## Proposed Algorithm Name:

**PALEO** - Predictive Adaptive Latency-Efficient Optimizer

## Problem Statement

### Background & Context

Kubernetes (K8s) is the de facto standard for deploying and managing containerized applications. One of its most powerful features is autoscaling, which dynamically adjusts CPU and memory allocations based on resource usage metrics. However, existing autoscalers, such as the Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA), have limitations, especially for latency-sensitive applications operating under dynamic and unpredictable workloads.

### Limitations of Existing Autoscalers

1. **Metric Dependence**

In Kubernetes, autoscalers automate the process of adjusting resources allocated to containers to improve performance and efficiency.

* **Horizontal Pod Autoscaler (HPA)**:
  + Adjusts the number of pod replicas in a deployment.
  + Mostly CPU utilization by default.
  + Can scale on other metrics if explicitly configured (e.g., Prometheus metrics), but this adds complexity.
  + Scaling interval is usually every 15 seconds to 1 minute, depending on configuration.

Example: If average CPU utilization across pods exceeds 80%, HPA may increase pod replicas from 3 to 5.

* **Vertical Pod Autoscaler (VPA)**
  + Adjusts the CPU and memory requests/limits of individual pods.
  + Observes CPU and memory usage over time.
  + Pod needs to be restarted to apply new resource configurations.

Example: If VPA observes consistent memory usage above 90%, it might recommend increasing memory requests from 128Mi to 256Mi.

* **Limitations of Relying Solely on CPU/Memory**

While CPU and memory usage are critical infrastructure-level metrics, they do not directly reflect application performance from a user’s perspective.

| Aspect | CPU/Memory Autoscaling | Application Latency |
| --- | --- | --- |
| Focus | Resource consumption | User-perceived performance |
| ++/Granularity | Low-level container stats | High-level end-to-end behavior |
| Sensitivity to workload | Limited (e.g., CPU-bound only) | High — reflects combined impact of CPU, memory, IO, GC, etc. |
| Real-time adaptation | Delayed reaction | Often too late to save user experience |

* **Why Latency Is a True Quality-of-Service (QoS) Indicator**

Latency measures the time taken to respond to a user request — the single most important performance metric for real-time systems.

Latency reflects:

* CPU performance and contention
* Memory pressure (e.g., garbage collection pauses)
* I/O bottlenecks (disk, network)
* Application-level issues (e.g., lock contention, algorithm complexity)
* Traffic load and concurrency

Advantages of Latency as a QoS Metric:

* Directly maps to user experience (e.g., app feels slow or responsive).
* Holistic: Captures effects of multiple layers of the system.
* Workload-aware: Responds to changes in request rate, complexity, and traffic pattern.

Example: An API might show only 50% CPU usage but still have 500ms response times due to a memory-intensive workload or lock contention — HPA/VPA won’t act, but the user experience suffers.

* **CPU/Memory vs Latency: A Concrete Example**

| Metric | Scenario | What HPA/VPA See | What User Sees | Outcome |
| --- | --- | --- | --- | --- |
| CPU = 60% | App under load but optimized | No action | Fast response (latency < 100ms) | OK |
| CPU = 30%, Memory = 40% | App struggling due to DB or thread starvation | No action | Latency = 1s+ | Poor UX |
| CPU = 90% | App is maxed out | HPA scales pods | Latency = 500ms dropping to 200ms | Slow reaction |
| Latency = 800ms | CPU = 50% | No HPA trigger | Users abandon app | Critical issue missed |

* **HPA/VPA Are Not Latency-Aware**

By design, Kubernetes does not treat latency as a native metric for autoscaling:

* Latency is an application-level metric (e.g., measured in Prometheus via histogram buckets, percentiles).
* Needs custom metrics pipeline (Prometheus + Adapter + HPA config).
* Even if used, latency thresholds can be hard to tune for dynamic workloads.
* **Why This Matters for Modern Applications**

Modern cloud-native applications:

* Have microservice architectures (latency adds up across services).
* Run on multi-tenant clusters (unpredictable resource contention).
* Face real-time SLAs (e.g., 95% requests < 100ms).
* Experience dynamic traffic (e.g., mobile, seasonal spikes).

In such cases, relying on CPU/memory is insufficient to ensure optimal performance.

While CPU and memory usage are useful for infrastructure-level scaling, end-to-end latency is the most direct and meaningful QoS signal from the user’s perspective. Any autoscaling solution that ignores latency is at risk of making poor decisions that fail to protect the user experience.