Network Aware Scheduler for Kubernetes

Yongyu Liu
University of Illinois Urbana-Champaign
yl201@illinois.edu

Yifei Zhang
University of Illinois Urbana-Champaign
yifei35@illinois.edu

Abstract

In cloud-native environments, the default Kubernetes scheduler optimizes pod placement based primarily on CPU and memory availability, while largely overlooking network conditions. However, for latency-sensitive microservices and bandwidth-intensive workloads, network performance can be a critical bottleneck. In this work, we propose a network-aware scheduling plugin that enhances Kubernetes scheduling decisions by factoring in live network state during the scoring phase. We implement the plugin using the official Kubernetes Scheduler Framework and collect real network metrics using lightweight agents deployed alongside nodes. To evaluate our approach, we conduct controlled experiments in a multi-node cluster, emulating bandwidth constraints and heterogeneous latency topologies.Experiments show that default scheduling results in almost random task location, while our plugin efficiently groups related tasks together when there are latency or bandwidth differences between nodes. This leads to modest enhancements in average latency and significant decreases in tail latency. This approach provides a lightweight, practical enhancement for network-sensitive workloads in Kubernetes.

1 Introduction

Kubernetes[8] is a widely adopted container orchestration system that automates the deployment, scaling, and management of containerized applications. At its core, the Kubernetes scheduler[10] is responsible for assigning newly created pods (the smallest deployable units in Kubernetes) to suitable worker nodes based on resource availability. The default scheduler evaluates a combination of filtering and scoring policies, considering node-level attributes such as CPU, memory, taints, and affinity rules. However, the scheduler does not natively take network performance metrics into account.

While the default scheduler is sufficient for compute- and memory-bound workloads, it may fall short for modern cloud-native applications where network behavior significantly affects overall performance. Many real-world applications, such as multi-tier web services, IoT pipelines, video streaming, and distributed databases, involve multiple microservices communicating intensively. In such scenarios, end-to-end latency and available bandwidth often become primary performance bottlenecks.

Motivated by these limitations, we explore whether real-time network awareness can improve Kubernetes pod placement, especially for latency- and bandwidth-sensitive applications. Prior work in service function chaining and microservice systems highlights the importance of minimizing latency and ensuring sufficient bandwidth to maintain Quality of Service [1].

To this end, we present a network-aware scheduler plugin for Kubernetes. Our system collects and aggregates real-time network metrics and integrates them into the scheduling process via the `Score` phase. Experiments show that our approach enables more informed placement decisions, reducing pod-to-pod RTT for latency-sensitive services, improving throughput for bandwidth-heavy workloads, and lowering loss for accuracy-critical communications—all with minimal overhead from the probing infrastructure.

In general, we make the following contributions.

- We develop a comprehensive framework for automatically collecting and aggregating real-time network metrics, including latency, bandwidth, and packet loss. The aggregated topology is exposed both as a RESTful API and as a native Kubernetes Custom Resource.

- We design a scoring algorithm that quantifies the network cost of placing a pod onto a node based on its communication dependencies. The plugin is implemented as a scoring-phase extension to the Kubernetes scheduler, adhering to the principle of minimal intrusion and enabling plug-and-play deployment.
- We introduce the AppGroup Custom Resource Definition to describe microservice-level communication patterns. This includes per-service importance and per-metric sensitivity weights, allowing fine-grained modeling of latency-, bandwidth-, and loss-sensitive interactions.
- We evaluate our scheduler on controlled network topologies using representative microservice workloads. Results show improvements in placement locality, end-to-end latency, and particularly tail latency under heterogeneous network conditions, with minimal scheduling overhead.

2 Background and System Context

2.1 Kubernetes Scheduler

`kube-scheduler` is a core control-plane component responsible for assigning unscheduled Pods to appropriate worker Nodes. For each newly created Pod that lacks a node assignment, the scheduler performs a two-phase decision process: filtering and scoring.

In the filtering phase, the scheduler identifies a subset of feasible nodes that satisfy the Pod's hard constraints, including resource availability (e.g., CPU and memory), node selectors, taints and tolerations, and affinity/anti-affinity rules. Nodes that fail to meet any of these criteria are excluded from consideration. If no feasible node is found, the Pod remains pending until the cluster state changes.

In the scoring phase, each feasible node is assigned a numeric score based on soft preferences, such as resource balancing or affinity hints. The node with the highest score is selected; ties are broken randomly.

Kubernetes supports a modular scheduling framework, where each filtering or scoring rule is implemented as a plugin. The `kube-scheduler` comes with a default set of plugins that prioritize CPU/memory resource efficiency and rule-based policies. However, it does not natively consider inter-node network metrics such as latency, bandwidth, or loss rate, even though these factors are increasingly critical for the performance of communication-intensive applications.

2.2 Scheduling Framework

The Kubernetes scheduling framework [11] provides a plugin-based architecture for customizing scheduling behavior through defined extension points. The scheduling process is divided into a *scheduling cycle* (filtering and scoring) and a *binding cycle* (finalizing placement).

Plugins can hook into various stages such as PreFilter, Filter, Score, NormalizeScore, and Bind. Among them, the Score extension point ranks feasible nodes and determines the final placement.

Our work extends this scoring phase by introducing network-aware logic that incorporates real-time metrics like latency, bandwidth, and loss rate—allowing communication-sensitive workloads to achieve better placement with minimal changes to the core scheduling framework.

2.3 Kubernetes Custom Resources

Kubernetes allows users to extend its API through Custom Resource Definitions (CRDs) [7], enabling new resource types to behave like native ones. Once registered, custom resources can be managed via `kubectl` and standard API interactions, supporting domain-specific extensions without modifying core components.

In our system, we define two CRDs: `NetworkTopology`, which exposes inter-node latency, bandwidth, and loss rate; and `AppGroup`, which captures microservice communication dependencies and their relative importance.

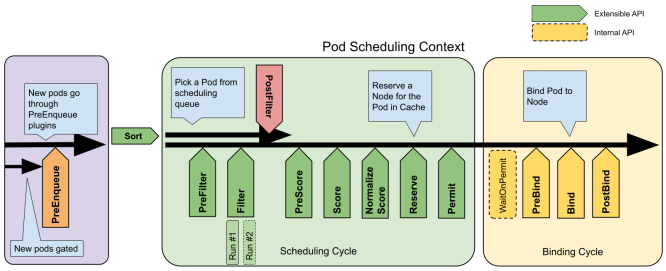


Figure 1. Scheduling Framework Extension Points

3 Related Work

3.1 Network-Aware Scheduling in Kubernetes

Senjab et al. [14] surveyed over 40 Kubernetes scheduling proposals, classifying them into four categories: generic, multi-objective, AI-driven, and autoscaling-based. Despite this breadth, they identified a key gap: *network-aware scheduling remains largely unaddressed*, especially with regard to real-time latency and bandwidth metrics.

Existing efforts are often limited in scope or loosely integrated. For instance, NetMARKS [16] uses Istio to collect network data but lacks integration with the scheduler plugin system. Other approaches like Kube-Knots [15] and BCDI [12] focus on hardware-level resources, neglecting inter-pod communication performance.

The Kubernetes SIG has proposed a network-aware scheduler plugin [9], but it is incomplete and non-production-ready. It uses static, hardcoded network topologies defined in YAML, lacks real-time metric support, and provides minimal documentation for its scoring logic. Its inclusion of a custom `QueueSort` adds unnecessary complexity, as scoring already determines placement in most cases.

Table 1 summarizes the capabilities of existing systems. While a few consider latency, support for bandwidth and real-time updates is rare. To our knowledge, our system is the first to integrate real-time latency, bandwidth, and loss rate directly into the official scheduler scoring pipeline.

Table 1. Comparison of Network-Aware Scheduling Approaches

Approach	Latency	Bandwidth	Dynamic Metrics	Plugin Integration
NetMARKS [16]	✓	✗	*Partial (via Istio)*	✗
Kube-Knots [15]	✗	✗	✗	✓
BCDI [12]	✗	✗	✗	✓
K8s SIG Plugins [9]	✓	✗	✗	✓
Our work	✓	✓	✓	✓

3.2 Scoring Algorithms for Communication-Aware Placement

Communication-aware scheduling has been extensively explored using task graphs to model inter-dependencies. Systems such as Firmament [3] and Ray [13] leverage global flow networks or DAGs to guide placement, focusing primarily on scheduling latency and resource fairness. However, they often overlook network-level metrics such as latency, bandwidth, and packet loss.

In edge and cloud computing, microservice placement strategies like EdgeWise [2] aim to optimize service locality and reduce network delay. These approaches typically rely on global optimization techniques (e.g., ILP or greedy search) and assume prior knowledge of the communication graph and infrastructure.

Faro [5] further advances this direction by modeling end-to-end latency—including queuing and transfer delays—for ML inference clusters. By integrating latency SLOs into placement decisions,

Faro demonstrates the value of communication-aware scheduling in performance-critical environments.

These efforts underscore the importance of accounting for inter-service communication in scheduling decisions. Our work draws from these principles to implement a practical network-aware scoring algorithm within Kubernetes. Unlike global optimization approaches, our plugin computes per-node scores based on live latency, bandwidth, and loss metrics, enabling scalable, localized decisions compatible with the native scheduler framework.

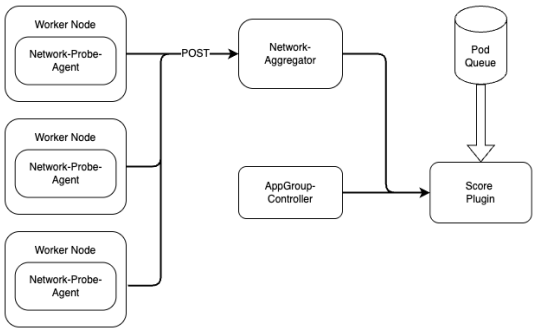


Figure 2. Components of our network-aware scheduling framework.

4 System Design

4.1 Framework

Figure 2 illustrate s the components of our network-aware scheduling framework:

Network Probe Agent: A lightweight DaemonSet deployed on each worker node. It periodically performs active measurements using `ping` and `iperf3` to capture node-to-node latency, bandwidth, and packet loss.

Aggregator Service: A centralized Deployment that collects metrics from all probe agents. It aggregates and exposes these metrics through both a REST API and updates to the `NetworkTopology` Custom Resource (CR).

AppGroup Controller: A controller that watches `AppGroup` CRs and resolves inter-service dependencies into a weighted communication graph. This graph is exposed via API to the scheduler.

Custom Scheduler: A modified kube-scheduler binary with our `NetworkScore` plugin. At scheduling time, it fetches the latest network metrics and dependency structure to compute placement scores.

4.2 Network topology CRD

We define a Custom Resource Definition named `NetworkTopology` to describe measured inter-node network conditions. It captures directional pairwise metrics between nodes, including latency, bandwidth, and loss rate, along with observed global extrema for score normalization.

- **spec.src.dst.latency**: Map from source node to destination node, value is latency in milliseconds.
- **spec.src.dst.bandwidth**: Map from source node to destination node, value is bandwidth in Mbps.
- **spec.src.dst.lossrate**: Map from source node to destination node, value is packet loss rate in percent (%), ranging from 0 to 100.
- **spec.[max/min]Latency**: Maximum/Minimun latency observed in the cluster.
- **spec.[max/min]Bandwidth**: Maximum/Minimun bandwidth observed in the cluster.
- **spec.[max/min]Lossrate**: Maximum/Minimun loss rate observed in the cluster.

Each metric is structured as a nested map. For example:

```
latency:
    nodeA:
      nodeB: 1.5
bandwidth:
    nodeA:
      nodeB: 300
lossrate:
    nodeA:
      nodeB: 10
```

4.3 AppGroup CRD

The `AppGroup` resource defines a communication-aware workload graph, where each node represents a service (or microservice), and edges capture

the communication dependencies and their relative importance across network metrics. This CRD enables the scheduler to interpret microservice-level interaction patterns during pod placement.

- **spec.workloads**: Array of workload specifications
  - **name**: Unique identifier for the workload
  - **selector**: Label selector to match pods associated with this workload
  - **weight**: Overall importance or traffic weight of the workload (floating-point value)
  - **dependencies**: Array of downstream services this workload depends on
    - **name**: Name of the dependent workload
    - **metrics**:
      - **latency**: Weight indicating sensitivity to latency
      - **bandwidth**: Weight indicating sensitivity to bandwidth
      - **lossrate**: Weight indicating sensitivity to packet loss

Below is an example of the `AppGroup` structure:

```
workloads:
    - name: frontend
      selector: app=frontend
     # Frontend is relatively high-traffic
      weight: 2.0
      dependencies:
        - name: checkout
          metrics:
            # Latency-sensitive
            latency: 0.7
            # Moderate throughput need
            bandwidth: 0.2
            # Some loss sensitivity
            lossrate: 0.1
```

4.4 Algorithm

To introduce network conditions into pod scheduling, we define a cost model that rewards pod placements with lower latency, higher bandwidth, or lower packet loss between dependent services.

The `ComputeScore` function computes the communication score between sourece node and destination node. It normalizes each metric into the range $[0, 1]$. Latency and loss rate are linearly normalized such that lower values yield higher scores. Bandwidth is logarithmically normalized to capture the diminishing returns of throughput increases and better reflect the quality perceived by the application.

Algorithm 1: ComputeScore Function

Input: Source node u, Destination node v, Metric weights m, Topology T
Output: Score $s \in [0, 1]$

if $u = v$ **then**
  $s \leftarrow$ `SameNodeScore`;
else if $T.latency[u][v]$ *and* $T.bandwidth[u][v]$ *and* $T.lossrate[u][v]$ *does not exsit* **then**
  $s \leftarrow$ `UnreachableNodeScore`;
else
  $normL \leftarrow \dfrac{T.maxLatency - T.latency[u][v]}{T.maxLatency - T.minLatency}$;
  $normB \leftarrow \dfrac{\log T.bandwidth[u][v] - \log T.minBandwidth}{\log T.maxBandwidth - \log T.minBandwidth}$;
  $normR \leftarrow \dfrac{T.maxLossrate - T.lossrate[u][v]}{T.maxLossrate - T.minLossrate}$;
  $s \leftarrow m[latency] \cdot normL + m[bandwidth] \cdot normB + m[lossrate] \cdot normR$;
return s

A small constant score of `80` (`SameNodeScore`) is returned when the source and destination nodes are the same, to mildly reward pod co-location while avoiding concentration. When all metric data is missing for a node pair, penalty score of `0` (`UnreachableNodeScore`) is applied to discourage placement in network-isolated zones.

Based on this, the score algorithm evaluates the communication quality between pods and their interdependent services.

For a candidate node, the plugin calculates a total communication score by summing over both

forward (dependencies) and reverse (dependents) paths defined in the AppGroup graph. The per-path scores are computed by aggregating weighted metrics including latency, bandwidth, and packet loss. Each application's influence on the total score is scaled by its corresponding communication weight, ensuring that services with higher throughput or latency sensitivity are prioritized accordingly. By including reverse dependency traversal, our approach avoids reliance on pod scheduling order and supports stable local optimizations even under asynchronous or randomized deployment patterns.

To ensure that the final node score is bounded and comparable across different workloads, we normalize the total weighted score by the sum of weights. This guarantees the output lies in $[0, \texttt{MaxNodeScore}]$, as required by the Kubernetes scheduling framework, with MaxNodeScore typically set to 100.

4.5 Alternative Algorithm Considered

In designing our plugin, we explored two extension points provided by the Kubernetes scheduling framework: QueueSort and Score. While our final implementation adopts the Score extension point, we initially considered using QueueSort to influence scheduling order. The idea was to apply a topological sort over the AppGroup dependency graph, prioritizing pods whose services have many downstream dependents. This would allow dependent pods, scheduled later, to benefit from more informed placement decisions.

Ultimately, we chose not to implement this strategy. Our scoring algorithm already accounts for both upstream (dependencies) and downstream (dependents) paths, reducing the significance of pod scheduling order. Moreover, achieving global placement optimality is unrealistic given Kubernetes' parallel and distributed scheduling process. Incorporating a queue-sorting layer would introduce complexity without clear benefit. Instead, we opted for a lightweight, modular scoring plugin that complements the default scheduler and remains effective without requiring global coordination.

Algorithm 2: Score

Input: Pod p, Candidate Node n, AppGroup Graph G, NetworkTopology T
Output: Score $s \in [0, \texttt{MaxNodeScore}]$

$app \leftarrow$ label of p
$thisNode \leftarrow n$
$w \leftarrow G[app].weight$
$totalScore \leftarrow 0.0$
$totalWeight \leftarrow 0.0$
foreach $(dep, metrics)$ *in* $G[app].dependencies$ **do**
  $depNodes \leftarrow$ findTargetNodes(dep)
  if $depNodes = \emptyset$ **then**
    continue
  $interscore \leftarrow 0.0$
  foreach $targetNode \in depNodes$ **do**
    $interScore +=$ ComputeScore($thisNode$, targetNode, metrics, T)
  $avgScore \leftarrow interscore/|depNodes|$
  $totalScore \mathrel{+}= w \cdot avgScore$
  $totalWeight \mathrel{+}= w$
foreach *dependent where* $G[dependent]$ *has app as dependency* **do**
  $metrics \leftarrow G[dependent].dependencies[app]$
  $w' \leftarrow G[dependent].weight$
  $depdentNodes \leftarrow$ findTargetNodes(dependent)
  if $epdentNodes = \emptyset$ **then**
    continue
  $interscore \leftarrow 0.0$
  foreach $targetNode \in depdentNodes$ **do**
    $interScore +=$ ComputeScore($targetNode$, thisNode, metrics, T)
  $avgScore \leftarrow interScore/|replicaNodes|$
  $totalScore \mathrel{+}= w' \cdot avgScore$
  $totalWeight \mathrel{+}= w$
return $s =$ $\texttt{MaxNodeScore} \cdot totalScore/totalWeight$

5 Implementation

We implement our system as an extension of the official Kubernetes scheduling framework, written in Go. All components are containerized and deployable in standard Kubernetes clusters. Key implementation details include:

Custom Resources: We define two CRDs—`AppGroup` and `NetworkTopology`—using OpenAPIv3 schemas. These allow external agents and controllers to declaratively report communication structure and network state.

Probe Agent: Each node runs a DaemonSet that actively measures inter-node latency, bandwidth, and packet loss at fixed intervals. Measurements are sent to the aggregator via HTTP POST.

Aggregator: Implemented as a Flask-based web service, the aggregator receives metrics, computes summary statistics, and publishes results to the REST API and `NetworkTopology` CR.

Scheduler Plugin: The core plugin logic is registered via the `Score` extension point. It is compiled into a custom scheduler binary built from the upstream `kube-scheduler` source.

Our full source code, deployment manifests, and experiment scripts are available at: https://github.com/YongyuLiu03/CS525_Project

6 Experiment

6.1 Network Emulation

To evaluate our network-aware scheduler under heterogeneous network conditions, we emulate an asymmetric network topology using Linux `tc` (traffic control) on Kubernetes worker nodes. Although the assigned virtual machines (VMs) are physically co-located on only two physical hosts—with default bandwidth up to 9000 Gbps and RTT as low as 0.5 ms—we artificially inject latency, bandwidth constraints, and packet loss to reflect realistic cluster heterogeneity.

The `tc` rules are applied on the `ens33` interface, which is the primary physical interface for outbound traffic on each VM.

The cluster consists of one control-plane node and up to 19 worker nodes, grouped into zones as shown in Table 2. Inter-zone characteristics are summarized in Table 3. Metrics are symmetric and apply to all node pairs between zones. Intra-zone links are configured to have low latency and high bandwidth, while the `FAR` zone simulates remote or degraded links with higher delay, loss, and limited throughput.

Table 2. Cluster Node Allocation

Zone	Node Range
Master (Control Plane)	`sp25-cs525-1401`
Zone A (Nodes A)	`sp25-cs525-1402` – `1406`
Zone B (Nodes B)	`sp25-cs525-1407` – `1411`
Zone C (Nodes C)	`sp25-cs525-1412` – `1416`
Zone FAR	`sp25-cs525-1417` – `1420`

Table 3. Simulated Inter-Zone Network Metrics

Zone Pair	Latency (ms)	Loss Rate (%)	Bandwidth (Mbps)
A ↔ A	0.5	0.0	1000
A ↔ B	1.0	2.0	300
A ↔ C	2.0	2.0	100
A ↔ FAR	10.0	5.0	20
B ↔ B	0.5	0.0	800
B ↔ C	5.0	2.0	100
B ↔ FAR	10.0	5.0	20
C ↔ C	0.5	0.0	500
C ↔ FAR	10.0	5.0	20
FAR ↔ FAR	2.0	2.0	100

6.2 Workload

We evaluate our network-aware scheduler using the Google Online Boutique microservice demo [4], a widely used benchmark for service-mesh and container orchestration research. It consists of 11 stateless microservices, written in multiple programming languages and communicating over gRPC, as illustrated in Figure 3.

To better capture realistic end-to-end latency, we exclude the default `loadgenerator` component and instead simulate user behavior from outside the cluster network. This setup ensures that frontend service delays reflect the full request-response chain, including network overhead.

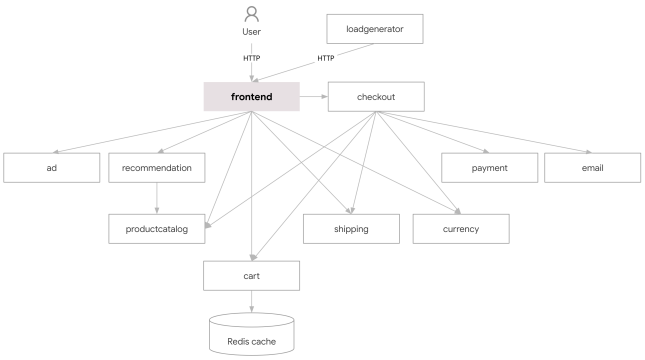


Figure 3. Google Online Boutique Microservices

Based on the gRPC call graph among services, we define an `AppGroup` resource to express communication dependencies and performance sensitivity across services. Each dependency is annotated with a set of weighted preferences over latency, bandwidth, and packet loss. These annotations guide the scheduler in selecting node placements that minimize total network cost for the most critical communication paths.

Table 4 summarizes the full workload configuration, including service importance and dependency-specific communication preferences.

6.3 Scheduler Configuration

To evaluate the impact of network-aware scheduling, we implement and deploy four different schedulers in our Kubernetes cluster:

- **Default Scheduler (`Default`)**: The standard Kubernetes scheduler with default scoring policies enabled, including resource requests, affinity rules, and topology spread constraints.
- **Network-Aware Only (`Custom`)**: A custom scheduler using only the `NetworkScore` plugin, with all default scoring plugins disabled. The plugin is assigned a weight of 1.
- **Hybrid-1 Scheduler (`Custom w=1`)**: A hybrid configuration where `NetworkScore` is combined with all default scoring plugins. The plugin is assigned a relative weight of 1.
- **Hybrid-5 Scheduler (`Custom w=5`)**: Same as above, but the network-aware plugin is assigned a higher weight of 5 to give greater emphasis to network conditions.

All custom schedulers are deployed using Kubernetes' out-of-tree scheduling framework. Each scheduler is configured with a unique `schedulerName` and launched as a separate Deployment. The default scheduler remains active throughout the experiment for comparative analysis. Scheduling decisions are logged for all pods to record placement outcomes under each strategy.

6.4 Load Generation

We use `Locust`[6], an open-source load testing framework, to emulate realistic user traffic against the Online Boutique frontend service. A custom Locust script simulates a typical shopping session, including the following steps:

1. Visiting the homepage
2. Browsing products and viewing details
3. Adding items to the shopping cart
4. Modifying quantities and checking out

All requests are issued from outside the Kubernetes cluster to capture realistic end-to-end latency. Each experiment is run with 100 concurrent virtual users over a 5-minute period. We collect response time metrics including mean latency, median, 90th percentile, and 99th percentile.

This setup allows us to assess the overall responsiveness of the application under different pod placements determined by each scheduler.

7 Evaluation

7.1 Pod Placement Comparison

We compare pod placement across four scheduler configurations. Figure 4 visualizes the number of pods placed in each zone for each scheduler, and Table 5 details the placement of individual services.

The default scheduler, unaware of network condition, distributes pods relatively uniformly across all available nodes, including those in the FAR zone, which simulates high-latency, low-bandwidth conditions. This uniformity arises from the default scheduler's emphasis on balancing node utilization and avoiding hot spots, rather than optimizing inter-service communication.

Table 4. Workload Specification and Communication Preferences

Service	Importance	Depends On	Latency	Bandwidth	Lossrate
frontend	1.0	recommendation	0.6	0.3	0.1
	-	productcatalog	0.7	0.2	0.1
	-	checkout	0.6	0.3	0.1
	-	currency	0.4	0.4	0.2
	-	cart	0.5	0.4	0.1
	-	shipping	0.4	0.3	0.3
	-	ad	0.7	0.2	0.1
recommendation	0.8	productcatalog	0.7	0.2	0.1
checkout	0.9	payment	0.4	0.3	0.3
	-	email	0.4	0.3	0.3
	-	shipping	0.5	0.3	0.2
	-	currency	0.4	0.3	0.2
	-	cart	0.4	0.3	0.2
	-	productcatalog	0.7	0.2	0.1
cart	0.8	redis-cart	0.5	0.4	0.1
ad	0.4	–	–	–	–
productcatalog	0.6	–	–	–	–
currency	0.8	–	–	–	–
payment	0.8	–	–	–	–
shipping	0.8	–	–	–	–
email	0.7	–	–	–	–
redis-cart	0.7	–	–	–	–

The custom scheduler places nearly all pods within Zone A, clearly preferring nodes with superior network characteristics. This concentrated placement demonstrates the effectiveness of our network-aware scoring mechanism. However, one outlier exists where the recommendation pod was placed in the FAR zone. This likely occurred because, at the time of scheduling, the pod's declared dependencies had not yet been placed, and the plugin had no connectivity data to inform a meaningful decision. In such cases, the scoring algorithm defaults to behavior similar to the standard scheduler. A further limitation is observed when a service with many downstream dependencies is scheduled early and randomly lands in a poor zone. This initial misplacement can negatively bias all subsequent decisions, as other services will optimize placement with respect to that poorly located pod.

Hybrid schedulers strike a better balance. With weight 1, pods are spread between Zones A and C, reflecting a compromise between resource distribution and network performance; with weight 5, placement shifts toward Zones A and B, reflecting stronger network preference. These results show that our plugin integrates well with default behavior and can steer decisions toward network-optimal placements when given higher weight.

7.2 Network Score Evaluation

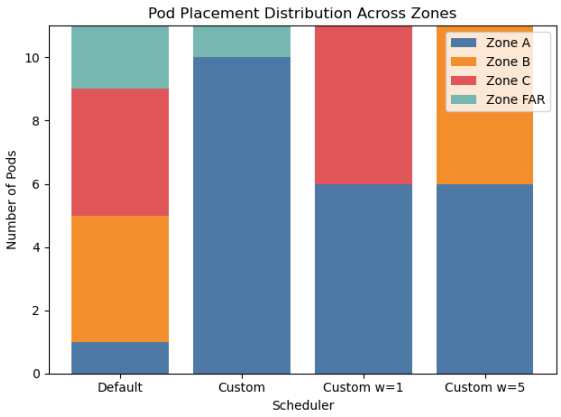
To evaluate the network efficiency of different schedulers, we analyze three key metrics: the total network score, the weighted average score per service, and the individual score breakdown for each service. These scores are derived based on the final pod placement of each scheduler and reflect the aggregated communication quality as defined by the `AppGroup` topology and the real-time network conditions.

Figure 5a shows the weighted average network score across all services. The default scheduler performs the worst, with an average around 65,

Table 5. Service Placement Across Zones

Service	Default	Custom	Custom w=1	Custom w=5
ad	C	A	A	A
cart	FAR	A	A	A
checkout	C	A	C	B
currency	FAR	A	C	A
email	B	A	C	B
frontend	B	A	A	A
payment	C	A	C	B
productcatalog	B	A	A	B
recommendation	B	FAR	A	A
redis-cart	C	A	A	A
shipping	A	A	C	B

Figure 4. Pod Distribution Across Zones

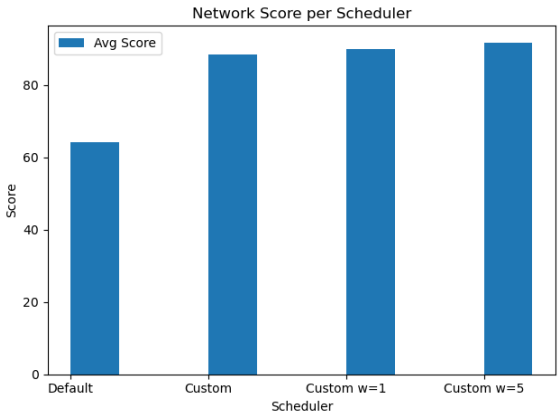


due to its lack of network or dependency awareness. In contrast, all custom schedulers score above 88, with the custom scheduler (weight 5) achieving the highest, confirming that stronger emphasis on network-aware scoring yields better placement. Notably, the custom scheduler with weight 5 achieves the highest score, showing that assigning a higher priority to network-aware scoring yields more optimized placements. This confirms the effectiveness of our plugin in improving service locality and communication quality.

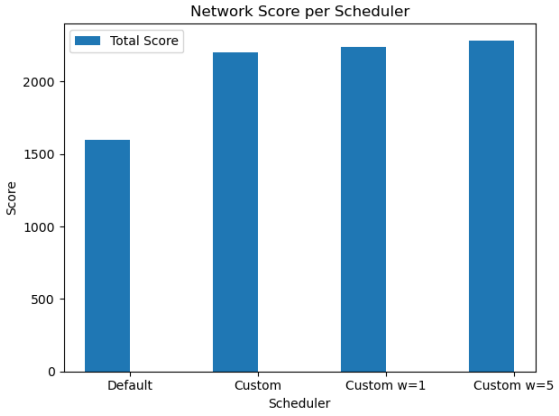
Figure 5b complements the previous figure by illustrating the total communication score summed over all services. Again, the default scheduler performs the worst due to uniformly distributing pods across all zones without considering traffic patterns. The custom schedulers yield significantly higher total scores, validating that the placements they make reduce inter-service communication cost at scale. Interestingly, the hybrid scheduler with weight 1 performs slightly worse than weight 5, suggesting that stronger emphasis on network metrics results in more globally optimal decisions.

Figure 6 provides a breakdown of per-app scores under each scheduler. The improvements are especially pronounced for communication-intensive services such as frontend, checkout, and recommendation. The default scheduler scores considerably lower on these services due to poor locality and placement in high-latency zones. In contrast, custom and hybrid schedulers consistently prioritize the communication needs of these core services, resulting in higher per-app scores and improved overall system performance.

Interestingly, and somewhat contrary to our expectations, the custom-only scheduler did not achieve the highest network score despite being fully dedicated to optimizing for communication quality. This outcome aligns with a limitation we previously observed: when none of a pod's declared dependencies have yet been scheduled, the custom scheduler lacks placement context and may fall back to decisions that resemble random or default behavior. As a result, the early placement of critical services can significantly influence subsequent scheduling quality. This highlights the value



(a) Average Score



(b) Total Score

Figure 5. Comparison of average and total network scores under different schedulers

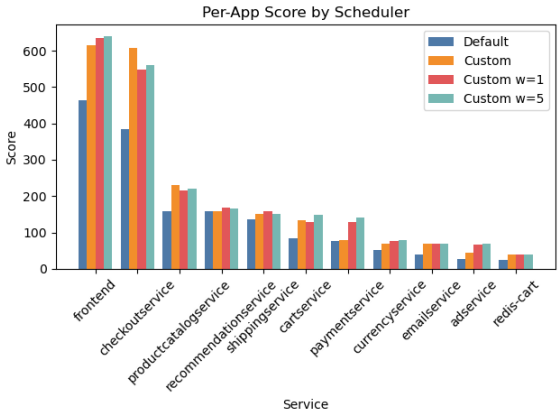


Figure 6. Per-App Network Score Breakdown

of hybrid schedulers that blend our plugin with the default logic—providing fallback strategies when network dependency information is sparse while still capitalizing on network-aware scoring when applicable.

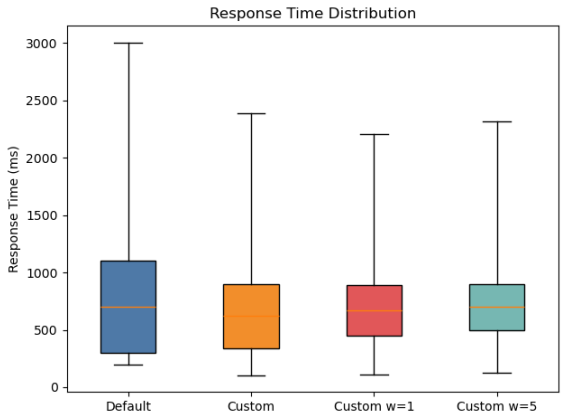


Figure 7. Response Time

In summary, these results demonstrate that our network-aware scheduler significantly improves pod placement decisions in terms of inter-service communication. Both average and total network scores validate the benefits of integrating real-time network conditions and service dependency weights into scheduling logic. The hybrid variants further show that our plugin can effectively co-exist with the default scoring pipeline, offering a tunable tradeoff between traditional resource-centric and communication-centric placement goals.

7.3 Response Time Analysis

Figure 7 shows the distribution of end-to-end response times observed under each scheduler configuration, based on simulated user behavior using Locust. The box plot summarizes median, interquartile range, and tail latency across thousands of HTTP requests.

The default scheduler yields the widest latency distribution, with a significantly higher median and more extreme outliers, indicating poor tail performance. This is expected, as the default scheduling logic does not account for inter-service communication patterns or network conditions, often leading to pod placements that span high-latency or lossy zones.

In contrast, our network-aware scheduler reduces both the median and tail latencies. The custom-only scheduler improves over the baseline by co-locating communication-heavy services, but it does not achieve the best latency profile due to the early placement uncertainty discussed earlier. Hybrid strategies that combine the default and custom scoring produce more robust results: they retain the placement stability of the default scheduler while leveraging network-aware scoring where available.

These results confirm that network-aware scheduling improves not just average communication cost metrics but also leads to tangible performance gains in user-perceived latency.

7.4 Overhead Analysis

We measure the resource consumption introduced by our network-aware scheduling infrastructure. Each worker node runs a lightweight probe agent as a DaemonSet, and a single aggregator Deployment collects metrics cluster-wide.

Based on `kubectl top`, `network-probe-agent` consumes approximately **1–4 mCPU** and **16–20 MiB** of memory each on average. The centralized `net-aggregator` consumes **1 mCPU** and **80 MiB** memory. These resource usages are negligible compared to typical application workloads and introduce minimal system overhead.

Additionally, we observe no measurable increase in pod scheduling latency when using our plugin compared to the default scheduler. This demonstrates that our network-aware scheduling approach is lightweight, scalable, and production-feasible.

8 Discussion

Our metric design takes advantage of the structural simplicity of microservice communication, which is typically composed of point-to-point RPC-style interactions rather than complex many-to-many flows. This allows us to model communication requirements at the service level with directional weights, capturing essential network sensitivity without excessive granularity.

Limiting our extension to the `Score` phase makes the plugin lightweight and easy to integrate. By considering both dependencies and dependents, our scoring algorithm mitigates the absence of global scheduling order and supports locally optimal placement, even under asynchronous deployments.

However, this local view can become a limitation. When a pod's dependencies are not yet placed, the plugin lacks context and may fall back to default-like behavior. This highlights the value of hybrid scheduling, where our plugin complements rather than dominates default scoring—aligning with Kubernetes' philosophy of composable scheduling.

Limitations

Our system introduces several limitations. First, the use of CRDs requires additional configuration overhead. Defining service-level weights and communication preferences demands manual effort and domain-specific knowledge, potentially increasing the operational burden for developers or DevOps engineers.

Second, while our evaluation focuses on the Google Online Boutique microservice workload, it remains limited in scope. Future work could explore other categories of network-sensitive applications such as streaming systems, scientific workflows, or distributed databases, and evaluate behavior under more diverse network topologies and failure conditions.

9 Conclusion

We present a network-aware scheduling framework for Kubernetes that enhances pod placement decisions based on real-time latency, bandwidth, and packet loss metrics. By introducing a pluggable scoring plugin and a structured AppGroup CRD, our approach allows fine-grained control over communication-aware scheduling while remaining lightweight and compatible with existing scheduler workflows. Experimental results demonstrate clear improvements in placement locality, network cost, and application-level performance.

References

[1] Deval Bhamare, Raj Jain, Mohammed Samaka, and Aiman Erbad. 2016. A survey on service function chaining. *Journal of Network and Computer Applications* 75 (2016), 138–155.

[2] Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee. 2019. EdgeWise: A Better Stream Processing Engine for the Edge. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 929–946. http://www.usenix.org/conference/atc19/presentation/fu

[3] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 99–115.

[4] Google. 2023. microservices-demo. https://github.com/GoogleCloudPlatform/microservices-demo.

[5] Beomyeol Jeon, Chen Wang, Diana Arroyo, Alaa Youssef, and Indranil Gupta. 2025. A House United Within Itself: SLO-Awareness for On-Premises Containerized ML Inference Clusters via Faro. In *Proceedings of the Twentieth European Conference on Computer Systems*. 524–540.

[6] Joakim Hamrén Hugo Heyman Jonatan Heyman, Carl Byström. [n. d.]. Locust Official Website. https://locust.io/.

[7] Kubernetes. [n. d.]. Kubernetes CRD (Custom Resource Definitions). https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[8] Kubernetes. [n. d.]. Kubernetes Homepage. https://kubernetes.io.

[9] Kubernetes. [n. d.]. Kubernetes Network-Aware Scheduling. https://scheduler-plugins.sigs.k8s.io/docs/plugins/networkaware/.

[10] Kubernetes. [n. d.]. Kubernetes Scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/.

[11] Kubernetes. [n. d.]. Kubernetes Scheduling Framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[12] Dong Li, Yi Wei, and Bing Zeng. 2020. A dynamic I/O sensing scheduling scheme in Kubernetes. In *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*. 14–19.

[13] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[14] Khaldoun Senjab, Sohail Abbas, Naveed Ahmed, and Atta ur Rehman Khan. 2023. A survey of Kubernetes scheduling algorithms. *Journal of Cloud Computing* 12, 1 (2023), 87.

[15] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2019. Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters. In *2019 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 1–13.

[16] Łukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. 2021. Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–9.