers excluded 123 (reasons: no graph model – 52; no resource optimization focus – 38; insufficient technical depth – 21; non-English – 7; retracted or superseded – 5). The final corpus comprises 43 sources.

Quality assessment. Each included paper was evaluated on three dimensions:

(Q1) presence of formal problem definition or mathematical model;

(Q2) experimental validation on realistic workloads (production traces, standard benchmarks such as DeathStarBench [29], or simulations with ≥ 10 services);

(Q3) comparison with at least one baseline method.

Papers satisfying all three criteria were classified as high quality (24 papers); those satisfying two as medium quality (14 papers); the remaining 5 are foundational works (algorithms textbooks, seminal algorithm papers) included for theoretical completeness.

Fig. 1 presents the PRISMA-style selection flowchart.

Scientific novelty. Unified classification of 11 graph algorithm categories specifically for microservice resource optimization (topological sort, CPM, network flow, centrality, community detection, partitioning, shortest paths, MST, graph coloring, DAG scheduling, SCC). Previous works covered these in isolation. Load propagation model with clear convergence conditions for cyclic dependency graphs ($\rho(A) < 1$) and a direct link to SCC detection. Comparative analysis of classical algorithms vs. GNN approaches (GRAF, PERT-GNN, DeepScaler, HGraphScale), showing scalability tiers and justifying the hybrid approach. This classification differs from prior surveys in three specific ways. First, Nguyen et al. [7] focus exclusively on GNN architectures and do not analyze classical graph algorithms; our survey covers 11 classical categories alongside GNN-based methods. Second, Waseem et al. [43] map microservice practices to DevOps lifecycle stages without classifying optimization algorithms; our taxonomy is organized by algorithm category and optimization task. Third, no prior survey provides a formal load propagation model connecting graph structure to resource requirements. Table 4 details this comparison, and Table 5 maps each surveyed work to the proposed classification categories.

Relevance. Microservice architecture has become the dominant paradigm for building large-scale cloud-native applications. Organizations such as Meta (over 18,500 services) [1], Uber (approximately 4,500 services) [2], Netflix (hundreds of microservices) [3], and Alibaba (over 20,000 services in production traces) [4] operate systems where thousands of interdependent services process millions of requests per second. Resource optimization in such systems - determining how much CPU, memory, and replicas to allocate to each service, and where to place workloads – directly impacts both operational cost and service quality.

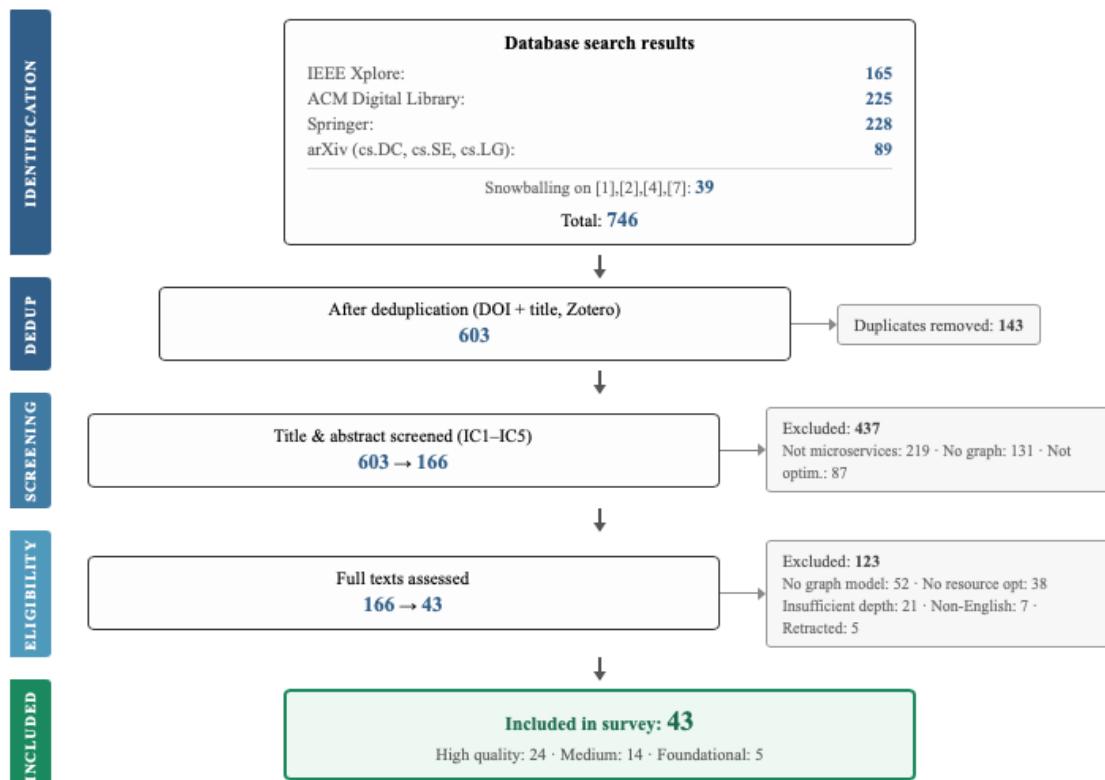


Fig. 1 The PRISMA-style selection flowchart

Source: compiled by the authors

Traditional approaches to resource management treat each service independently: the Kubernetes Horizontal Pod Autoscaler scales individual workloads based on local CPU or memory utilization [5], while the Vertical Pod Autoscaler adjusts per-container resource requests. However, microservices are not isolated components; they form networks of dependencies where a single external request may traverse 10-15 services [2], generating significant internal traffic amplification. A system receiving 1,000 external requests per second may produce 9,000 or more internal calls. Consequently, scaling one service without considering its downstream dependencies may simply shift the bottleneck rather than resolve it [6].

This fundamental limitation of per-service optimization motivates a graph-based approach. When a microservice system is modeled as a weighted directed graph - where vertices represent services and edges represent dependencies annotated with metrics such as request rate, latency, and fan-out - graph algorithms can formally analyze load propagation, identify bottleneck chains, optimize placement, and coordinate scaling decisions across the entire dependency structure [7], [8]. The paper is organized as follows. The literature selection methodology is described first; the graph model is then formalized; a survey of eleven applicable algorithm categories follows; next, the load propagation model is derived with discussion of convergence conditions; modern graph-neural-network approaches are reviewed; and comparative analysis leads to conclusions and future directions.

Practical significance. The results of this survey can be directly applied by engineers working with microservice systems. The load propagation model allows quick what-if analysis – for example, estimating how a 30 % traffic increase at the entry point will affect every downstream service before bottlenecks appear. The paper provides specific algorithm recommendations for specific tasks: CPM for finding bottleneck chains, METIS for service placement optimization (which can save around \$3,500/month on cross-zone data transfer costs for a single high-traffic service pair), and PageRank for identifying which services should get priority resource allocation. These algorithms naturally combine into an optimization pipeline – topological sort for deployment ordering, then CPM for bottleneck detection, centrality for priority ranking, community detection for clustering, and graph partitioning for placement. The paper also gives an honest assessment of

GNN limitations such as cold-start problems, training scalability, and low interpretability, which helps practitioners decide when classical algorithms are enough and when investing in GNN-based solutions is justified.

Graph model of a microservice system. A microservice system can be naturally represented as a weighted directed graph:

$$G = (V, E, W) \quad (1)$$

where V is the set of vertices representing services and infrastructure components; $E \subseteq V \times V$ is the set of directed edges representing runtime calls and data flows, and $W: E \rightarrow \mathbb{R}^n$ is a weight function assigning a vector of metrics to each edge.

Vertices in this model correspond to microservices, databases, message queues, caches, API gateways, and external APIs. Each vertex $v \in V$ has attributes including CPU and memory requests and limits, replica count, actual resource utilization, maximum concurrency, SLO/SLA targets, business priority tier, and cost per replica [9]. Empirical analysis of production systems shows that these graphs are sparse (Meta’s 18,500 services produce only ~393,622 edges versus ~342 million possible), modular, and dynamic, with services constantly being added and removed [1].

Edges represent dependencies between components and carry weight vectors including: requests per second, average calls per upstream request (fan-out factor), P50/P95/P99 latency, retry rate, error rate, payload size, network cost, synchronous/asynchronous flag, and dependency criticality [7]. The distinction between synchronous edges (REST/HTTP, gRPC) and asynchronous edges (message queues, event buses) is essential: synchronous dependencies create tight temporal coupling where downstream latency directly adds to upstream response time, while asynchronous edges decouple temporal behavior but introduce consumer lag as a scaling signal [10].

An important architectural consideration is the difference between static and dynamic graphs. Static graphs, derived from configuration and source code analysis, represent the space of possible dependencies. Dynamic graphs, constructed from distributed traces (OpenTelemetry, Jaeger, Zipkin) and service mesh telemetry (Istio, Linkerd), capture actual runtime behavior [11]. Resource optimization requires dynamic graphs, as the effective topology may change with workload patterns, circuit breaker activations, and canary deployments. Morphis [12] demonstrates that production call graphs at ByteDance concentrate around a small set of recurring invocation patterns despite continuous structural evolution.

Fig. 2 illustrates a typical microservice dependency graph where edge weights represent the average number of downstream calls generated per upstream request.

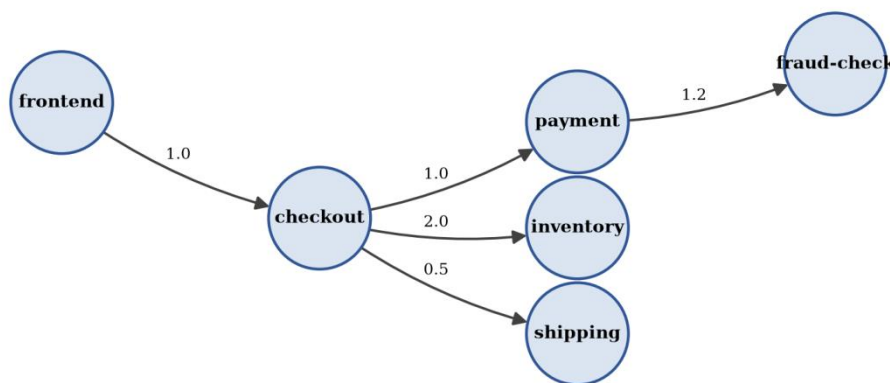


Fig. 2. Microservice dependency graph with fan-out weights

Source: compiled by the authors

Applicability constraints and required transformations. The graph algorithms surveyed in “Survey of graph algorithms” section were originally developed for specific graph classes. Applying them to real microservice dependency graphs requires explicit transformations and entails limitations that must be understood by practitioners.

DAG requirement and cycle handling. Topological sorting, CPM, and DAG scheduling algorithms require directed acyclic graphs. However, production microservice graphs frequently contain cycles arising from bidirectional health checks, callback patterns, and shared-state services – Meta's analysis [1] identified multiple non-trivial strongly connected components in their 18,500-service graph. To apply DAG-based algorithms, cycles must first be resolved through SCC condensation: Tarjan's algorithm (described in the “Strongly connected component detection” subsection) identifies all SCCs in $O(V + E)$ time, and each SCC is contracted to a single super-vertex, producing a condensation DAG. The condensed graph preserves the inter-cluster dependency structure while collapsing intra-cluster cycles. Within each SCC, services must be treated as a single scaling unit or analyzed separately using cycle-aware methods (e.g., the spectral radius condition in “Load propagation model” section / equation (4)). This two-phase approach – SCC condensation followed by DAG analysis – is the standard transformation for applying topological sort and CPM to cyclic graphs [13].

Parallel branches and fork-join semantics. CPM computes the longest path through a DAG, which correctly models sequential dependencies. However, microservice request flows frequently involve parallel fan-out: a single upstream request triggers multiple concurrent downstream calls (e.g., checkout simultaneously calling payment, inventory, and shipping). For parallel branches, the end-to-end latency at a join point equals $\max$ (latencies of parallel branches), not their sum. Standard CPM handles this correctly when the DAG is structured with explicit fork and join vertices – the forward pass naturally computes the maximum across incoming edges at each join vertex. Practitioners must ensure that the dependency graph includes explicit join vertices where parallel branches reconverge, rather than modeling parallel calls as independent sequential paths.

Shortest path versus end-to-end latency. Dijkstra's and Bellman–Ford algorithms compute minimum-weight paths, which represent minimum achievable latency routes. However, actual end-to-end latency includes vertex processing time (service computation), queueing delays, serialization overhead, and retry latency — not just edge (network) latency. For accurate latency analysis, edge weights must include the full processing latency of the destination service: $W(U,V) = \text{network latency}(U,V) + \text{processing time}(V) + \text{queue wait}(V)$. With this augmented weight definition, shortest path algorithms correctly compute minimum end-to-end latency. Without it, the results represent only network-optimal routes and may underestimate actual latency by an order of magnitude.

Network flow capacity versus service capacity. In classical network flow formulations, edge capacities represent link bandwidth. In the microservice context, the bottleneck is typically service processing capacity (requests per second given allocated CPU/memory), not network bandwidth. The network flow model must therefore use vertex capacities (transformed to edge capacities via standard vertex-splitting [16]): each service vertex V is split into V_{in} and V_{out} connected by an edge with capacity equal to the service's maximum throughput. This transformation preserves the max-flow/min-cut properties while correctly modeling service-level bottlenecks rather than network-level constraints.

Survey of graph algorithms. This section surveys eleven categories of graph algorithms evaluated for their applicability to resource optimization in microservice dependency graphs. For each category, time and space complexity, scalability, support for weighted and directed graphs, and the specific microservice optimization task addressed are analyzed.

Topological sorting for deployment ordering. Topological sorting establishes a linear ordering of services such that for every dependency edge (u, v) , service u appears before v . This is fundamental for deployment ordering, initialization sequencing, and rolling updates. Two optimal algorithms exist: Kahn's algorithm (BFS-based) iteratively removes nodes with zero in-degree, naturally producing level-by-level ordering suitable for parallel deployment; DFS-based topological sort uses reverse post-order traversal and extends naturally to Tarjan's algorithm for detecting strongly connected components – clusters of circular dependencies [13]. Both algorithms achieve optimal $O(V + E)$ time complexity and scale linearly. For a typical microservice system with 50–5,000 services and linearly many dependencies, execution is effectively instantaneous. Kahn's

algorithm is preferred when parallel deployment levels are needed; DFS-based is preferred when combined with strongly-connected-component analysis.

Applicability note: Topological sorting requires a DAG. For microservice graphs with cycles, SCC condensation must be applied first (see “Applicability constraints” above). Within each SCC, deployment ordering requires additional strategies such as simultaneous deployment or staged rollout with backward compatibility.

Critical Path Method for bottleneck identification. The Critical Path Method (CPM) is highly relevant for microservice resource optimization. It computes the longest weighted path through a directed acyclic graph, directly identifying the bottleneck chain of service calls that determines end-to-end response time [14]. The algorithm performs a forward pass computing earliest start times and a backward pass computing latest start times for each service in the request processing pipeline. Services with zero slack (float) constitute the critical path – these are the services where resource investment yields maximum latency reduction. For example, consider a checkout flow with an SLO of 300 ms traversing the path: frontend (20 ms) → checkout (40 ms) → payment (60 ms) → fraud-check (80 ms) → external API (100 ms). CPM reveals that scaling fraud-check or optimizing the external API call produces greater benefit than scaling frontend, despite frontend receiving the highest traffic. The algorithm runs in $O(V + E)$ time - linear in graph size – making it suitable for real-time analysis [14]. The stochastic extension, PERT (Program Evaluation and Review Technique), models latency variability using three-point estimates (optimistic, most likely, pessimistic), which naturally align with P50/P95/P99 latency percentiles from monitoring data. PERT enables probabilistic SLO compliance estimation. The PERT-GNN system [15] demonstrated that combining PERT analysis with Graph Transformers significantly outperforms baseline methods for end-to-end latency prediction on Alibaba production traces.

Applicability note: CPM requires a DAG and assumes sequential execution along each path. For parallel fan-out patterns, the dependency graph must include explicit fork-join structure so that the forward pass correctly computes $\max(\text{parallel branch latencies})$ at join vertices. Edge weights should include full processing latency (network + computation + queue), not just network delay.

Network flow for capacity planning. Network flow algorithms model capacity constraints in microservice architectures. Each service has a processing capacity (throughput limit), and edges represent communication channels with bandwidth constraints. The maximum flow algorithm determines the maximum throughput a system can sustain, while the min-cut theorem identifies the bottleneck set - the minimum number of service connections whose saturation limits overall system throughput [16]. Three algorithms are applicable: Ford–Fulkerson ($O(E \times f_{\max})$, conceptual foundation), Edmonds–Karp ($O(VE^2)$, strongly polynomial, suitable for sparse graphs), and Push–Relabel ($O(V^3)$, asymptotically fastest for dense graphs and parallelizable). Push–Relabel is recommended for capacity planning in large deployments due to its parallelizability and independence from edge count. Min-cut analysis is particularly valuable for identifying single points of failure and designing redundancy.

Applicability note: In the microservice context, capacity constraints are typically at service vertices (throughput limits), not at edges (network bandwidth). Vertex-splitting transforms each service into an edge with capacity equal to its maximum throughput, enabling standard max-flow algorithms to identify service-level bottlenecks.

Centrality metrics for critical service identification. Centrality algorithms identify the most critical services - those whose failure or performance degradation has the greatest cascading impact. Betweenness centrality, computed by Brandes’ algorithm in $O(VE)$ time, identifies gateway and broker services through which a disproportionate fraction of request paths flow [17]. Services with high betweenness are bottleneck candidates and single points of failure; they should receive priority resource allocation and redundancy. PageRank, running in $O(k(V + E))$ time with $k \approx 50\text{--}100$ iterations, captures transitive importance: a service is ranked highly if it is depended upon by many important services. This naturally models cascading failure propagation – a failure in a high-PageRank service affects the most important parts of the system. PageRank was originally designed for web-scale graphs with billions of nodes, so scalability is not a concern for microservice

deployments [18]. The MicroRank system [19] applies PageRank-style propagation to end-to-end latency issue localization in microservice environments. Closeness centrality identifies services that can reach or be reached by all other services with minimum total latency – useful for determining optimal placement of shared infrastructure services such as caches and configuration servers.

Community detection for service clustering. Community detection algorithms identify groups of microservices that are more densely interconnected with each other than with the rest of the system. This reveals natural service clusters for team ownership boundaries (Conway's Law alignment), failure domain isolation, independent scaling units, and candidate microservice merges or splits [20]. The Louvain algorithm achieves near-linear $O(n \log n)$ time through greedy modularity optimization and produces hierarchical community structure matching organizational levels (team $\rightarrow$ domain $\rightarrow$ platform). Its improved variant, Leiden, addresses the resolution limit problem for detecting small communities in large graphs. For real-time dynamic environments, Label Propagation Algorithm (LPA) provides the fastest detection at $O(n + m)$ with trivial parallelizability. Alibaba's analysis [4] revealed that microservice graphs naturally cluster into categories with intra-cluster similarity $3.4\times$ higher than inter-cluster similarity.

Graph partitioning for placement optimization. Graph partitioning divides the dependency graph into balanced subsets with minimal inter-partition communication, directly applicable to placing service groups on physical nodes, designing failure domains, and multi-region deployment. The METIS algorithm [21] is the clear choice for production systems, achieving near-linear $O(V + E)$ time through a three-phase multilevel approach: coarsening (collapsing the graph via heavy-edge matching), initial partitioning, and uncoarsening with refinement. METIS supports vertex weights (resource requirements) and edge weights (communication volumes), directly modeling the microservice placement problem. Co-locating frequently communicating services reduces cross-zone traffic costs (approximately \$0.01/GB per direction in AWS) and latency (0.5–2 ms per zone crossing). For a service pair exchanging 1,000 requests per second with 40 KB payloads, cross-zone placement incurs roughly \$3,500/month in data transfer costs alone. METIS-based partitioning minimizes such costs while respecting capacity constraints.

Shortest path algorithms for latency-aware routing. Shortest path algorithms – Dijkstra's algorithm and Bellman–Ford – compute minimum-weight paths through weighted graphs and are directly applicable to latency-aware request routing and SLO compliance analysis in microservice dependency graphs. Dijkstra's algorithm, running in $O((V + E) \log V)$ time with a binary heap, computes the minimum-latency path from any entry point (API gateway) to every downstream service, identifying the fastest route through the dependency graph [13]. This is particularly relevant for systems with multiple paths between services – for instance, when a request can reach a database service through either a caching layer or a direct query service. Bellman–Ford, though slower at $O(VE)$, handles negative edge weights, which can model latency reductions from caching or connection pooling [28]. In microservice environments, shortest path analysis serves three practical purposes: first, identifying the minimum achievable end-to-end latency for SLO feasibility assessment; second, detecting routing inefficiencies where requests traverse longer paths than necessary; third, computing the latency impact of service degradation by re-running the algorithm with updated edge weights when a service's response time increases [29]. The Seer system [30] leverages deep learning on microservice call graphs for performance debugging, where shortest-path analysis through dependency structures helps localize the root cause of latency anomalies.

Applicability note: Edge weights must include the full processing latency of the destination service (network delay + computation time + queueing delay), not just network latency. Without this, shortest-path results represent network-optimal routes but may significantly underestimate actual end-to-end latency.

Minimum spanning trees for infrastructure cost optimization. Minimum spanning tree (MST) algorithms – Kruskal's and Prim's – find the subset of edges that connects all vertices with minimum total weight and are applicable to infrastructure topology optimization in microservice deployments. Kruskal's algorithm runs in $O(E \log E)$ time using union-find data structures, while Prim's achieves $O(E \log V)$ with a priority queue [13]. In microservice resource optimization, MST

algorithms address the problem of designing minimal-cost communication overlays. When deploying a service mesh (Istio, Linkerd) or configuring inter-service communication across multiple availability zones or regions, MST identifies the minimum set of direct connections needed to maintain full connectivity while minimizing total network cost [31]. For a system with N services across K availability zones, the cross-zone communication cost can be modeled as a complete weighted graph where edge weights represent data transfer costs; the MST of this graph provides the optimal communication topology. Additionally, MST-based analysis reveals redundancy requirements: edges not in the MST represent backup communication paths, and the difference between MST cost and the cost of adding these edges quantifies the price of fault tolerance [32]. Borůvka's algorithm, with the same $O(E \log V)$ complexity, is particularly suitable for parallel implementation in distributed microservice environments.

Graph coloring for resource conflict avoidance. Graph coloring assigns labels (colors) to vertices such that no two adjacent vertices share the same color; the chromatic number $\chi(G)$ is the minimum number of colors required. While optimal graph coloring is NP-hard in general, greedy heuristics achieve $O(V + E)$ time and produce near-optimal solutions for sparse graphs typical of microservice architectures [13]. In resource optimization context, graph coloring addresses scheduling and isolation problems. Services that share dependencies or compete for the same resources (CPU cores, GPU accelerators, network bandwidth) are connected by conflict edges; coloring this conflict graph determines the minimum number of resource pools or time slots needed to avoid contention. This is directly applicable to batch job scheduling in Kubernetes, where CronJobs accessing the same databases should be assigned to different time windows to prevent resource contention [33]. Furthermore, graph coloring can optimize multi-tenant service placement: services from different tenants sharing physical nodes must be colored such that co-located services do not create security isolation violations or noisy-neighbor interference. Welsh–Powell algorithm, which colors vertices in decreasing order of degree, naturally prioritizes high-connectivity hub services that are most constrained, achieving practical results within a factor of 2 of optimal for microservice conflict graphs [34].

DAG scheduling for workflow orchestration. Directed Acyclic Graph (DAG) scheduling algorithms assign tasks represented as DAG vertices to heterogeneous processors, minimizing makespan (total completion time). The Heterogeneous Earliest Finish Time (HEFT) algorithm [35] is particularly relevant to microservice workflow orchestration, running in $O(V^2P)$ time where P is the number of available processors (nodes). HEFT operates in two phases: an upward ranking phase that computes priority based on critical path from each task to the exit node, and a processor selection phase that assigns each task (in priority order) to the processor minimizing its earliest finish time. In microservice context, HEFT directly models the problem of scheduling service replicas across heterogeneous nodes – services have different computational costs on different node types (CPU-optimized vs. memory-optimized instances), and communication costs between services depend on co-location. Kubernetes scheduler extensions implementing HEFT-inspired algorithms have demonstrated improved makespan for DAG-structured microservice workflows compared to default bin-packing scheduling [36]. The CPOP (Critical Path on a Processor) variant focuses critical-path tasks on a single fast processor, applicable when a bottleneck chain should be co-located on high-performance nodes. For dynamic environments where the DAG structure changes with traffic patterns, list scheduling heuristics provide $O(V \log V)$ amortized complexity suitable for online rescheduling [37].

Strongly connected component detection for circular dependency analysis. Strongly connected components (SCCs) – maximal subsets of vertices where every vertex is reachable from every other – identify circular dependency clusters in microservice architectures. Tarjan's algorithm computes all SCCs in $O(V + E)$ time using a single DFS pass, while Kosaraju's algorithm achieves the same complexity using two DFS passes [13]. Circular dependencies between microservices create several operational challenges: deadlock risk during coordinated scaling (service A waits for B to scale, B waits for A), cascade failure amplification (failure in any SCC member can propagate to all others), and deployment ordering impossibility (no valid topological order exists within an

SCC) [38]. Empirical analysis of production microservice graphs reveals that SCCs are more common than expected: Meta's 18,500-service graph contains multiple non-trivial SCCs arising from bidirectional health checks, callback mechanisms, and shared state services [1]. SCC detection enables three practical interventions: first, identifying circular dependencies that should be refactored into unidirectional flows using message queues or event buses; second, treating each SCC as a single scaling unit that must be scaled atomically; third, designing circuit breaker configurations that account for circular propagation paths. The condensation graph - obtained by contracting each SCC to a single vertex – is always a DAG and can be processed by topological sorting algorithms for deployment ordering of the remaining acyclic structure [39].

Load propagation model. A key advantage of the graph model is the ability to formally calculate how load propagates through the system. Let λ_i denote the total load on service i , λ_i^0 denote the external load on service i , and a_{ij} denote the average number of calls to service j generated by one request to service i .

Then the total load on each service is:

$$\lambda_j = \lambda_j^0 + \sum_i a_{ij} \cdot \lambda_i \quad (2)$$

In matrix form, this becomes:

$$\lambda = \lambda^0 + A^T \lambda \quad (3)$$

Solving for λ :

$$\lambda = (I - A^T)^{-1} \cdot \lambda^0 \quad (4)$$

where $A \in \mathbb{R}^{n \times n}$ is the fan-out matrix with $n = |V|$ (number of services), and element a_{ij} equals the fan-out factor from service i to service j if edge $(i, j) \in E$, and 0 otherwise. The vector $\lambda^0 \in \mathbb{R}^n$ contains external request rates (non-zero only for entry-point services such as API gateways), and $\lambda \in \mathbb{R}^n$ is the resulting total load vector.

Convergence conditions. The matrix inversion $(I - A^T)^{-1}$ in equation (4) exists if and only if the spectral radius $\rho(A) < 1$. For DAGs, this is automatically satisfied: after topological ordering, A is strictly upper triangular and nilpotent ($A^n = 0$), so $(I - A^T)^{-1} = I + A^T + (A^T)^2 + \dots + (A^T)^{n-1}$ converges in at most n steps [13]. For graphs with cycles (SCCs), convergence requires that the product of fan-out factors along every cycle is strictly less than 1 – intuitively, each traversal of the cycle must attenuate rather than amplify load. When $\rho(A) \geq 1$, the geometric series diverges, indicating an unstable feedback loop: the system would generate unbounded internal traffic. This is a mathematically rigorous signal of an architectural problem (positive feedback loop) requiring intervention — circuit breakers, rate limiters, or dependency refactoring [39].

Handling retries and error-driven traffic. The base fan-out factor a_{ij} represents successful single calls. In practice, retries amplify traffic. Let r_{ij} denote the retry rate on edge (i, j) .

The effective fan-out factor becomes:

$$a_{ij_eff} = a_{ij} \times (1 + r_{ij} + r_{ij}^2 + \dots) = a_{ij} / (1 - r_{ij}) \text{ for } r_{ij} < 1. \quad (5)$$

Substituting a_{ij_eff} into A yields an effective fan-out matrix A_{eff} that accounts for retry amplification. This can increase $\rho(A_{eff})$ beyond 1 even when $\rho(A) < 1$, explaining how retry storms destabilize microservice systems [11].

Circuit breaker effects. When a circuit breaker on edge (i, j) activates (open state), the effective fan-out drops to $a_{ij_eff} = 0$, removing that edge from the propagation model. The system therefore has multiple operational modes, each with a different effective matrix a_{mode} . The load propagation model applies within each mode; transitions between modes are triggered by latency or error rate thresholds and can be analyzed as a piecewise-linear system.

Asynchronous communication. For asynchronous edges (message queues, event buses), the steady-state load equation (4) remains valid: in equilibrium, consumer throughput must equal producer throughput regardless of temporal decoupling. However, the model does not capture transient behavior – queue depth buildup during traffic spikes, consumer lag, or backpressure

effects. For transient analysis, the model should be extended with queue capacity constraints and consumption rate parameters, which is beyond the scope of this survey.

Dynamic topology. The fan-out matrix A changes over time due to deployments, feature flag changes, canary routing, and circuit breaker activations. Equation (4) provides a snapshot estimate valid for the current topology. In practice, $A(t)$ should be estimated using a sliding window over distributed traces (e.g., 5-minute window), and equation (4) recomputed periodically. Morphis [12] demonstrates that production call graphs at ByteDance concentrate around a small set of recurring patterns, suggesting that $A(t)$ is approximately piecewise-constant, making periodic recomputation practical.

Validation considerations. The load propagation model is an analytical result derived from the linear system of equations (2); its correctness follows from the mathematical properties of geometric matrix series [13]. Empirical consistency is supported by production observations: Uber reports 10–15 $\times$ traffic amplification in microservice call chains [2], and Alibaba traces show similar multiplicative effects [4], both consistent with the model's predictions. Direct validation against production traces (e.g., comparing equation (4) predictions with measured per-service request rates from Alibaba traces [4] or DeathStarBench [29]) constitutes an important direction for future experimental work.

This means that, knowing external traffic and edge weights (fan-out factors), we can estimate the real load on every service, including all downstream propagation effects. Consider the graph in Fig. 2 with frontend receiving 500 external requests per second. The adjacency matrix A encodes: checkout receives 1.0 calls per frontend request, payment receives 1.0 calls per checkout request, inventory receives 2.0 calls per checkout request, shipping receives 0.5 calls per checkout request, and fraud-check receives 1.2 calls per payment request. The results of applying equation (4) are presented in Table 1.

Table 1. Load propagation calculation for the example microservice graph

Service	External RPS	Propagated RPS	Fan-out source
Frontend	500	500	External
Checkout	0	500	$1.0 \times \text{frontend}$
Payment	0	500	$1.0 \times \text{checkout}$
Inventory	0	1,000	$2.0 \times \text{checkout}$
Shipping	0	250	$0.5 \times \text{checkout}$
Fraud-check	0	600	$1.2 \times \text{payment}$

Source: compiled by the authors

This illustrative example demonstrates that a single external stream of 500 requests per second generates 2,850 total internal requests per second – a 5.7 $\times$ amplification factor. While these specific numbers are derived from the synthetic graph in Fig. 2 rather than production measurements, they are consistent with empirical observations: Uber reports 10–15 $\times$ amplification in their microservice call chains [2], and Alibaba traces show similar multiplicative effects across dependency subgraphs [4]. Without a graph model, one might allocate resources for checkout at 500 requests per second, ignoring that inventory requires capacity for 1,000 and fraud-check for 600. This demonstrates why per-service optimization is fundamentally insufficient: the resource decision for any service cannot be made locally but must account for the entire dependent subgraph [7].

The load propagation model also enables what-if analysis: if frontend traffic grows by 30%, equation (4) immediately yields the new load on every downstream service, enabling proactive scaling of the entire dependency chain before bottlenecks emerge. This is the mathematical foundation for coordinated scaling strategies such as Chamulteon [6] and pHPA [10], which monitor traffic and performance metrics across the entire call chain.

Graph neural networks for adaptive optimization. Classical graph algorithms provide static, offline solutions. Modern Graph Neural Network (GNN) approaches learn adaptive policies that respond to dynamic workloads, capture complex non-linear relationships between services, and optimize objectives difficult to express analytically [7]. GNNs learn node and edge representations

by iterative message passing: each node aggregates feature information from its neighbors, building representations that encode both local attributes and structural context, with $O(L \cdot (V + E) \cdot d^2)$ time per forward pass (L layers, d feature dimensions).

Several recent systems demonstrate the effectiveness of GNN-based resource optimization. GRAF [22] uses a GNN to model service dependencies for SLO-aware proactive resource allocation, achieving 14-19% CPU savings compared to Kubernetes HPA and 2.6× faster tail-latency convergence during traffic surges. DeepScaler [23] employs a spatio-temporal GNN with adaptive graph learning using expectation-maximization to discover implicit service dependencies, reducing SLA violations by 41% at lower cost. HGraphScale [24] introduces a hierarchical three-layer GNN (physical machine → virtual machine → container) capturing both service topology and infrastructure hierarchy, achieving up to 80.16 % reduction in average response time.

Despite promising results, GNN-based approaches have several important limitations that must be acknowledged. First, regarding benchmark representativeness: GRAF was evaluated on the Alibaba cluster trace ($\approx 4,000$ services) and a custom testbed, DeepScaler on the Train-Ticket benchmark (41 services), and HGraphScale on a simulated three-tier environment - none of these fully represent the scale and complexity of production systems at Meta (18,500+ services) or Alibaba (20,000+ services in production) [1], [4]. The reported improvement ranges are therefore not directly comparable: GRAF's 14-19 % refers to CPU cost savings, DeepScaler's 41 % to SLA violation count reduction, and HGraphScale's 80 % to average response time reduction - fundamentally different metrics measured on different systems at different scales. Second, training scalability remains a challenge: GNN training complexity $O(L \cdot (V + E) \cdot d^2)$ per epoch makes training on graphs with more than 10^4 vertices computationally expensive, and none of the surveyed systems have demonstrated training on production-scale graphs [7]. Third, interpretability is limited compared to classical algorithms: while GNN models learn effective policies, explaining why a particular scaling decision was made is significantly harder than tracing the logic of a CPM or centrality calculation – a concern for production systems requiring auditability. Fourth, cold-start and concept drift: GNN models require historical trace data for training, making them ineffective for newly deployed services or after major architectural changes until sufficient data is collected [40]. No surveyed GNN system has been reported as deployed in production at the scale of Meta, Google, or Alibaba; existing deployments are limited to controlled environments and academic testbeds.

A particularly significant trend is the combination of classical graph algorithms with GNN models. PERT-GNN [15] constructs PERT graphs from distributed traces and applies Graph Transformer for end-to-end latency prediction, demonstrating that the structured analytical power of classical algorithms enhances GNN performance. PBScaler [25] uses TopoRank - a random walk algorithm based on topological potential theory – for bottleneck identification before applying genetic algorithm optimization, achieving the lowest mean response time with minimal resource consumption. TPRAM [26] applies Transformer attention mechanisms to model microservice dependencies for critical-path-aware resource adjustment. Table 2 summarizes the key GNN-based systems for microservice resource optimization.

Table 2. GNN-based systems for microservice resource optimization

System	Graph model	Method	Key result	Year
GRAF	Service dep. + GNN	Optimization	19% CPU savings	2024
PERT-GNN	PERT + Transformer	Supervised	Latency prediction	2023
DeepScaler	ST-GNN	Adaptive RL	41% fewer SLA viol.	2023
HGraphScale	Hierarchical GNN	DRL	80% latency reduction	2025
PBScaler	Topo. potential	GA	Min. response time	2024
TPRAM	Transformer	DDPG	CP-aware resources	2025

Source: compiled by the authors

Comparative analysis. Table 3 provides a comprehensive comparison of the surveyed algorithm categories across key dimensions relevant to microservice resource optimization. The evaluation considers time complexity, support for directed and weighted graphs (essential for modeling microservice dependencies), scalability to production-sized systems, and the specific optimization task addressed.

Table 3. Comparative analysis of graph algorithms for microservice optimization

Category	Complexity	Dir.	Wt.	Relevance
Topological sort	$O(V+E)$	Yes	No	HIGH
CPM / PERT	$O(V+E)$	Yes	Yes	HIGH
Dijkstra / Bellman–Ford	$O((V+E)\lg V)$	Yes	Yes	MEDIUM
Network flow	$O(VE^2)–O(V^3)$	Yes	Yes	MEDIUM
Betweenness centrality	$O(VE)$	Yes	Yes	HIGH
PageRank	$O(k(V+E))$	Yes	Yes	HIGH
Louvain / LPA	$O(n \lg n)$	No	Yes	HIGH
METIS partitioning	$O(V+E)$	No	Yes	HIGH
Graph coloring	$O(V+E)–O(V^2)$	No	No	MEDIUM
DAG scheduling (HEFT)	$O(V^2P)$	Yes	Yes	HIGH
GNN + RL	$O(L(V+E)d^2)$	Yes	Yes	V. HIGH

Source: compiled by the authors

To substantiate the novelty of the proposed classification, Table 4 compares it with the two most relevant prior surveys: Nguyen et al. [7], which focuses on GNN applications for microservice-based cloud systems, and Waseem et al. [43], which provides a systematic mapping of microservices in DevOps. Neither prior survey offers a unified taxonomy of graph algorithm categories for resource optimization, nor do they provide a load propagation model or comparative complexity analysis.

Table 4. Comparison with prior survey studies

Dimension	Nguyen et al. [7]	Waseem et al. [43]	This survey
Scope	GNN for microservice cloud applications	Microservices in DevOps practices	Graph algorithms for resource optimization
Algorithm taxonomy	3 GNN architecture types (GCN, GAT, GraphSAGE)	No algorithm classification	11 classical categories + GNN-based
Classical graph algorithms	Not covered	Not covered	Systematic analysis with complexity
Load propagation model	Not provided	Not provided	Matrix-based model (Eq. 4)
Complexity comparison	Partial (GNN only)	None	Full (Table 3)
Hybrid classical+GNN analysis	Mentioned as future work	Not addressed	Analyzed as key trend
Applicability mapping	Per GNN system	Per DevOps practice	Per algorithm category × task
Number of sources	52 (all GNN-focused)	47 (DevOps-focused)	43 (graph optimization-focused)
Resource optimization focus	Partial (one of several applications)	Not primary focus	Primary focus

Source: compiled by the authors

Table 5 maps each analyzed source to its algorithm category, optimization task, graph type, evaluation metrics, and level of experimental evidence. This mapping demonstrates the coverage of the proposed classification and enables identification of under-explored combinations.

Table 5. Mapping of surveyed works to the proposed classification

Source	Algorithm category	Optimization task	Graph type	Key metrics	Evidence level
[1] Huye et al. (Meta)	Topology analysis	Characterization	Directed, dynamic	Degree, path length	Production (18,500 svc)
[2] Zhang et al. (CRISP)	CPM	Bottleneck identification	DAG (traces)	Latency, cost	Production (Uber)
[4] Luo et al. (Alibaba)	Community detection, topology	Characterization	Directed, weighted	Clustering coeff., call freq.	Production (>20,000 svc)
[6] Bauer et al. (Chamulteon)	Network flow	Coordinated scaling	Directed, weighted	Latency, throughput	Testbed (14 services)
[9] Rzacca et al. (Autopilot)	DAG scheduling	Autoscaling	Directed	CPU, memory	Production (Google Borg)
[10] Choi et al. (pHPA)	CPM	Proactive scaling	DAG (chain)	Latency, SLO	Testbed (microservice chain)
[11] Soldani et al.	SCC detection	Failure analysis	Directed, cyclic	Cascade depth	Case study
[12] Tang et al. (Morphis)	Topology, scheduling	SLO-aware scheduling	Dynamic, weighted	SLO compliance, cost	Production (ByteDance)
[15] Tam et al. (PERT-GNN)	CPM + GNN	Latency prediction	DAG (PERT)	E2E latency (MAE)	Alibaba traces
[19] Yu et al. (MicroRank)	Centrality (PageRank)	Root cause localization	Directed, weighted	Localization accuracy	Production traces
[22] Park et al. (GRAF)	GNN	SLO-aware scaling	Directed, weighted	CPU cost, tail latency	Alibaba traces + testbed
[23] Meng et al. (DeepScaler)	GNN (spatio-temporal)	Holistic autoscaling	Adaptive (learned)	SLA violations, cost	Train-Ticket (41 svc)
[24] Fang et al. (HGraphScale)	GNN (hierarchical)	Autoscaling	3-layer hierarchical	Response time	Simulated 3-tier
[25] Xie et al. (PBScaler)	Centrality + genetic alg.	Bottleneck-aware scaling	Directed, weighted	Response time, resources	Sock-Shop, Train-Ticket
[26] Chen et al. (TPRAM)	Transformer (attention)	Proactive resource alloc.	Directed	Prediction accuracy	Sock-Shop
[27] Luo et al. (ERMS)	Network flow, Scheduling	Shared resource Mgmt.	Directed, Weighted	SLA viol. prob., cost	Alibaba traces
[29] Gan et al. (DeathStarBench)	Benchmark (multiple)	Benchmark suite	Directed, weighted	Latency, throughput	Open-source benchmark
[30] Gan et al. (Seer)	Shortest path + DL	Performance debugging	Directed (traces)	RCA accuracy	DeathStarBench
[40] Qiu et al. (FIRM)	Centrality, scheduling	Fine-grained resource mgmt.	Directed	SLO, resource usage	Testbed (10 services)

Source: compiled by the authors

The analysis reveals a clear stratification. Linear-time algorithms (topological sort, CPM, LPA, METIS) scale to graphs with 10^7+ vertices, making them suitable for even the largest production deployments. Near-linear algorithms (Louvain, Dijkstra, minimum spanning trees) handle up to 10^6

vertices. Quadratic algorithms (betweenness centrality, graph coloring heuristics) remain practical for up to 10^4 vertices, covering most enterprise systems. GNN-based approaches currently scale to 10^3 – 10^4 vertices for training, but inference can handle larger graphs [7].

A significant finding is that no single algorithm category addresses all optimization tasks. Effective dependency-aware resource optimization requires a pipeline combining multiple algorithms: topological sorting for deployment ordering, CPM for bottleneck identification, centrality metrics for priority ranking, community detection for clustering, and graph partitioning for placement. Modern GNN-based systems increasingly adopt this combined approach: PERT-GNN combines CPM with Graph Transformer [15], PBScaler combines topological potential with genetic algorithms [25], and GRAF uses GNN-encoded graph structure for SLO-aware optimization [22].

The scalability of algorithms must be considered against the empirical properties of production microservice graphs. Meta’s analysis [1] established that these graphs are sparse, modular, and – importantly – not scale-free. The moderate heterogeneity with some high-degree hub services, typical depth of 2-6, and branching factor of 2-14 mean that even quadratic algorithms like betweenness centrality remain practical for the vast majority of real-world deployments.

Results

Findings from the reviewed literature. The load propagation model (equation (4)) demonstrates that inter-service dependencies create significant traffic amplification: the illustrative example yields a $5.7\times$ factor, consistent with production observations at Uber ($10 - 15\times$ [2]) and Alibaba [4]. The comparative analysis (Table 3) reveals a clear scalability stratification: linear-time algorithms (topological sort, CPM, METIS) handle graphs with up to 10^7 vertices, near-linear algorithms (Dijkstra, Louvain) up to 10^6 , quadratic algorithms (betweenness centrality) up to 10^4 , and GNN-based approaches $10^3 - 10^4$ for training. Among GNN systems evaluated in their respective environments, GRAF [22] reports 14-9 % CPU savings on Alibaba cluster traces, DeepScaler [23] reports 41 % SLA violation reduction on the Train-Ticket benchmark (41 services), and HGraphScale [24] reports up to 80.16% response time reduction in a simulated three-tier environment. ERMS [27] reports $5\times$ reduction in SLA violation probability and $1.6\times$ less resource usage compared to independent per-service optimization on Alibaba traces.

Synthesis and recommendations. Based on the analysis of the surveyed literature, we identify the following patterns. First, no single algorithm category addresses all optimization tasks – effective dependency-aware resource optimization requires combining multiple algorithms. Second, the most promising systems in the reviewed literature combine classical graph algorithms for structural analysis with GNN-based methods for dynamic adaptation (e.g., PERT-GNN [15], PBScaler [25], GRAF [22]). Third, coordinated scaling strategies that account for inter-service dependencies consistently outperform per-service approaches in the studies where they were evaluated.

Important caveats. The improvement figures cited above were obtained on different benchmarks (Alibaba traces, Train-Ticket, DeathStarBench, custom testbeds), at different scales (41 to $\sim 4,000$ services), and using different metrics (CPU cost, SLA violations, response time). Direct quantitative comparison across these systems is therefore not possible without a common evaluation framework. The recommendation of a hybrid classical+GNN pipeline is an analytical synthesis from the surveyed literature, not an empirically validated result on a single benchmark.

Conclusions. This paper has presented a comprehensive survey of graph-theoretic algorithms and approaches for dependency-aware resource optimization in microservice architectures. The key findings are as follows.

First, the graph model $G = (V, E, W)$ provides an optimal representation for microservice resource optimization because the mapping between the real system (services, dependencies, traffic flows, and constraints) and the mathematical model (vertices, edges, weights, and constraints) is

nearly direct. This minimal representational gap enables the application of well-established graph algorithms with known complexity guarantees and scalability properties.

Second, the load propagation model based on adjacency matrix inversion (equation (4)) indicates that local per-service resource decisions may be significantly suboptimal when inter-service dependencies create substantial traffic amplification. The traffic amplification effect - where 500 external requests per second generate 2,850 internal requests per second (illustrated by the synthetic example in “Load propagation model” section / equation (4), consistent with production amplification ratios reported by Uber [2] and Alibaba [4]) across the dependency subgraph - makes graph-based analysis essential for accurate capacity planning and scaling.

Third, a promising direction identified in the reviewed literature is the combination of classical graph algorithms for structural analysis with GNN-based methods for dynamic adaptation. Systems such as PERT-GNN [15], GRAF [22], and PBScaler [25] suggest that this hybrid approach can improve upon traditional per-service methods. However, the reported improvements – 14–19% CPU cost savings (GRAF, Alibaba traces), 41% SLA violation reduction (DeepScaler, Train-Ticket benchmark), and up to 80 % response time reduction (HGraphScale, simulated environment) — were obtained on different systems, benchmarks, and metrics. Consequently, these figures should not be aggregated or interpreted as evidence for the effectiveness of a single hybrid pipeline. Validating the proposed hybrid approach on a common benchmark remains an important open challenge.

Fourth, coordinated scaling strategies – exemplified by Chamulteon [6], ERMS [27], and Uber's CRISP [2] – provide evidence that dependency-aware resource management can yield significant improvements in the evaluated settings. For example, ERMS [27] reports a 5× reduction in SLA violation probability and 1.6× reduction in resource usage compared to independent per-service optimization on Alibaba cluster traces. Whether these improvements generalize across different workload characteristics, system scales, and infrastructure configurations remains to be established.

Future work should address several open challenges: the cold-start problem in scaling (where JVM warm-up consumes 2–3× CPU), observability gaps in dependency detection (incomplete tracing, metric propagation delays), multi-cluster and multi-region dependency graphs, and the integration of graph-based optimization with emerging infrastructure paradigms such as serverless computing and edge-cloud architectures. The development of standardized benchmarks for evaluating dependency-aware resource optimization algorithms – like DeathStarBench for microservice workloads – would significantly advance the field.

Limitations. This survey has several limitations that should be considered when interpreting its findings. First, as a literature survey rather than an experimental study, the comparative analysis (Table 3) is based on reported results from heterogeneous evaluation environments; we did not independently reproduce or validate any of the cited experimental results. Second, the literature search, while systematic, was limited to four databases and English-language publications, potentially missing relevant work in other venues or languages. Third, the load propagation model (equation (4)) assumes steady-state conditions and linear fan-out relationships; real systems exhibit non-linear behavior under saturation, bursty traffic patterns, and adaptive routing that the model does not capture. Fourth, the proposed hybrid pipeline (classical algorithms + GNN) is an analytical recommendation based on observed trends in the literature, not a validated system; its practical effectiveness, implementation complexity, and operational overhead remain to be demonstrated. Fifth, the mapping of algorithms to optimization tasks (Table 5) reflects applicability as described in the surveyed papers; algorithms not yet applied to a particular task are marked as not applicable, which may understate their potential.

Conflict of interests: Authors declare that they do not have conflict of interests, in particular financial, personal, copyright or any of a different nature, which could have influenced the research, as well as results published in this article

Financing: Research was carried out without financial support

Accessibility data: All data are available in digital or graphic form in the main text of the manuscript

Using artificial intelligence tools: During preparation manuscript used tool Claude Opus 4.6 of artificial intelligence (AI) for grammatical and stylistic text checks. All fragments processed by him were checked and edited by the author. AI was not used to generate text, formulas, and tables, formation scientific provisions, receipt results or formulation conclusions. Scientific the provisions, results and conclusions are my own, intellectual by the contribution of the authors

REFERENCES

1. Huye, D., Shkuro, Y. & Sambasivan, R. R. “Lifting the veil on Meta’s microservice architecture: analyses of topology and request workflows”. *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023. p. 431–446. – Available from: <https://www.usenix.org/conference/atc23/presentation/huye>. – [Accessed: 21.09.25].
2. Zhang, Z., et al. “CRISP: critical path analysis of large-scale microservice architectures”. *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022. p. 665–678. – Available from: <https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou>. – [Accessed: 21.09.25].
3. Bryant, D. “Adopting microservices at Netflix: lessons for architectural design”. *NGINX (F5) Blog*. 2015. – Available from: <https://www.f5.com/company/blog/nginx/microservices-at-netflix-architectural-best-practices>. – [Accessed: 21.09.25].
4. Luo, S. et al. “Characterizing microservice dependency and performance: Alibaba trace analysis”. *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2021. p. 412–426. DOI: <https://doi.org/10.1145/3472883.3487003>.
5. “Horizontal Pod Autoscaling”. *Kubernetes Documentation*. 2024. – Available from: <https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/>. – [Accessed: 21.09.25].
6. Bauer, A., et al. “Chamulteon: coordinated auto-scaling of micro-services”. *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019. p. 2015–2025. DOI: <https://doi.org/10.1109/ICDCS.2019.00199>.
7. Nguyen, H. X., Zhu, S. & Liu, M. “A survey on graph neural networks for microservice-based cloud applications”. *Sensors*. 2022; 22 (23): 9492. DOI: <https://doi.org/10.3390/s22239492>.
8. Ma, S.-P., Ding, C.-Y. & Shih, M.-J. “Using service dependency graph to analyze and test microservices”. *Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. 2018. p 58–64. DOI: <https://doi.org/10.1109/COMPSAC.2018.10207>.
9. Rzacca, K., et al. “Autopilot: workload autoscaling at Google scale”. *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*. 2020. p. 1–16. DOI: <https://doi.org/10.1145/3342195.3387524>.
10. Choi, B., Yoon, J., Lim, J., Chung, I. & Jeon, M. “pHPA: A proactive autoscaling framework for microservice chain”. *Proceedings of the 5th Asia-Pacific Workshop on Networking (APNet)*. 2021. p. 8–14. – Available from: <https://conferences.sigcomm.org/events/apnet2021/papers/apnet2021-8.pdf>. – [Accessed: 21.09.25].
11. Soldani, J., Tamburri, D. A. & Van Den Heuvel, W.-J. “Explaining microservices’ cascading failures from their logs”. *Software: Practice and Experience*. 2025; 55 (5): 809–828. DOI: <https://doi.org/10.1002/spe.3400>.
12. Tang, Y., et al. “Morphis: SLO-aware resource scheduling for microservices with time-varying call graphs”. *arXiv*. 2026. DOI: <https://doi.org/10.48550/arXiv.2602.01044>.
13. Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. “Introduction to algorithms”. 4th ed. Cambridge: MIT Press. 2022. – Available from: <https://www.cs.mcgill.ca/~akroitt/math/compsci/Cormen%20Introduction%20to%20Algorithms.pdf>. – [Accessed: 21.09.25].
14. Kelley, J. E. & Walker, M. R. “Critical-path planning and scheduling”. *Proceedings of the Eastern Joint IRE-AIEE-ACM Computer Conference*. 1959. p. 160–173. DOI: <https://doi.org/10.1145/1460299.1460318>.
15. Tam, D. S. H., Liu, Y., Xu, H., Xie, S. & Lau, W. C. “PERT-GNN: latency prediction for microservice-based cloud-native applications via graph neural networks”. *Proceedings of the 29th*

ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 2023. p. 2155–2165. DOI: <https://doi.org/10.1145/3580305.3599465>.

16. Ford, L. R. & Fulkerson, D. R. “Maximal flow through a network”. *Canadian Journal of Mathematics*. 1956; 8: 399–404. DOI: <https://doi.org/10.4153/CJM-1956-045-5>.

17. Brandes, U. “A faster algorithm for betweenness centrality”. *Journal of Mathematical Sociology*. 2001; 25 (2): 163–177. DOI: <https://doi.org/10.1080/0022250X.2001.9990249>.

18. Page, L., Brin, S., Motwani, R. & Winograd, T. “The PageRank citation ranking: bringing order to the web”. *Stanford InfoLab*. 1999. – Available from: <https://web.archive.org/web/20241231011600/http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf>. – [Accessed: 21.09.25].

19. Yu, G. et al. “MicroRank: end-to-end latency issue localization with extended spectrum analysis in microservice environments”. *Proceedings of the Web Conference (WWW)*. 2021. p. 3087–3098. DOI: <https://doi.org/10.1145/3442381.3449905>.

20. Blondel, V. D., Guillaume, J.-L., Lambiotte, R. & Lefebvre, E. “Fast unfolding of communities in large networks”. *Journal of Statistical Mechanics: Theory and Experiment*. 2008. p. P10008. DOI: <https://doi.org/10.1088/1742-5468/2008/10/P10008>.

21. Karypis, G. & Kumar, V. “A fast and high quality multilevel scheme for partitioning irregular graphs”. *SIAM Journal on Scientific Computing*. 1998; 20 (1): 359–392. DOI: <https://doi.org/10.1137/S1064827595287997>.

22. Park, J., Choi, B., Lee, C. & Han, D. “Graph neural network-based SLO-aware proactive resource autoscaling framework for microservices”. *IEEE/ACM Transactions on Networking*. 2024; 32 (4): 3331–3346. DOI: <https://doi.org/10.1109/TNET.2024.3393427>.

23. Meng, C., Song, S., Tong, H., Pan, M. & Yu, Y. “DeepScaler: Holistic autoscaling for microservices based on spatiotemporal GNN with adaptive graph learning”. *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023. p. 53–65. DOI: <https://doi.org/10.1109/ASE56229.2023.00038>.

24. Fang, Z., Ma, H., Chen, G. & Buyya, R. “HGraphScale: hierarchical graph learning for autoscaling microservice applications in container-based cloud computing”. 2025. – DOI: <https://doi.org/10.48550/arXiv.2511.01881>.

25. Xie, S., Wang, J., Li, B., Zhang, Z. & Hung, P. C. K. “PBScaler: a bottleneck-aware autoscaling framework for microservice-based applications”. *IEEE Transactions on Services Computing*. 2024; 17 (2): 604–616. DOI: <https://doi.org/10.1109/TSC.2024.3376202>.

26. Chen, Y., Hao, J., Peng, Y. & Xia, H. “Transformer-based performance prediction and proactive resource allocation for cloud-native microservices”. *Cluster Computing*. 2025; 28 (9). DOI: <https://doi.org/10.1007/s10586-025-05237-9>.

27. Luo, S., et al. “ERMS: efficient resource management for shared microservices with SLA guarantees”. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2023. p. 62–77. DOI: <https://doi.org/10.1145/3567955.3567964>.

28. Yen, J. Y. “Finding the K shortest loopless paths in a network”. *Management Science*. 1971; 17 (11): 712–716. DOI: <https://doi.org/10.1287/mnsc.17.11.712>.

29. Gan, Y., et al. “An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems”. *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019. p. 3–18. DOI: <https://doi.org/10.1145/3297858.3304013>.

30. Gan, Y., Pancholy, M., Cheng, Y., Kedia, R. & Delimitrou, C. “Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices”. *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019. p. 19–33. DOI: <https://doi.org/10.1145/3297858.3304004>.

31. Burns, B., Grant, B., Oppenheimer, D., Brewer, E. & Wilkes, J. “Borg, Omega, and Kubernetes: lessons learned from three container-management systems over a decade”. *ACM Queue*. 2016; 14 (1): 70–93. DOI: <https://doi.org/10.1145/2898442.2898444>.

32. Tian, C., et al. “Characterizing and synthesizing task dependencies of data-parallel jobs in Alibaba cloud”. *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2019. p. 139–151. DOI: <https://doi.org/10.1145/3357223.3362711>.
33. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. & Wilkes, J. “Large-scale cluster management at Google with Borg”. *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. 2015. p. 1–17. DOI: <https://doi.org/10.1145/2741948.2741964>.
34. Welsh, D. J. A. & Powell, M. B. “An upper bound for the chromatic number of a graph and its application to timetabling problems”. *The Computer Journal*. 1967; 10 (1): 85–86. DOI: <https://doi.org/10.1093/comjnl/10.1.85>.
35. Topcuoglu, H., Hariri, S. & Wu, M.-Y. “Performance-effective and low-complexity task scheduling for heterogeneous computing”. *IEEE Transactions on Parallel and Distributed Systems*. 2002; 13 (3): 260–274. DOI: <https://doi.org/10.1109/71.993206>.
36. Boutin, E., et al. “Apollo: Scalable and coordinated scheduling for cloud-scale computing”. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014. p. 285–300. – Available from: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin>. — [Accessed: 21.09.25].
37. Grandl, R. et al. “Multi-resource packing for cluster schedulers”. *Proceedings of the ACM SIGCOMM Conference*. 2014. p. 455–466. DOI: <https://doi.org/10.1145/2619239.2626334>.
38. Taibi, D., Lenarduzzi, V. & Pahl, C. “Microservices anti-patterns: a taxonomy”. *Proceedings of the International Conference on Software Architecture Companion (ICSA-C)*. 2020. p. 98–105. DOI: <https://doi.org/10.1109/ICSA-C50368.2020.00024>.
39. Sharir, M. “A strong-connectivity algorithm and its applications in data flow analysis”. *Computers & Mathematics with Applications*. 1981; 7 (1): 67–72. DOI: [https://doi.org/10.1016/0898-1221\(81\)90008-0](https://doi.org/10.1016/0898-1221(81)90008-0).
40. Qiu, H., et al. “FIRM: an intelligent fine-grained resource management framework for SLO-oriented microservices”. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020. p. 805–825. – Available from: <https://www.usenix.org/conference/osdi20/presentation/qiu>. — [Accessed: 21.09.25].
41. Zhou, X., et al. “Fault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study”. *IEEE Transactions on Software Engineering*. 2021; 47 (12): 2675–2702. DOI: <https://doi.org/10.1109/TSE.2018.2887384>.
42. Zhou, X., et al. “Delta debugging microservice systems with parallel optimization”. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2018. p. 338–348. DOI: <https://doi.org/10.1145/3238147.3240730>.
43. Waseem, M., Liang, P. & Shahin, M. “A systematic mapping study on microservices architecture in DevOps”. *Journal of Systems and Software*. 2020; 170: 110798. DOI: <https://doi.org/10.1016/j.jss.2020.110798>.

Received 30.09.2026

Received after revision 05.10.2026

Accepted 12.10.2026

DOI: <https://doi.org/10.15276/ict.03.2026.68>

УДК 004.75:519.17

Графова оптимізація ресурсів з урахуванням залежностей між мікросервісами: огляд алгоритмів та підходів

Лисевич Євген Ігорович¹⁾

ORCID: <https://orcid.org/0009-0004-6161-9453>; lisevich.e.i@gmail.com

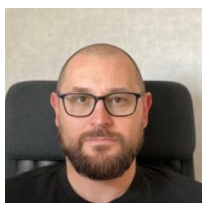
¹⁾ Одеський національний університет імені І. І. Мечникова, Всеволода Змієнка, 2. Одеса, 65082

АНОТАЦІЯ

Сучасні хмарні системи, побудовані на мікросервісній архітектурі, створюють фундаментальну проблему оптимізації ресурсів: окремі сервіси не є незалежними, а утворюють складні мережі залежностей, де рішення щодо масштабування, розміщення та ємності одного сервісу безпосередньо впливають на продуктивність і споживання ресурсів іншими. У цій статті представлено комплексний огляд алгоритмів та підходів теорії графів, застосовних до оптимізації ресурсів з урахуванням залежностей у мікросервісних системах. Систему мікросервісів формалізовано як зв'язаний орієнтований граф, у якому вершини відповідають сервісам та інфраструктурним компонентам, ребра – викликам сервісів і потокам даних, а ваги відповідають метрикам: частоті запитів, латентності, частоті повторних спроб та витратам на мережу. Систематично проаналізовано одинадцять категорій графових алгоритмів - включно з топологічним сортуванням, методом критичного шляху, мережевим потоком, алгоритмами найкоротшого шляху, метриками центральності (betweenness, PageRank), виявленням спільнот (Louvain), секціонуванням графів (бібліотека METIS), мінімальними кістяковими деревами, розфарбуванням графів, плануванням у направлених ациклічних графах (гетерогенний найкращий час закінчення) та виявленням сильно зв'язних компонент – з оцінкою часової та просторової складності, масштабованості й застосовності до задач оптимізації ресурсів мікросервісів. Представлено модель розповсюдження навантаження на основі обернення матриці суміжності з обговоренням умов збіжності для циклічних графів залежностей. Крім того, розглянуто сучасні підходи на основі графових нейронних мереж у поєднанні з навчанням з підкріпленням для адаптивного автомасштабування, зокрема системи Графові мережі з урахуванням механізму уваги та злиття даних, техніка оцінки та аналізу проєкту - графова нейронна мережа, DeepScaler та HGraphScale, які демонструють покращення: до двадцяти процентів економії обчислювальних потужностей (Графові мережі з урахуванням механізму уваги та злиття даних), до сорока відсотків зменшення порушень гарантій рівня сервісу (DeepScaler) та до восьмидесяти відсотків зменшення часу відповіді (HGraphScale) порівняно з традиційними підходами. Порівняльний аналіз виявляє, що поєднання класичних графових алгоритмів для структурного аналізу з методами на основі графових нейронних мереж для динамічної адаптації є найперспективнішим напрямком для виробничої оптимізації ресурсів з урахуванням залежностей.

Ключові слова: мікросервісна архітектура; теорія графів; оптимізація ресурсів; граф залежностей; метод критичного шляху; мережевий потік; графові нейронні мережі; автомасштабування; метрики центральності; Kubernetes

ABOUT THE AUTHOR



Yevhen I. Lysevych - Graduate student, Mechanics, Automation and Information Technologies Department. Odesa I.I. Mechnikov National University, 2, Vsevoloda Zmiiienka St. Odesa, 65082
ORCID: <https://orcid.org/0009-0004-6161-9453>; lisevich.e.i@gmail.com

Research areas: computer science

Лисевич Євген Ігорович – аспірант кафедри Механіки, автоматизації та інформаційних технологій. Одеський національний університет імені І. І. Мечникова, Всеволода Змієнка, 2, Одеса, 65082, Україна

Галузь досліджень: комп'ютерні науки