Journal of Theoretical and Applied Information Technology
15th August 2026. Vol.104. No.15

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

NEXUS: EVENT-DRIVEN DEPENDENCY-AWARE SCHEDULER FOR BURSTY MICROSERVICE WORKLOADS

MITULKUMAR RAJ[1] , PARAG SANGHANI[2] , NIRAJ SHAH[3]
[1]Assistant Professor, School of Engineering, P P Savani University, India
[2]Professor, P P Savani University, India
[3]Professor, School of Engineering, P P Savani University, India
E-mail: [1]rajmitul@gmail.com, [2]parag@ppsu.ac.in, [3]niraj.shah@ppsu.ac.in

ABSTRACT

Microservices workloads experience 10-20x traffic spikes within seconds, and hopping between nodes slows response times, so the Kubernetes scheduler must place pods quickly while keeping related services nearby. Existing schedulers address only one side of this tradeoff. Gang schedulers like Volcano incur continuous CPU overhead when idle, but network-aware frameworks lack runtime dependency detection and adaptive activation. In this paper, we introduce NEXUS, an event-driven Kubernetes scheduler extender that combines a spike detection engine integrated with Prometheus, a dependency graph engine that creates runtime DAGs from pod annotations, a gang lifecycle manager enforcing the temporary coordination, and a locality-aware dynamic node scoring system. Evaluating a 6-node AWS EKS cluster using the Google Online Boutique benchmark with 20x Flash, NEXUS reduces P95 latency by 8.1% compared to the default scheduler, 46.8% compared to YuniKorn, achieves a colocation ratio of 0.75, eliminates scheduling errors, reduces control plan overhead 22x compared to Volcano.

Keywords: *Kubernetes Scheduling, Container Orchestration, Microservice Architecture, Cloud-native System, Gang Scheduling, Burst workloads.*

1. INTRODUCTION

Cloud Native design principles had brought architectural shift from monolithic systems to microservices deployments. This shift has fundamentally reshaped the workload management in distributed environments. Moder Applications decomposed into tens or hundreds of loosely coupled microservices communicating via gRPC or REST [1]. While this decomposition improves maintainability and fault isolations, it introduces critical orchestration challenges under dynamic, often bursty traffic condition [2]

To ensure clarity, we briefly define the key terms used throughout this article. "Microservices are independently deployable software components that communicate with peer services through network protocols such as gRPC and REST. Gang Scheduling places groups of interrelated pods together as an atomic unit. This ensures that no pod run in isolation while other remain unscheduled. A directed acyclic graph (DAG) in this context is a topology for communication between microservices at runtime, where nodes are services and edges represent service-to-service calls. P95 tail latency (95th percentile) is the response time below which 95% of requests complete. This is a standard metric for determining the worst user experience under load. Control plane overhead refers to the CPU and memory resources consumed by the scheduling infrastructure itself, regardless of the application workload it manages.

Kubernetes(K8s) has become the de Facto standard for container orchestration, providing a flexible scheduling framework that assigns pods to cluster nodes based on resource availability and constraint satisfactions [3]. However, its default scheduler is designed for relatively stable and resource-consistent workloads, and it has fundamental limitations when facing microservice-specific requirements. Among all the primary limitation is the absence of runtime dependency awareness, as the default scheduler treats each pod as an isolated scheduling unit, ignoring the communication topology that binds related service together. This difference causes dependent services to be placed across distributed nodes, which exposing inter-service invocations to cross-node network serialization penalties. [4] [5].

Sudden traffic spikes events further exacerbate scheduling problems in microservice environments [6]. Burst of flash events caused by viral content, advertising campaigns, or cascading replays can increase request rates by an order of magnitude per second. During such events, the scheduler must quickly allocate new pods while ensuring that interdependent services remain collocated to minimize communication overhead. Since network latency greatly contribute to response time degradation in muti-tier applications [7]. scattered placement can significantly increase end to end latency and degrade user experience.

Existing solutions address only part of this problem. Gang Schedulers such as Volcano [8] and CoScheduler [9] provide coordinated placement but incur continuous control-plane overhead through always-on scheduling mechanism. Network aware schedulers such as Diktyo [7] and NetMarks [10] are topology aware but remain active at all times regardless of workload intensity. Adaptive framework such as TraDE [2] introduces dynamic scheduling policies, but do not support the formation of Gangs based on the runtime dependencies and do not eliminate idle overhead. Therefore, there is no existing platform that simultaneously provides runtime dependency awareness, spike-aware coordinated placement, and near-zero downtime overhead.

To address these limitations, this paper introduces NEXUS: a lightweight Kubernetes scheduler extender built on four complementary engines. First, the Prometheus driven spike detection engine activates scheduling logic only when spike condition is observed. NEXUS then dynamically creates a Directed Acyclic Graph (DAG) at runtime based on the actual pod annotations to identify communication dependencies. Third, a temporary coordination group is formed and scheduled using Locality aware node scoring to facilitate co-location of dependent services. Finally, a custom recovery process dissolves all execution structures once the burst completes, returning the scheduler to dormant state.

Evaluation on a six-node AWS EKS (v1.29) cluster running a 390-second burst workload demonstrated that NEXUS reduced idle-state scheduler CPU overhead to only 0.41ms approximately 22x lower than Volcano. It reduces P95 latency by 8.1% compared with the default Kubernetes scheduler and 46.8% compared with Apache YuniKorn. NEXUS achieves a pod co-location ration of 0.75, which reduce critical-path cross-node communication to 25%, eliminates scheduling-induced errors.

The main contribution of this paper is as following:

- Event Driven Triggering: A multi-signal burst detection model that uses QPS, error rate, P95 latency, and HPA events in combination with hysteresis-based state transition to eliminate scheduling activity during idle periods.
- Runtime Directed Acyclic Graph (DAG) construction: A framework for discovering dependency that dynamically creates communication graphs based on live pod annotations enables topology-aware planning without static configuration.
- Temporary Gang Scheduling: A lifecycle model that creates and dissolves temporary coordination groups within a burst window so that no overhead remains between events
- Locality-weighted node scoring: A composite scoring function that prioritizes gang co-location while considering node heterogeneity and available capacity.
- Comprehensive experimental evaluation: Validation of sie benchmark schedulers on AWS EKS using Google's Online Boutique benchmark. Statistically significant improvements were demonstrated across multiple performance metrics.

The remainder of this paper is structured as follows. Section II reviews related work. Section III describes the background on K8s scheduling mechanics and formalizes the problem. Section IV presents the optimization model. Section V describes the design of NEXUS. Section VI provides an experimental evaluation of our framework and Section VII Discussion and VIII concludes the paper.

2. RELATED WORK

This section evaluates the existing scheduling environment in two main areas: theoretical Gang and network-aware frameworks and open-source production cluster schedulers.

2.1. Gang Scheduling, Network-Aware, and Adaptive Frameworks

Gang scheduling/co-scheduling allows groups of related tasks to be scheduled at the same time, avoiding partial execution and blocking at the head of the queue. In Kubernetes, Gang Scheduling is primarily supported by external schedulers such as Volcano [8] and YuniKorn [11] for batch processing, data analytics, and machine learning workloads. This system assumes static group membership and long-running tasks, and does not support dynamic discovery of service dependencies or the formation of temporary groups. Therefore, it is not well suited for microservices where dependencies are dynamic and workload intensity fluctuates rapidly. Sakkara [12] leverages topology aware gang scheduling for AI workload but does not support dynamic microservice dependency awareness. KubeAI [13] is a PPO-based Kubernetes scheduler extender that uses real-time telemetry to optimize pod placement and improve resource utilization. However, continuous reinforcement learning and telemetry data collection add runtime overhead.

Network-aware scheduler considers infrastructure topology when making placement decisions. Diktyo [7] intrduces AppGroup and Network topology controller that score nodes based on measured network costs, providing up to 45% latency reduction and 22% higher throughput compared to the standard K8s schedulers. NetMARKs [10] similarly uses throughout and latency metrics to facilitate low-latency deployments. Both improve placement quality, but rely on static or slowly updated topology information and lack runtime dependency detection. Most importantly, both remain active at all times resulting in control-plane overhead regardless of workload intensity and lacking the dormant state efficiency of NEXUS scheduler. Although DRS [14] improves resource utilization, it introduces additional control overhead due to continuous monitoring, RL-based decision making, and communication between components, [15] enhances scheduling decisions using reinforcement learning but incurs additional scheduler computation due to neural-network inference and reward evaluation.

Optimization-based approached have also been applied to network-enabled placements [16] [17]. although the ILP and MILP formulations [18] provide optimal solution under resource and latency constrains, their computational complexity makes them impractical for real-time scheduling. Heuristic approached improves scalability, but are often evaluated only in simulations or small testbeds. Research on data centers [19], edge clouds [7] and fog environments [20] consistently demonstrate the value of location-based scheduling for latency critical applications.

Adaptive scheduling is a recent phenomenon. TraDE introduce traffic-aware scheduling policy that adapt to communication patterns, resulting in significant improvements over static approaches. However, TraDE operates continuously and focuses on adaptive routing rather than batch colocation or temporary gang formation. Research on serverless platforms [21] [22] further confirms the prevalence of bursty workloads that motivates dormant-active life cycle of the NEXUS. Studies on SLO-based resource management [23][24] and topology-aware scheduling [25][26] emphasizes the importance of knowledge of scheduling, cluster heterogeneity, and NUMA topology, which motivated the development of locality- and heterogeneity-aware NEXUS scheduling.

2.2. Open-Source Production Cluster Scheduler

Many production grade frameworks extend Kubernetes scheduling capabilities. Volcano provides a comprehensive scheduling framework with the support of Gang Scheduling and Task Topology plugins that support coordinated placement and similarity-based clustering. Additional plugin such as Binpack and DRF focus on resource utilization and fairness. However, Volcano remains active all the time, consuming 9.36ms of CPU during idle periods. This is 22 times more than NEXUS dormant-state overhead of 0.41ms.

The Kubernetes scheduler plugins repository provides additional plugins including CoScheduling, which ensure that pods belonging to the same PodGroup are scheduled automatically. CoScheduling shares Volcano's always-on design philosophy, registering 6.20 scheduling invocation per second during steady-state operation versus NEXUS's 0.00 and does not support complex heterogeneous pod dependencies. Apache YuniKorn provides hierarchical queue management with fine grained gang scheduling capabilities optimized for batch workloads. Its queue-based design results in significant scheduling delays under burst conditions. In our evalution, YuniKorn

produced a highest P95 tail latency which is 88% higher than NEXUS, and an error rate of 5.10% during flash spikes. This reflects scheduling delays that cause request to accumulate beyond the timeout threshold.

Table 1 summarizes a comparative analysis of the evaluated scheduling approaches along the most important aspects of bursty microservices workloads. NEXUS is the only platform that simultaneously offers event-driven activation, runtime DAG construction, temporary group formation, a verified dormant state, complete agility, low overhead, and batch-optimized throttling. Previous studies have either focused on coordination quality at the expense of ongoing overhead (Volcano, YuniKorn, Coscheduler) or on topology knowledge without adaptive lifecycle management (Diktyo, NetMARKS). NEXUS goes beyond the state-of-the-art by integrating runtime dependency discovery, temporary group scheduling, and event-driven lifecycle management into a single production-compatible K8- extender framework.

3. MICROSERVICE SCHEDULING IN KUBERNETES

Microservices in Kubernetes are packaged into pods, which are the smallest schedulable units consisting of containers that share a network namespace and storage volume [27]. Kube Scheduler selects target nodes through a two-phase pipeline. Filter predicates eliminate nodes that cannot satisfy resource requirements, affinity rules, and constraints. The scoring function then ranks the remaining nodes for binding. The scheduling frameworks [5] provides extension points (QueueSort, Filter, Score, NormalizeScore) that allow custom plugins or extenders to be used without modifying the core scheduler binary. NEXUS leverages the Extender interface and implements it at the filtering and prioritization stage via HTTP callbacks while maintaining compatibility with Managed Kubernetes Services.

Kube Scheduler schedules each pod independently, ignoring communication relationship between pods. For microservices applications, where end-to-end latency is the sum of delays in the critical service chain [28], this indifference comes at a cost. Therefore, scheduling plans must simultaneously accelerate placement and ensure locality of interdependent services. Existing schedulers address one concern or the other: Gang schedulers (Volcano, Coscheduler) ensure coordinated placement but consume 9.36 ms CPU even at zero workload; Dependency-aware [29][30] uses service dependency information to optimize orchestration time and response latency. [31]A dependency-aware deployment strategy based on heterogeneous graph neural networks and reinforcement learning has been proposed to optimize microservice placement by capturing complex inter-service relationships. However, the method focuses on deployment optimization and does not address adaptive runtime scheduling under bursty workloads. network-aware schedulers (Diktyo, NetMARKS) improve locality aware placement but lack adaptive activation during burst event. Persistent polling loops, thread pools, and queue structures impose non-trivial overhead during extended idle periods that characterise real workloads that costs accumulate across cluster lifetimes. NEXUS addresses both of these issues with its event driven extender mechanism. Dependency-aware coordinated placement during the burst window with dormant operational state during normal workloads.

4. OPTIMIZATION FORMULATION

4.1. Model Description and Variables

Table 2. List of mathematical notations utilized throughout the model description.

4.2. Scheduling Objective

The NEXUS placement problem is formulated as a multi-objective global allocation optimization a function φ that balances the end-of-peak latency, the communication cost between nodes, and the control plane scheduling overhead. $\varphi^* = \text{argmin}_\varphi \{ \alpha \cdot L_{tail}(\varphi) + \beta \cdot C_{net}(\varphi) + \gamma \cdot O_{ctrl}(\varphi) \}$

The following restrictions apply:

- Resource feasibility: The total resource sum of node n <= cap (n) for all nodes.
- Co-location preferences L if W(u,v)> W_thresh implies φ(u)= φ(v).
- Constraints of Activation: O_ctrl(φ) approx. 0 when the state is inactive.

4.2.1. Burst Detection Function

B(t) The burst detection function evaluates system state at time t via a logical disjunction across four infrastructure telemetry channels

$B(t) = [r\_t > R\_thresh] \vee [e\_t > E\_thresh] \vee [l\_t > L\_thresh] \vee [p\_t > P\_thresh]$

The following baseline threshold used by system: R_thresh = 100.0 QPS, E_thresh = 5.0%, L_thresh = 500 ms, and P_thresh = 10 pods. If Prometheus becomes unavailable, the pod pending metric serves as a fallback mechanism.

4.2.2. Scheduler State Transition Model

The activation state of scheduler $S: R \rightarrow \{0, 1\}$ captures the hysteresis characteristics of the NEXUS life cycle. Activation upon crossing a single threshold ensures rapid detection of bursts, while deactivation requires constant telemetry below the threshold throughout the recovery window $\Delta\_cd = 30$ seconds, avoiding oscillatory behavior at maximum load.

- $S(t) = 1$, if $B(t) = true \wedge S(t^-) = 0$ [DORMANT ->ACTIVE]
- $S(t) = 1$, if $S(t^-) = 1$ and burst exists within cooldown window
- $S(t) = 0$, if $S(t^-) = 1$ and no burst exist across the full cooldown window [ACTIVE->DORMANT]avoiding

4.3. Co-location and Communication Cost Model

The R_col pod colocation rate tracks the proportion of critical path service pairs mapped to identical infrastructure nodes. NEXUS increases this ratio to R_col = 0.75, which limits cross-node network traffic to 25%, representing a 3x reduction in network overhead compared to the default scheduler baseline.

4.3.1. Node Scoring Function

If this option is active, the node priority of pod P in target gang g on node n is computed using a weighted addition composite function.:

$Score(n, P, g) = LocalityScore(n, g) + HeterogeneityScore(n) + ResourceScore(n, P)$

A weighting of 100 points per participant in the locality score ensures a progressive colocation attraction that prioritizes colocation targets over infrastructure sublayers during active burst windows and concentrates dependent pods at shared node boundaries.

5. NEXUS FRAMEWORK: SYSTEM DESIGN

5.1. System Overview

Figure 1. shows the architecture of the NEXUS, a Kubernetes scheduling extender that integrates with the native scheduling pipeline during the filtering and prioritization stages without changing the Kube scheduler core. NEXUS consists of five main components: a Spike detection engine, a Dormant / Active state controller, a Dependency Graph Engine, a Gang Lifecycle Manager, and a Dynamic Node Scorer. The framework runs hibernated by default, with only the spike detection engine continuously monitoring cluster activity. When it detects a workload spike, NEXUS is activated and builds a runtime service dependency graph from Kubernetes metadata, forms a temporary group of dependent microservices, and performs ranking of nodes based on locality via an extension interface. Once the surge is mitigated, any temporary scheduling structures dissolve and the platform automatically returns to a sleep state, providing better service coordination during peaks while maintaining minimal overhead during normal operation

5.2. Spike Detection Engine

It runs in a continuous background loop and polls Prometheus at 1-second intervals to estimate requests per second, tail latency, error rate, , and HPA events. The transition from idle to active operation occurs in 12.5 ms, capturing a burst pods before the request queue begins to experience overflow condition.

5.3. Dependency Graph Engine

Upon activation of NEXUS, the Dependency Graph Engine creates an in-memory DAG of the microservice topology by querying the Kubernetes API server. It is to inspect running pods and parse 'nexus.io/depends-on' and 'nexus.io/service-group' metadata annotations.

5.4. Gang Life Cycle Manager

The Gang Lifecycle Manager implements an eight-stage state machine that restricts bulk scheduling memory allocation to strictly active batch windows. Once the cooldown window expires without interruption, memory structures are

cleaned up, resulting in a 6.7x faster cleanup compared to the persistent queue baseline.

5.5. Dynamic Node Scorer

The Dynamic Node Scorer implements the Extender-compatible Prioritize endpoint, evaluating target nodes based on locality alignment, resource stabilization weights and infrastructure heterogeneity.

5.6. Algorithm Design

Three algorithms formalize the basic NEXUS procedure described above. Algorithm 1 provides an event-driven spike detection and lifecycle Control loop. Algorithm 2 present Runtime DAG Construction and Critical Path Extraction. Algorithm 3 provides a locality-aware node scoring function.

Algorithm 1: NEXUS Event-Driven Spike Detection and Lifecycle Control

Input: r_t (QPS), e_t (error rate %), l_t (P95 latency ms), p_t (pending pods)
Output: S_state ∈{DORMANT = 0, ACTIVE = 1}

Initialize: S_state ← DORMANT

Loop every 1 second:
if Prometheus is reachable:
r_t ← queryQPS() ▷ Locust fallback if metric absent
if r_t > R_thresh → ACTIVATE(); continue
e_t ← queryErrorRate()
if e_t > E_thresh → ACTIVATE(); continue
l_t ← queryP95Latency() ▷ Locust fallback
if l_t > L_thresh → ACTIVATE(); continue
if checkHPAScaleUp() →ACTIVATE(); continue
▷ *All signals within bounds — no activation*
else:
if p_t ≥ P_thresh → ACTIVATE() ▷ conservative fallback only

if S_state = ACTIVE:
start cooldown_timer ← Δ_cd = 30 s
if all signals below threshold for Δ_cd:
S_state ← DORMANT
DissolveAll(); ClearDAG()

Procedure ACTIVATE():
if S_state = DORMANT:
S_state ← ACTIVE
BuildFromAnnotations() ▷ construct runtime DAG (Algorithm 2)
groups ← FindLongestPath(G) ▷ extract critical path
FormGangs(groups) ▷ create ephemeral gangs

Algorithm 2: Runtime DAG Construction and Longest-Path Critical Path Extraction

Input: Running pod set Pods across all namespaces
Output: RuntimeGroup CP representing the critical communication path

Function BuildFromAnnotations(Pods):
Initialize G.nodes ← empty adjacency list
for each pod p in Pods:
s ← ExtractServiceName(p.name) ▷ pattern matching + service registry
if annotation nexus.io/depends-on exists:
deps ← Split(annotation_value, ',')
for each dep d in deps:
G.nodes[s].append(Trim(d))
if G.nodes is empty:
LoadExperimentDefaults() ▷ Online Boutique critical path fallback
G.built ← true

Function FindLongestPath(G) → RuntimeGroup:
longest ← [ ]
for each start_node u in G.nodes:
path ← DFS_LongestPath(u, visited = { })
if |path| > |longest|: longest ← path
return RuntimeGroup { Name: 'critical-path', Services: longest }

Function DFS_LongestPath(u, visited):
visited.add(u); maxPath ← [ ]
for each neighbour v in G.nodes[u]:
if v ∉ visited:
path ← DFS_LongestPath(v, visited)
if |path| > |maxPath|: maxPath ← path
visited.remove(u)
return [u] + maxPath

Algorithm 3: Gang-Aware Locality-Weighted Node Scoring for Extender

Input: Pod P_target, NodeList N, Gang g (from GangManager)
Output: HostPriority list [(node_name, score)] for kube-scheduler

Function ScoreForExtender(P_target, N, g):

```
priorities ← [ ]
for each node n in N:
▷ Locality: reward co-located gang members
member_count ← g.NodePrefs[n.name]          ▷ O(1) map lookup
locality_score ← member_count × 100

▷ Heterogeneity: reward preferred tiers/regions
hetero_score ← (n.labels['cpu-tier'] = 'high') × 50
             + (n.labels['region'] = 'region-a') × 30

▷ Resource: secondary tiebreaker
cpu_avail ← n.Allocatable.CPU.MilliValue() / 1000
mem_avail_gb ← n.Allocatable.Memory.Value() / 1073741824
resource_score ← (cpu_avail × 10) + (mem_avail_gb × 1)

▷ Composite score
total ← locality_score + hetero_score + resource_score
priorities.append({ Host: n.name, Score: total })
return priorities
```

5.7. Complexity Analysis

All three algorithms are designed to run within the scheduling loop of each kube scheduler module, where low latency is important. Algorithm 1 runs in O(1) per polling iteration. Each telemetry request is an independent HTTP request to Prometheus that returns a scalar value, and state transitions require only logical validation and matching operations. Algorithm 2 (The dependency graph generation (DAG) constructs in O(|V| +|E|) time in by traversing all annotated pods and their dependencies once. The critical path is then extracted through DFS with backtracking requiring O(|V|·(|V|+|E|)) time. The microservices applications typically comprise on the order of 10^2 services, this process is complete in less than 1 ms. Algorithm 3 scales linearly with the number of available nodes in O(|N|) where each node score is computed using three arithmetic searches without recursive calls. In reality, the main cost is the bidirectional HTTP connection between the kube scheduler and the NEXUS extender endpoint, and the average scheduling latency per pod during the active phase was measured at 4.12 ms, well within the default kube scheduler extender timeout of 15 seconds.

6. EXPERIMENTAL EVALUATION

This section presents an evaluation of the NEXUS framework. Section VI-A describes the infrastructure configuration, target applications, workload structure, and the underlying scheduler. Section VI-B analyzes the scheduler overhead during steady-state operation. Section VI-C presents the end-to-end results of the testbed in terms of latency, stability, coordination quality, and recovery, and concludes with statistical tests of five experimental runs.

6.1. Experimental Configuration

6.1.1. Infrastructure

For our evaluation, we used Amazon EKS v1.30 on AWS us-east-1 in a dedicated VPC (nexus-vpc) with public/private subnets. Six t3.small worker nodes (2 vCPU, 2 GiB each) form two heterogeneous groups, Node Group A (Region = Region A, CPU Level = High) and Node Group B (Region = Region B, CPU Level = Standard) to provide location-based scheduling evaluation. The CNI VPC plugin with prefix delegation now supports up to 110 modules per node. Prometheus v2.45.0 (2 second cleaning interval) and Grafana provided monitoring. NEXUS ran on the Nexus system as an HTTP scheduler extension (/filter, /prioritize, port 9099) with the ability to poll Prometheus every 10 seconds and fall back to a Locust v2.24.0 in-cluster pod (port 8089) to obtain realistic traffic metrics with low variance.

6.1.2. Target Application

We used Google's Online Boutique Microservices (an e-commerce Application) as a evaluation target. which consists of 11 services (front end, payment service, payment service, currency service, shopping cart service, redis-cart, product-catalog service, delivery service, email service, recommendation service, and advertising service). All these services primarily communicate via gRPC and form a well-defined dependency graph with realistic critical paths (particularly through payment and delivery), making the application suitable for evaluating dependency-aware scheduling. Resource requests and limits are configured for each service to avoid monopolies while maintaining realistic operating conditions. Google's Online Boutique is widely adopted for Kubernetes and cloud planning research, allowing comparisons with previous benchmark results.

6.1.3. Workload Design

The 390-s experiment consisted of three stages. Phase 1 (0-120 seconds) applied 100 concurrent users in NEXUS idle to establish baseline latency and idle overhead. Phase 2 (120-270 seconds) scaled the load to 2000 users (~20x) and triggered NEXUS activation. We measured scheduling latency, P95 latency, colocation, and error rates while capturing Prometheus metrics, creating dependency graphs, identifying critical paths, forming service teams, and applying locality-aware prioritization. Phase 3 (270-390 seconds) brought the load back to baseline and assessed group dissolution, recovery time, and steady-state overhead. Together these steps cover the entire NEXUS lifecycle.

6.1.4. Baseline Scheduler and Metrics

NEXUS was evaluated against six baseline scheduling configurations that were deployed independently in separate experimental runs on the same infrastructure. The default Kube scheduler (vanilla Kubernetes), Volcano v1.8, Apache YuniKorn v1.3, Diktyo, Coscheduler (scheduling plugin), and static node affinity. Each benchmark used the same 390-second workload profile with the same online store deployment configuration. All experiments were repeated five times to establish statistical significance. The following metrics were collected in 2 second increments: P95 and P99 request latency (ms), HTTP error rate (%), scheduler CPU (ms) and memory (MB), call scheduling per second, module colocation rate, recovery time (sec), and activation delay (ms).

6.2. Scheduler Overhead During Steady-State Operation

Figure. 2 compares the scheduler's CPU load during the steady-state phase in descending order. NEXUS records a CPU delay of 0.41ms. That's 22.8 times faster than Volcano (9.36 ms), 11.0 times faster than Diktyo (4.51 ms), 7.9 times faster than Coscheduler (3.24 ms), and 5.1 times faster than YuniKorn (2.09 ms). The default scheduler does not process any scheduling work during idle periods, so the overhead is almost zero. NEXUS represents the irreducible cost (0.41ms) per second polling cycle for the Prometheus spike detection engine. This provides full active phase functionality while keeps hibernation overhead on the same order of magnitude as non-scalable schedulers.

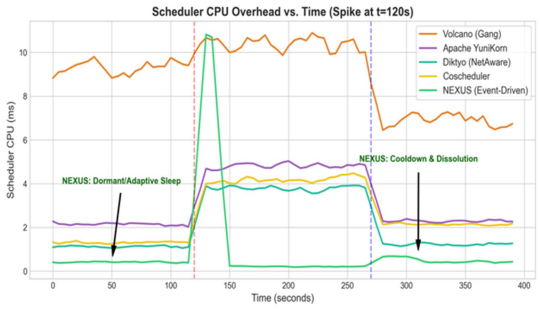

Figure 2: Scheduler Overhead Landscape — Idle-state CPU (ms): Default approx. 0 | NEXUS 0.41 | YuniKorn 2.09 | Coscheduler 3.24 | Diktyo 4.51 | Volcano 9.36

The memory overhead also follows a consistent pattern. When Idle, The NEXUS consumes 5.01 MB while Volcano consumes 19.62 MB. This reflects the lack of a persistent queue management in thread pools, structure and state cache scheduling during idle periods.

6.3. Scheduler End-to-End Testbed Results

6.3.1. P95 / P99 Tail Latency During Burst Workload

Figure. 3(a) and 3(b) demonstrate the comparison of the tail latencies (P95 and P99) during the period of workloads. The NEXUS has the lowest latency in both cases, The P95 having a latency of 11,333ms and whereas P99 having a latency of 94,333ms. Compared to the evaluated baseline schedulers, NEXUS maintains P95 the latency lowest (46.8%) compared to baselines under burst conditions.

We see this improvement due to dependency-aware colocation strategy of NEXUS. This strategy co-schedule interdependent microservices pods closer during peak traffic periods which reducing inter-node communication overhead and request propagation delays. In contrast, the Kubernetes default scheduler performs independent pod placement, while YuniKorn and Coscheduler schedule overdue tasks at the start of a spike, resulting in high latency. These results demonstrate that NEXUS's event driven mechanism and dependency aware placement mechanism are effective in improving service flexibility during sudden workload spikes.

Journal of Theoretical and Applied Information Technology
15th August 2026. Vol.104. No.15

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

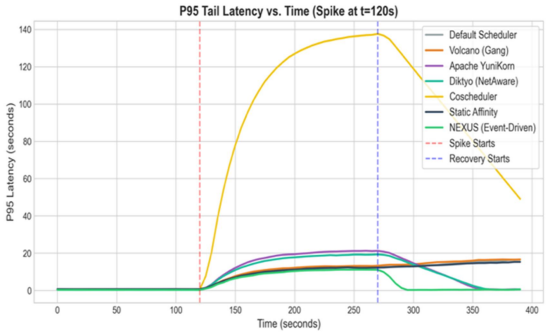

Figure 3 (a). P95 tail-latency during flash-spike

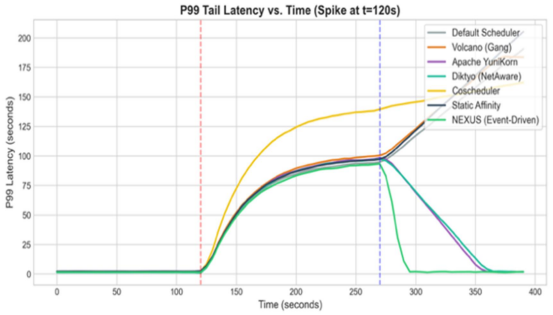

Figure 3(b).P99 tail-latency during flash-spike spike

6.4. Scheduler Stability Under Burst Workload

Figure. 4 shows the normalized normalized jitter metrics for each scheduler during the burst phase, reflecting the cumulative variance of scheduling throughput, error rate increase, and latency spike rate. The instability score is mentioned in a relative scale. NEXUS has the lowest instability score followed by Volcano, Default and Static Affinity. YuniKorn, Diktyo, and Coscheduler received a highest rating which reflects severe instability.

NEXUS provides maximum stability through a default inactive architecture, temporary group scheduling, and location-based placement. Together, these mechanisms eliminate scheduling delays, prevent partial service allocation, and support efficient scheduling when workloads increase by up to 20x.

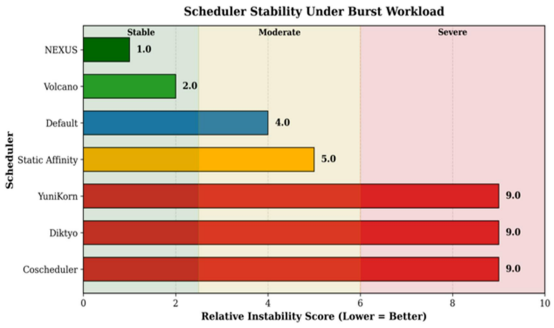

Figure 4. Scheduler Stability Under Burst Workload (Relative Instability Score).

6.5. Service Coordination Quality

Figure. 5 presents the pod co-location ratio. For each scheduler the fraction of co-dependent service pairs measured which placed on the same node boundary. NEXUS achieves a module colocation ratio of 0.75, which matches Volcano and outperforms Static Affinity (0.50) and the default scheduler (0.25). Unlike static approaches, NEXUS achieves this through DAG-aware group scheduling at runtime, reducing inter-node traffic by up to 25% compared to 75% with the default scheduler.

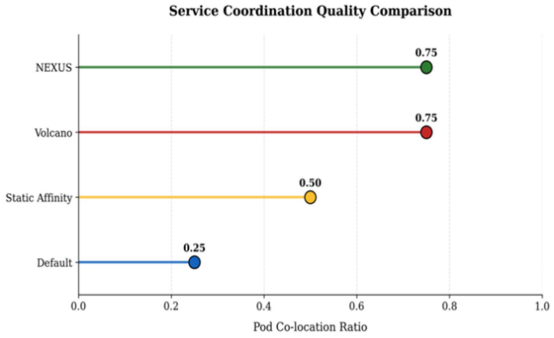

Figure 5. Service Coordination Quality

6.6. Strategic Positioning

In Figure 6, each scheduler is represented on a two-dimensional plane of costs for coordinating the point (5, 5) with the quadrant boundary. NEXUS occupies an ideal adaptation area (3, 8). Volcano achieves comparable tuning (8.0) with 2.7x more overhead. Faults account for weak adjustment (2 , 2).

6.7. NEXUS Adaptive Scheduling Lifecycle

Figure. 7 shows the temporal activity profile of the NEXUS across all operational phases. During the stages of steady state (1.0), burst detection (4.0) at t = 120 s, activation (8.0), adaptive regulation (9.0), stabilization (5.0), and

sleep recovery (1.0) at t = 299 s. The observed D_a = 180 s and D_d = 210 s are consistent with the theoretical predictions of equations, without any residual additional cost after dissolution.

Figure 6. Strategic Positioning of Scheduling Approaches (NEXUS occupies the Ideal Adaptive Region (coordinates 3, 8). Volcano achieves comparable coordination at 2.7x higher overhead.)

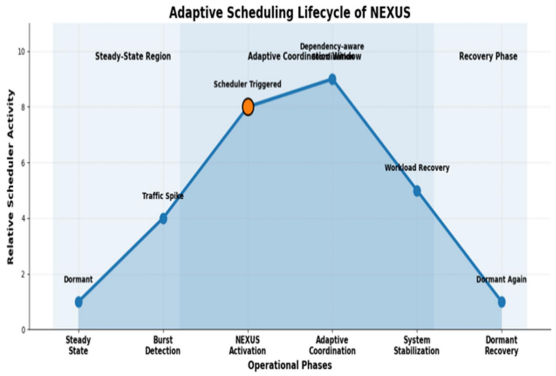

Figure 7. Adaptive Scheduling Lifecycle of NEXUS (Dormant | Burst Detection | Activation | Adaptive Coordination | Stabilisation | Dormant Recovery)

6.8. Comparative Capability Profile:

In Figure 8, compares all the baseline schedulers along six evaluation axes. NEXUS provides excellent or near excellent performance in terms of Latency, Throughput stability, Quality of coordination, batch adaptation, system stability, and overall balance. Its lower control efficiency reflects non zero cost of spike detection engine downtime.

6.9. Error Rate and Recovery

During peak spike phase, NEXUS achieves a lowest HTTP/gRPC error rate. Default scheduler records 3.07%, error rate where Volcano & YuniKorn achieves 0.60% & 5.10% respectively. Default and YuniKorn errors reflect scheduling delays that cause overflows and HTTP timeouts. The volcano error rate reflects a temporary state while waiting for the gangs to form. NEXUS eliminates errors with activation latency and pod scheduling latency. The Recovery state from spike to normal workload completes in 14.25 seconds. This is 6.7 times faster than Volcano, YuniKorn, Coscheduler and Default, reflecting memory CRD state clearing.

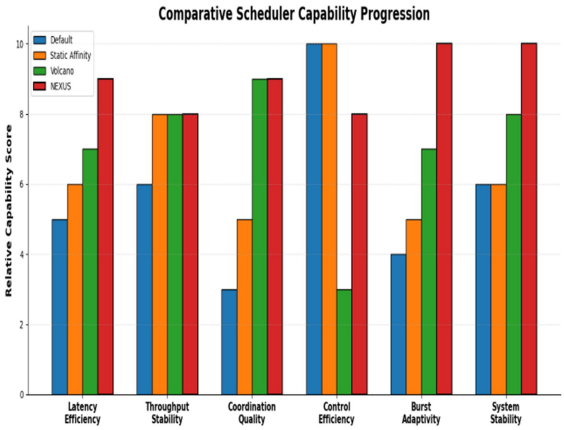

Figure 8. Comparative Evaluation of Scheduler Features

6.10. Statistical Validation

Table 3 shows the analysis of means, standard deviations, and 95% Lower and Upper CIs for the five analyses. Not all CI ranges overlap between NEXUS and competitors, confirming the significance of the p-level.

NEXUS CI error rate [0.000, 0.000] is conclusive for all five runs. CPU idle IC [0.392, 0.428] milliseconds reflect a 1 second deterministic polling cycle. The recovery time of CI [13.20, 15.30] does not completely overlap with the recovery times of Volcano [91.10, 98.90] and UniCorn [75.44, 84.56]. Taken together, the results confirm that NEXUS satisfies the three formulation constraints of Section IV with statistically reproducible evidence.

7. DISCUSSION

7.1. Deployment Feasibility and Kubernetes Compatibility

NEXUS is deployed via a Kubernetes Scheduler Extender YAML configuration, This requires no changes to the control plane or core scheduler binaries, and remains compatible with all managed Kubernetes offerings starting with version 1.10. It includes a single stateless Go binary that only requires access to the Prometheus API and standard RBAC pod-listing permissions; no external database or persistent volumes are required. The 0.41 ms CPU latency and 5.01 MB dormant footprint fits within standard component resource allocation without node reservation or priority class adjustment.

The annotation-based dependency model is natively compatible with Istio, Linkerd, and other service meshes and can automatically inject nexus.io annotations from observed traffic metrics, eliminating the need for extensive manual annotation. Gang Lifecycle Manager scales linearly based on the number of gangs. $O(10^2)$ service applications fit within the control plane budget. For $O(10^3)$-service deployments, the DFS longest-path algorithm may benefit from memorised dynamic programming. The polling interval is configurable for high-frequency or cost-sensitive environments.

7.2. Threats to Validity

Internal validity: All benchmarks were run on the same infrastructure with identical workload configurations, reducing confounding variables. You cannot fully control the residual AWS virtualization variability and external Locust network latency. However, the consistent plot conditions across all five runs and the low between-run standard deviations in Table III confirm that the variability remains within acceptable limits.

External Validity: The Online Boutique, although widely used, may not reflect production workloads with hundreds of services, circular models, or heterogeneous profiles. A six-node cluster does not reflect enterprise size, where scheduling latency and cluster complexity can vary. A 20x flash peak profile is one of several possible burst patterns. Gradual ramps, vibration loads, and multi-peak events are identified for future evaluation.

Construct Validity: P95/P99 Latency, Error Rate, and Scheduler CPU are standard metrics for distributed systems. Colocation rates vary by application and are adjusted depending on the topology of your Online Boutique. Detection thresholds have been established through preliminary experiments and may need to be adjusted for each deployment. Automatic annotation or special operator tools are required to implement large-scale production.

8. CONCLUSION AND FUTURE WORK

This paper presents NEXUS, an event-driven, dependency-aware Kubernetes scheduler extender for bursty microservices workloads. During steady state, NEXUS maintains zero scheduling invocations. without modifying the core scheduler binary, activates within 12.5 ms of burst detection, and orchestrates temporary gang formation along critical dependency paths discovered at runtime.

Evaluation on AWS EKS cluster under three-phase workload a 390-second demonstrates that NEXUS reduce CPU Overhead to 0.41ms during idle-state, which represent 22x improvement over Volcano. It returns to dormant state within 14.25 seconds (6.7x faster than the nearest compititors). It further achives P95 latency 8.1% below Default and 46.8% below YuniKorn; co-location ratio 0.75 reducing critical path cross node communication to 25%. Throughout the 20x workload spike, NEXUS sustain 0.00% error rate. Non-overlapping 95% CIs across five runs confirm all advantages at $p < 0.001$. These results demonstrate that event-driven scheduling with runtime dependencies can simultaneously achieve colocation quality, no burst phase errors, and negligible idle overhead. This combination is not achievable in the evaluated benchmarks.

Future work will focus on extending NEXUS while maintaining its lightweight default design. Promising areas include multi-cluster federation for coordinated scheduling across cluster boundaries, energy-aware node pricing for sustainable resource usage, and edge cloud orchestration for resource-constrained environments. Additional research will focus on lightweight predictive activation and dependency detection mechanisms.

Which will Improve scheduling efficiency without incurring significant overhead on control plane of cluster during regular operation.

REFERENCES:

[1] N. Dragoni et al., “Microservices: Yesterday, Today, and Tomorrow,” in Present and Ulterior Software Engineering, M. Mazzara and B. Meyer, Eds., Cham: Springer International Publishing, 2017, pp. 195–216. doi: 10.1007/978-3-319-67425-4_12.

[2] M. Chen, M. T. Islam, M. R. Read, and R. Buyya, "TraDE: Network and Traffic-aware Adaptive Scheduling for Microservices Under Dynamics," IEEE Trans. Parallel Distrib. Syst., vol. 37, no. 1, pp. 76–89, Jan. 2026, doi: 10.1109/TPDS.2025.3626424.

[3] C. Carrión, "Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges," ACM Comput. Surv., vol. 55, no. 7, pp. 1–37, Jul. 2023, doi: 10.1145/3539606.

[4] W.-K. Lai, Y.-C. Wang, and S.-C. Wei, "Delay-Aware Container Scheduling in Kubernetes," IEEE Internet Things J., vol. 10, no. 13, pp. 11813–11824, Jul. 2023, doi: 10.1109/JIOT.2023.3244545.

[5] Z. Rejiba and J. Chamanara, "Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches," ACM Comput. Surv., vol. 55, no. 7, pp. 1–37, Jul. 2023, doi: 10.1145/3544788.

[6] G. Somashekar, A. Dutt, M. Adak, T. Lorido Botran, and A. Gandhi, "GAMMA: Graph Neural Network-Based Multi-Bottleneck Localization for Microservices Applications," in Proceedings of the ACM Web Conference 2024, Singapore Singapore: ACM, May 2024, pp. 3085–3095. doi: 10.1145/3589334.3645665.

[7] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-Aware Scheduling in Container-Based Clouds," IEEE Trans. Netw. Serv. Manage., vol. 20, no. 4, pp. 4461–4477, Dec. 2023, doi: 10.1109/TNSM.2023.3271415.

[8] Volcano Project, "Volcano: Kubernetes-native Batch Scheduling System," GitHub. [Online]. Available: https://github.com/volcano-sh/volcano

[9] kubernetes-sigs/scheduler-plugins. (Jun. 17, 2026). Go. Kubernetes SIGs. Accessed: Jun. 21, 2026. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins

[10] L. Wojciechowski et al., "NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh," IEEE INFOCOM, 2021.

[11] "Gang Scheduling | Apache YuniKorn." Accessed: Mar. 13, 2026. [Online]. Available: https://yunikorn.apache.org/docs/user_guide/gang_scheduling/

[12] J. Santos *et al.*, "*Sakkara*: Intelligent Topology-Aware Scheduling for Kubernetes in the Age of AI," *IEEE Trans. Netw. Serv. Manage.*, vol. 23, pp. 5437–5451, 2026, doi: 10.1109/TNSM.2026.3703831.

[13] H. Dhanyatha *et al.*, "KubeAI-An AI-Powered Pod Placement Scheduler for Kubernetes," *IEEE Access*, vol. 14, pp. 80268–80279, 2026, doi: 10.1109/ACCESS.2026.3696055.

[14] Z. Jian, X. Xie, Y. Fang, Y. Jiang, T. Li, and Y. Lu, "DRS: A Deep Reinforcement Learning enhanced Kubernetes Scheduler for Microservice-based System," Jan. 04, 2023. doi: 10.22541/au.167285897.72278925/v1.

[15] H. Zhou, H. Y. Chan, S. Y. Zhang, M. E. Lin, and J. Ni, "A Kubernetes custom scheduler based on reinforcement learning for compute-intensive pods," in *Proceedings of the 2025 10th International Conference on Cloud Computing and Internet of Things*, Ho Chi Minh , Vietnam: ACM, Nov. 2025, pp. 82–91. doi: 10.1145/3785520.3785532.

[16] Y. Lin, Y. Shi, and N. Mohammadnezhad, "Optimized dynamic service placement for enhanced scheduling in fog-edge computing environments," Sustainable Computing: Informatics and Systems, vol. 44, p. 101037, Dec. 2024, doi: 10.1016/j.suscom.2024.101037.

[17] M. Zambianco, S. Cretti, and D. Siracusa, "Disruption-aware Microservice Re-orchestration for Cost-efficient Multi-cloud Deployments," IEEE Trans. Serv. Comput., vol. 18, no. 5, pp. 2754–2767, Sep. 2025, doi: 10.1109/TSC.2025.3604373.

[18] P. Soumplis et al., "Performance Optimization Across the Edge-Cloud Continuum: A Multi-agent Rollout Approach for Cloud-Native Application Workload Placement," SN COMPUT. SCI., vol. 5, no. 3, p. 318, Mar. 2024, doi: 10.1007/s42979-024-02630-w.

[19] G. Kontos, P. Soumplis, P. Kokkinos, and E. Varvarigos, "Cloud-Native Applications' Workload Placement over the Edge-Cloud Continuum:" in Proceedings of the 13th International Conference on Cloud Computing and Services Science, Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, 2023, pp. 57–66. doi: 10.5220/0011850100003488.

[20] C. Centofanti, W. Tiberti, A. Marotta, F. Graziosi, and D. Cassioli, "Taming latency at the edge: A user-aware service placement approach," Computer Networks, vol. 247, p. 110444, Jun. 2024, doi: 10.1016/j.comnet.2024.110444.

[21] F. Shaikh, G. Reali, and M. Femminella, "Mitigating Temporal Blindness in Kubernetes Autoscaling: An Attention-Double-LSTM

Framework," Mar. 2026, [Online]. Available: https://consensus.app/papers/mitigating-temporal-blindness-in-kubernetes-autoscaling-shaikh-reali/d24b3be534715d9d89df3a862e31ed4a/

[22] D. Fan and D. He, "A Scheduler for Serverless Framework base on Kubernetes," in Proceedings of the 2020 4th High-Performance Computing and Cluster Technologies Conference & 2020 3rd International Conference on Big Data and Artificial Intelligence, Qingdao China: ACM, Jul. 2020, pp. 229–232. doi: 10.1145/3409501.3409503.

[23] M. Adeppady, A. Conte, P. Giaccone, H. Karl, and C. F. Chiasserini, "Dynamic Management of Constrained Computing Resources for Serverless Services," IEEE Transactions on Network and Service Management, vol. 22, pp. 1646–1663, Apr. 2025, doi: 10.1109/tnsm.2024.3497155.

[24] A. Marchese and O. Tomarchio, "SLO-Aware Container Orchestration on Kubernetes Clusters," in 2025 IEEE 18th International Conference on Cloud Computing (CLOUD), Jul. 2025, pp. 318–327. doi: 10.1109/CLOUD67622.2025.00040.

[25] S. Akbari and M. Hauswirth, "Hiku: Pull-Based Scheduling for Serverless Computing," 2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 450–461, Feb. 2025, doi: 10.1109/ccgrid64434.2025.00034.

[26] Y. Qiao, J. Xiong, and Y. Zhao, "Network-aware container scheduling in edge computing," Cluster Comput, vol. 28, no. 2, p. 78, Apr. 2025, doi: 10.1007/s10586-024-04733-8.

[27] A. Marchese and O. Tomarchio, "Communication Aware Scheduling of Microservices-based Applications on Kubernetes Clusters:," in Proceedings of the 12th International Conference on Cloud Computing and Services Science, Online Streaming, --- Select a Country ---: SCITEPRESS - Science and Technology Publications, 2022, pp. 190–198. doi: 10.5220/0011049300003200.

[28] M.-A. Mirzaei, P. Poorsistani, and S. A. Javadi, "LEAD: Latency-Efficient Application Deployment for Microservices Architecture," in 2024 29th International Conference on Automation and Computing (ICAC), Sunderland, United Kingdom: IEEE, Aug. 2024, pp. 1–6. doi: 10.1109/ICAC61394.2024.10718812.

[29] S. Merkouche, T. Haroun, C. Bouanaka, and M. Smaali, "TERA-Scheduler for a Dependency-based Orchestration of Microservices," in 2022 International Conference on Advanced Aspects of Software Engineering (ICAASE), Constantine, Algeria: IEEE, Sep. 2022, pp. 1–8. doi: 10.1109/ICAASE56196.2022.9931568.

[30] A. Alelyani, A. Datta, and G. M. Hassan, "DAScheduler: Dependency-Aware Scheduling Algorithm for Containerized Dependent Jobs," J Grid Computing, vol. 21, no. 3, p. 46, Sep. 2023, doi: 10.1007/s10723-023-09679-6.

[31] H. Nie, X. Zhao, X. Bi, X. Yao, J. Zhou, and Y. Yuan, "Dependency-Aware Microservice Deployment Optimization via Neural Heuristic Learning," *IEEE Trans. Cloud Comput.*, vol. 14, no. 2, pp. 1038–1054, Apr. 2026, doi: 10.1109/TCC.2026.3680335.

Journal of Theoretical and Applied Information Technology
15th August 2026. Vol.104. No.15

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

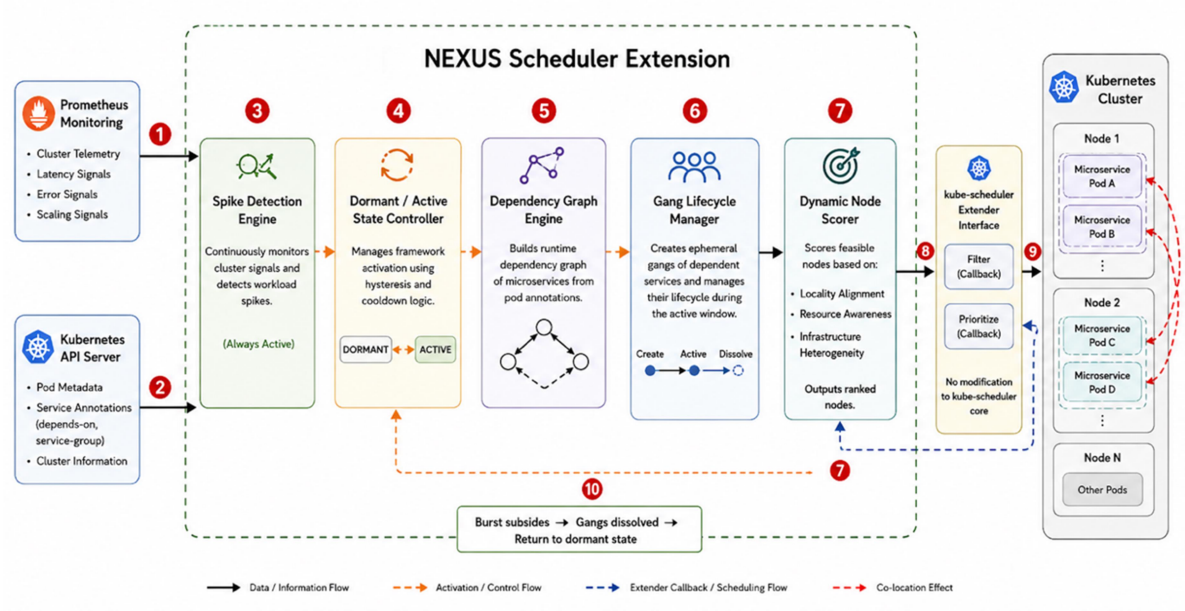

Figure 1. Architecture and Workflow of the NEXUS Framework

Table 1: Comparison of Kubernetes Scheduling Approaches

Scheduler	Event-Driven	Runtime DAG	Gang Sched.	Dormant State	Adaptive	Low Overhead	Burst-Optimized
Default K8s	No	No	No	Partial	No	Yes	No
Volcano	No	No	Yes	No	Partial	No	Partial
YuniKorn	No	No	Yes	No	No	No	No
Coscheduler	No	No	Yes	No	No	Partial	No
Diktyo	No	Static	No	No	Partial	Partial	No
TraDE	Part.	No	No	No	Yes	Partial	Partial
NEXUS	**Yes**	**Runtime**	**Ephemeral**	**Yes**	**Yes**	**Yes**	**Yes**

Table 2: Mathematical Notation for the NEXUS Formulation.

Symbol	Definition
r_t	API request rate (QPS) at time t
e_t	Error rate % (HTTP/gRPC) at time t
l_t	P95 tail latency at time t
p_t	pod pending count in namespace at time t
$S(t)$	activation state: DORMANT = 0, 1 = ACTIVE
$G = (V, E, W)$	Dependency Graph DAG: service vertices V, directed communication edges E, traffic-volume weights.
$CP(G)$	Dependency graph (G) of critical path
$\varphi: V \rightarrow N$	A Function of Pod placement that assigns each service v ∈ V to a cluster node n ∈ N
R_col	co-location ratio of Pod: percentage of critical-path service pairs that are collocated on the same node.
C_net	Percentage of inter node network traffic fraction = (1 − R_col)×100%
L_d	Spike detection latency: t_active−t_spike (sec/s)
$D\_a,\ D\_d$	Duraiton of Active and dormant during total experimental window
Δ_cd	Recovery window duration
α, β, γ	Weighting for tail latency, communication cost, and control overhead in the objective function

Table 3: Statistical Validation (Mean, Standard Deviation, and 95% Confidence Intervals (n = 5 Independent Runs)

Metric	Scheduler	Mean	Std Dev (σ)	95% CI Lower	95% CI Upper
Idle CPU (ms)	**NEXUS**	**0.410**	**0.021**	**0.392**	**0.428**
Idle CPU (ms)	Volcano	9.360	0.412	8.999	9.721
P95 Latency (ms)	**NEXUS**	**11,333**	**184.50**	**11,171**	**11,495**
P95 Latency (ms)	Default	12,333	241.10	12,122	12,544
Error Rate (%)	**NEXUS**	**0.000**	**0.000**	**0.000**	**0.000**
Error Rate (%)	Default	3.071	0.245	2.856	3.286
Error Rate (%)	YuniKorn	5.100	0.310	4.828	5.372
Recovery Time (s)	**NEXUS**	**14.25**	**1.20**	**13.20**	**15.30**
Recovery Time (s)	Volcano	95.00	4.50	91.10	98.90
Recovery Time (s)	YuniKorn	80.00	5.20	75.44	84.56