## 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.

### Key Features of PALEO

|  |  |
| --- | --- |
| Feature | Description |
| Online Learning | Continuously updates models with new data; no offline retraining required. |
| Application-Agnostic | Learns behavior from usage/latency patterns; no hardcoded rules per app. |
| Latency-Aware Optimization | Predicts impact of new allocations on latency before applying changes. |
| Dual Prediction Pipelines | Separates resource demand forecasting and latency impact estimation. |
| Safe Resource Enforcement | Applies changes only if future latency stays under a defined threshold. |

#### Deep Dive

**Online Learning**

Continuously updates models with new data; no offline retraining required.

**What it means:**

PALEO uses online machine learning algorithms that learn incrementally, updating the model every time new observations (metrics) arrive or after a batch of observation arrive (to reduce the overfitting).

**Why it matters:**

* Real-world Kubernetes workloads change frequently due to time-of-day patterns, user behavior, deployments, or system noise.
* Traditional ML models require periodic offline retraining, which is slow and may not reflect the current state.
* Online learning ensures the autoscaler adapts immediately to recent data, avoiding stale decisions.

**How it works:**

As new data comes in (e.g., CPU usage, memory usage, request rate, observed latency):

* The resource usage model is updated to forecast future resource demand.
* The latency model is updated to reflect how latency reacts to resource allocations.

This allows PALEO to operate in a streaming, stateful fashion, learning over time without full retraining.

**Benefits:**

* Adaptability to new workloads without manual intervention.
* Resilience to concept drift (i.e., changes in workload/resource/latency relationships).
* Lower operational overhead — no need for dataset storage, retraining, versioning, and redeployment.

**Application-Agnostic**

Learns behavior from usage/latency patterns; no hardcoded rules per app.

**What it means:**

PALEO is designed to operate on any application, regardless of:

* Programming language
* Tech stack (e.g., Python, Node.js, Java)
* Internal logic
* Service type (e.g., API, batch, stateful)

It does this by observing universal behavioral metrics, not application-specific internals.

**How it works:**

* Uses generic system metrics: CPU/memory requests/limits/usage, request rate, and latency (P50/P95/P99).
* Learns patterns via statistical or ML models, not hardcoded thresholds or service rules.
* Does not need prior profiling or custom configuration per application.

**Benefits:**

* Scalability across services: One engine for many workloads.
* Zero-tuning deployment: No need to craft service-specific autoscaling policies.
* Generalizability: New services get optimized from Day 1.

Example: It can handle a real-time chatbot, a machine learning inference engine, and a Node.js API using the same logic.

**Latency-Aware Optimization**

Predicts impact of new allocations on latency before applying changes.

**What it means:**

PALEO doesn’t just observe high CPU or memory — it predicts whether changing resources will improve or harm latency.

**Why it's revolutionary:**

* Traditional autoscalers use reactive logic — increase CPU if it’s >80%.
* But high CPU doesn’t always mean latency is bad. Likewise, increasing memory doesn’t always improve performance.
* PALEO uses a predictive model to simulate what would happen to latency if a proposed resource change is applied.

**How it works:**

* Predicts future latency using:
* Current and proposed CPU/memory allocation
* Current resource usage
* Request rate and trend

* If the prediction shows latency will stay under a configured threshold (e.g., 95th percentile < 100 ms), the change is allowed.

**Benefits:**

* Minimizes performance regression from poorly chosen resource updates.
* Saves resources by not over-provisioning unless it actually improves user experience.
* Avoids QoS violations by accounting for future user impact, not just system state.

**Dual Prediction Pipelines**

Separates resource demand forecasting and latency impact estimation.

**What it means:**

PALEO runs two independent ML pipelines:

* Resource Forecasting Pipeline:
* Predicts how much CPU/memory the application will need in the near future.
  + Takes into account past usage trends, request rates, and usage bursts.
* Latency Impact Estimation Pipeline:
* Predicts how a proposed CPU/memory configuration will affect future latency.
  + Uses models trained on past resource configurations and observed latencies.

**Why it’s important:**

* Decouples supply forecasting (how much you might need) from QoS assurance (what impact will this have).
* Prevents overfitting a single model to two very different prediction tasks.
* Allows independent optimization of prediction accuracy for efficiency (resources) and experience (latency).

**Benefits:**

* Precision: Resource forecasting isn’t distorted by latency noise.
* Control: Latency model can enforce strict SLAs, while usage model can track growth.
* Balance: Enables cost-efficient scaling without sacrificing latency guarantees.

**Safe Resource Enforcement**

Applies changes only if future latency stays under a defined threshold.

**What it means:**

Before PALEO enforces a resource update (e.g., reducing CPU from 0.2 to 0.1), it checks if that action will still preserve latency SLAs.

**Why this matters:**

* Many autoscalers apply changes based on utilization alone, which can backfire:
* E.g., VPA reduces CPU and causes latency spike.
* PALEO integrates QoS validation into the decision loop.
* Ensures that any change applied is safe, i.e., will not degrade performance or violate user experience guarantees.

**How it works:**

Compares predicted latency post-change against the configured SLA threshold.

If the predicted value is:

* Below threshold (e.g., 95th percentile latency < 100ms): apply the change.
* Above threshold: deny or defer the change, and optionally alert the user.

**Benefits:**

* QoS Assurance: Keeps user-perceived performance stable and predictable.
* Error Avoidance: Prevents autoscaler from applying harmful updates (like VPA restarts or CPU throttling).
* Trust & Reliability: Safe changes build confidence in automation.

Think of this as a "safety belt" on top of predictive optimization.

|  |  |  |
| --- | --- | --- |
| Feature | Role in PALEO | Key Benefit |
| Online Learning | Enables real-time model updates | Keeps up with changing workloads |
| Application-Agnostic | Works across all app types | No custom logic required |
| Latency-Aware Optimization | Predicts impact before acting | Prevents latency spikes |
| Dual Prediction Pipelines | Splits resource & latency modeling | Improves accuracy & control |
| Safe Resource Enforcement | Checks future QoS before updating | Protects user experience |

PALEO isn't just an autoscaler — it's a predictive, adaptive, and intelligent resource orchestrator designed for modern, latency-sensitive cloud-native environments. With online learning, predictive foresight, and safety constraints, it goes far beyond traditional autoscalers like HPA/VPA to ensure performance, efficiency, and reliability.

## ALGORITHM MODULES (LOGICAL ARCHITECTURE)



1. App Pods → Metrics Collection (CPU/Memory usage & requests)
2. Metrics Collection → Base Latency Calculation (processes metrics into latency data)
3. Base Latency Calculation → Prediction Module (ML models for forecasting)
4. Prediction Module → Decision Module (threshold check)
5. Decision Module → Resource Enforcement (K8s API patches)
6. K8s API → App Pods (resource updates applied)