## 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)

**Metrics Collection**

**Purpose**

This is the eyes and ears of our intelligent autoscaling system. Its job is to periodically monitor and collect key runtime metrics about the application and its surrounding infrastructure in a non-intrusive and efficient way.

These metrics become the input features for our online learning and optimization pipelines.

**Periodic Data Collection**

The module collects data at regular intervals, e.g., every 15–30 seconds. The interval is a tunable parameter depending on:

* Model sensitivity
* System resource limits
* SLA criticality

Typical collection involves scraping, querying, or subscribing to metric sources like:

* Kubernetes Metrics API
* Prometheus time-series DB
* Node exporters
* Sidecar agents (e.g., Envoy, Istio, custom probes)
* cAdvisor or kubelet stats

The most suitable method depends on our system's needs for granularity, performance, accuracy, and ease of integration. Here's a breakdown comparing each option, followed by a clear recommendation for different use cases — especially for latency-aware, resource-optimized autoscaling like our PALEO system.

**Comparison of Metric Sources**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Source | What It Provides | Pros | Cons | Use Case Fit |
| Kubernetes Metrics API | Basic CPU/memory usage of pods/nodes (collected by metrics-server) | Simple to use, built-in | No historical data, no latency or custom metrics, limited granularity | Basic autoscaling, Not enough for latency-aware decisions |
| Prometheus | Full time-series metrics: CPU, memory, custom metrics, latency, request rate, etc. | Highly customizable, supports alerting, P95 latency, integrates with exporters | Requires setup, data volume can grow large | Best for detailed autoscaling and latency models |
| Node Exporter | Node-level stats: disk, CPU, memory, network, etc. | Low overhead, good for infrastructure monitoring | Not pod-specific; doesn’t provide latency or application-level data | Good for node health, Not enough for PALEO-type scaling |
| Sidecar Agents (Envoy, Istio, Linkerd) | Per-request latency, throughput, retries, errors, etc. | Application-agnostic, no app code changes, works well with Prometheus | Adds some overhead, mesh configuration can be complex | Ideal if app doesn’t expose metrics natively |
| cAdvisor / Kubelet stats | Real-time container resource usage | Fine-grained raw stats, used under the hood by others | Can be noisy and low-level; not meant for long-term use directly | Used by VPA/HPA internals, but not ideal standalone |
| Custom Exporters / Probes | Anything we define (latency, queues, memory, etc.) | Maximum flexibility, tailored metrics | Must build/maintain per service or proxy | Great fallback for apps without built-in telemetry |

**Most Suitable Method for PALEO**

Since we're designing a latency-aware, online learning, application-agnostic autoscaler, here’s what we need:

|  |  |
| --- | --- |
| Requirement | Source |
| CPU/memory usage (per pod) | Prometheus or cAdvisor |
| CPU/memory requests & limits | Kubernetes API |
| Real-time latency (P95, P99) | App’s Prometheus metrics, Service Mesh, or Sidecar |
| Request rate | Prometheus, service mesh, or custom probe |
| Historical time-series for online learning | Prometheus |

**Final Recommendation**

|  |  |
| --- | --- |
| Metric Type | Recommended Source |
| CPU & Memory Usage | Prometheus scraping cAdvisor metrics (via kubelet endpoints) |
| CPU & Memory Requests/Limits | Kubernetes API (e.g., kubectl get pod -o json, or direct API calls) |
| Latency Metrics (P95, etc.) | Prometheus metrics exposed by app or collected via sidecar proxy (Envoy/Istio) |
| Request Rate | Service mesh metrics, or add a lightweight custom probe or Prometheus exporter |
| Historical Data Storage | Prometheus time-series DB |

**CPU & Memory Usage**

Source: Prometheus scraping cAdvisor metrics (via kubelet endpoints)

What is it?

* These metrics reflect the actual usage of CPU (in millicores or nanocores) and memory (in bytes) by a container or pod in real time.
* cAdvisor (Container Advisor) runs inside the kubelet and provides detailed usage statistics for each container.

What’s scraped:

* container\_cpu\_usage\_seconds\_total: cumulative CPU time
* container\_memory\_usage\_bytes: current memory usage
* container\_memory\_working\_set\_bytes: "hot" memory (excluding cache)
* Usage can be converted to percentages using known CPU limits or system cores.

Collection Method:

* Prometheus scrapes these metrics via:
  + kubelet endpoint (e.g., https://<node>:10255/metrics/cadvisor)
  + Or via kube-state-metrics for Kubernetes abstractions

Why it’s important for PALEO:

* CPU/memory usage is a critical input for the resource demand prediction pipeline.
* Helps detect under- or over-utilization, efficiency trends, and resource bottlenecks.
* Used to train online models that map resource usage → optimal allocations.

**CPU & Memory Requests / Limits**

Source: Kubernetes API Server

* kubectl get pod -o json
* Direct REST API calls to /api/v1/namespaces/{namespace}/pods

What is it?

* The requested and maximum (limit) amount of CPU/memory configured in the pod/deployment.

Expressed in:

* CPU: millicores (e.g., 100m = 0.1 vCPU)
* Memory: bytes/MiB/GiB

Fields:

* .spec.containers[].resources.requests.cpu
* .spec.containers[].resources.limits.memory

Why it’s important for PALEO:

* These are control variables that PALEO will adjust.
* Knowing the current config is essential to predict how changing resources will affect latency.
* Used to compare with usage to detect over/under-provisioning.

**Latency Metrics (P95, P99)**

Source: Prometheus metrics exposed by the application or collected via service mesh/sidecar proxy (e.g., Istio, Envoy)

What is it?

* End-to-end request latency, often exposed as a histogram or summary metric in Prometheus format.
* Percentile values (e.g., P95, P99) reflect user experience under load.

Use rate() and histogram\_quantile() in PromQL to compute percentiles.

If the app does not expose metrics:

* Use sidecars (e.g., Envoy, Istio) that automatically collect latency, request count, and errors.
* If sidecar is unavailable, deploy a custom latency probe (like Prometheus Blackbox Exporter).

Why it’s important for PALEO:

* Latency is our primary QoS metric.
* Used to:
  + Predict impact of new resource settings
  + Avoid actions that would push latency above thresholds
  + Train latency estimation models

**Request Rate**

Source:

* Service Mesh Metrics (e.g., Istio, Linkerd, Envoy)
* Application Prometheus Metrics
* Custom Probe / Exporter

What is it?

* Number of incoming requests per second (RPS) per pod or per service.
* Reflects application load and helps model resource demand trends.

**How to collect:**

* If application exposes Prometheus counters for request count, use those.
* Otherwise:
  + Use Istio telemetry (istio\_requests\_total)
  + Use Envoy’s stats (envoy\_cluster\_upstream\_rq\_total)
  + Deploy a lightweight API gateway / sidecar that tracks and exports request counts.

Why it’s important for PALEO:

* Request rate helps correlate spikes in load to spikes in latency/usage.
* Crucial for forecasting future demand and understanding burst behavior.
* Used as a feature in both latency and usage prediction pipelines.

**Historical Data Storage**

Source: Prometheus Time-Series Database (TSDB)

What is it?

* Prometheus stores every scraped metric as a timestamped time series in its internal TSDB.
* Retention typically ranges from hours to weeks, depending on setup.

Storage model:

* metric\_name{label1="val1",label2="val2"} → list of (timestamp, value) pairs
* Data is compressed, indexed, and queryable via PromQL.

Why it’s important for PALEO:

* Enables feature extraction from historical behavior:
* Moving averages
* Variance
* Temporal patterns
* Powers online learning pipelines by feeding them the most recent data in real time.
* Can be exported to external ML pipelines via exporters like Prometheus remote-write, Thanos, or Cortex.

**If Apps Don’t Expose Prometheus Metrics?**

* First fallback: Use sidecar proxy (e.g., Envoy) to collect latency and request metrics passively.
* Second fallback: Run a custom latency probe pod (like [Prometheus Blackbox Exporter](https://github.com/prometheus/blackbox_exporter)) to measure from outside.
* Worst-case: If no latency metrics at all → mark as low observability app, and use conservative resource-based policies only.

|  |  |
| --- | --- |
| Goal | Recommended Tool |
| End-to-end autoscaling for real-time, latency-sensitive services | Prometheus + Sidecar Proxies + Kubernetes API |
| Minimum viable metrics pipeline | Prometheus + metrics-server |
| Fully observable, safe autoscaling engine | Prometheus (time-series) + latency + usage metrics with proper fallback logic |

## Online Learning

**Purpose**

This is the **c**entral intelligence of our autoscaler. It continuously learns and adapts to changing workload behavior by incrementally training two separate models:

1. Resource Demand – Predicts how much CPU/memory the service will need in the near future.  
    – Helps proactively scale resources before saturation.
2. Calculate the Allocation

– After predicting the resource usage, the resource allocation will be calculated to keep the 20% headroom

1. Latency Simulation Model – Predicts what latency the application will experience under a given CPU/memory allocation.  
    – Helps avoid QoS violations by only applying changes that keep latency under the SLA.

Both models are trained online: they are updated continuously with each new observation, rather than being retrained offline in batches.

**Why Two Models? (Resource Demand vs. Latency Simulation)**

**High-Level Idea**

|  |  |
| --- | --- |
| **Model** | **Goal** |
| Resource Demand Model | Predict how much CPU/memory the app will need in the near future based on usage trends |
| Latency Simulation Model | Predict how latency will respond to a hypothetical CPU/memory change |

These are separate modeling problems — both are essential to optimize resource allocation without breaking performance.

**Resource Demand Model**

Learn a function: f(usage\_t, request\_rate\_t, history\_t) → predicted\_resources\_(t+1)

This model answers:

* How much CPU and memory will this app need in the next time range, based on current usage and recent trends?

Input Features:

|  |  |
| --- | --- |
| Feature | Description |
| cpu\_usage\_pct | CPU usage as a percentage of request/limit |
| memory\_usage\_pct | Memory usage as a percentage of request/limit |
| request\_rate\_rps | Current requests per second |
| cpu\_allocated, memory\_allocated | Current CPU/memory allocations |
| past\_usage\_trend | Rolling average/variance of usage over last N minutes |
| time\_of\_day, day\_of\_week | Optional temporal features to capture periodicity |

Output (Predicted Target):

* cpu\_needed\_t+1: predicted needed CPU for next interval (e.g., in millicores)
* memory\_needed\_t+1: predicted needed memory for next interval (e.g., in MiB)

Model Type:

* Regression (continuous targets)
* Online models such as:
* SGDRegressor (Stochastic Gradient Descent)
* Online Decision Trees (e.g., Hoeffding Tree in River)
* Recursive Least Squares
* Lightweight neural nets with streaming updates

Why it matters:

* Reacts instantly to changing workloads (e.g., traffic surges)
* Avoids underprovisioning, which leads to latency spikes
* Avoids overprovisioning, which wastes resources

**Latency Simulation Model**

Objective

* Learn a function: g(cpu, memory, request\_rate, usage\_stats) → predicted\_latency\_(t+1)

This model answers:

* If I apply this new CPU/memory allocation, what will happen to latency?

It enables safe simulation before making changes.

Input Features:

|  |  |
| --- | --- |
| Feature | Description |
| cpu\_allocated, memory\_allocated | Proposed resource allocation |
| cpu\_usage\_pct, memory\_usage\_pct | Recent resource usage |
| request\_rate\_rps | Incoming traffic rate |
| latency\_p95\_t | Previous observed latency |
| burstiness\_score | Difference between recent max & average request rate |

Optional: number\_of\_threads, GC time, I/O stats if available

Output (Predicted Target):

* latency\_p95\_t+1: Predicted 95th percentile latency under the new allocation

Model Type:

* Regression (continuous output)
* Possible models:
  + Kernel Ridge Regression
  + Online SVR (Support Vector Regression)
  + LightGBM/XGBoost with incremental updates
  + Feedforward Neural Network with experience replay
  + Bayesian Regression (to capture uncertainty)

Why it matters:

* Prevents applying a resource change that causes latency regression
* Lets the system simulate multiple options and pick the best
* Guarantees QoS by only applying changes if predicted latency < threshold

Online Learning Process

1. Collect new data: every 30s–1m via Observation Module
2. Feature extraction: build vector of current state
3. Predict:
   * Future CPU/memory demand (f(...))
   * Future latency if new config is applied (g(...))
   * Calculate the limits & requests with 20% headroom
4. Update:
   * Train models with latest data point:
     + X = current features
     + y = actual resources used (for f), actual latency (for g)
   * Use online regression algorithms to update in real time
5. Act:
   * Only apply new allocation if predicted latency is acceptable
   * Store predictions and errors for self-evaluation

Advanced Ideas

* Model Confidence: Add uncertainty estimates (e.g., quantile regression, ensemble variance)
* Multi-objective optimization: Balance cost vs. latency
* Meta-learning: Learn how well models work per application type
* Warm-start from offline trained models, then refine online

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Purpose** | **Input Features** | **Output** | **Benefit** |
| Resource Demand Model | Predict resource need | Usage %, rate, trends | CPU & memory needed | Avoids overload, ensures readiness |
| Latency Simulation Model | Predict latency of new config | Proposed resources, usage, rate | P95 latency | Ensures safe scaling, preserves QoS |

## Prediction

**Objective**

The Prediction Module is the decision-making engine of PALEO. It uses up-to-date metric observations to:

1. Forecast the required CPU and memory for an upcoming time window
2. Simulate the latency the application would experience if those predicted resources were applied

This ensures the system proactively adjusts resources in a safe, QoS-preserving way.

Decision

**Objective**

The Decision Module is responsible for:

* Validating predictions (from the Prediction Module)
* Enforcing resource safety constraints
* Applying or rejecting configuration changes to Kubernetes

It only applies new CPU/memory allocations when the predicted P95 latency is less than or equal to the user-defined latency\_threshold

This ensures that autoscaling:

* Is QoS-aware
* Doesn’t break SLAs or user experience
* Balances aggression and safety

**Input Signals**

The Decision Module receives:

|  |  |  |
| --- | --- | --- |
| Signal | Source | Description |
| predicted\_cpu\_millicores | Resource Demand Model | Suggested CPU request |
| predicted\_memory\_mebibytes | Resource Demand Model | Suggested memory request |
| predicted\_latency\_p95\_ms | Latency Simulation Model | Expected latency if above resources are applied |
| latency\_threshold\_ms | User-defined config | Maximum allowed P95 latency (e.g., 100ms) |
| current\_allocation | Kubernetes API | Current CPU/memory request values |
| cooldown\_period\_active | Scheduler state | Prevents frequent flapping |

**Core Logic**

Here’s the heart of the module logic in structured form:

Algorithm (Pseudocode)

if predicted\_latency\_p95\_ms <= latency\_threshold\_ms:

    if significant\_change(predicted\_cpu, predicted\_memory, current\_cpu, current\_memory):

        if not cooldown\_period\_active:

            apply\_resource\_change(predicted\_cpu, predicted\_memory)

        else:

            log("Cooldown active — change delayed.")

    else:

        log("Change insignificant — skip.")

else:

    log("Predicted latency too high — rejecting new allocation.")

Conditions Explained

1. Latency Threshold Check

predicted\_latency ≤ latency\_threshold

* Primary safety condition.
* Prevents applying configurations that would degrade user-perceived performance.
* Latency is predicted by the Latency Simulation Model.

2. Significant Change Check

abs(new\_cpu - current\_cpu) ≥ min\_delta

* Avoids noisy small fluctuations that lead to unnecessary rescheduling.
* Define a minimum meaningful change, e.g.:
* CPU: ≥ 20m
* Memory: ≥ 64Mi

3. Cooldown Period Check

* Once a change is applied, enforce a grace period (e.g., 5 min) before next change.
* Prevents churning (frequent up/down adjustments that destabilize pods).

**Resource Enforcement**

* If all checks pass, the module applies the new allocation using the Kubernetes API. We can use Rolling Updates or create custom controllers to prevent disruptions.

**What If Latency Prediction Is Unreliable?**

Use uncertainty-aware safety margin:

if (predicted\_latency + prediction\_error\_margin) ≤ threshold:

    # Accept

This can come from:

* Ensemble variance
* Confidence intervals
* Rolling MAE from previous predictions

**Optional: Latency-Aware Fallback Scaling**

If latency prediction fails or exceeds the threshold:

* Fall back to conservative scaling using CPU/memory
* Raise an alert for operator attention
* Use a last-known-safe configuration

|  |  |  |
| --- | --- | --- |
| Check | Description | Purpose |
| predicted\_latency ≤ threshold | Main QoS condition | Prevents SLA violation |
| Δ resources ≥ min\_delta | Significance filter | Prevents churn |
| !cooldown\_active | Temporal gate | Ensures system stability |
| apply\_patch() | Action step | Changes resource requests in K8s |

## Resource Enforcement

**Purpose**

The Resource Enforcement Module is responsible for:

* Applying new resource requests (CPU & memory) to the appropriate containers/pods.
* Doing so safely, efficiently, and non-disruptively.
* Using the Kubernetes API to PATCH deployments, pods, or custom controllers.

It executes the final step of the autoscaling process:  
 “Talk to Kubernetes and apply the change.”

**Input Signals**

|  |  |  |
| --- | --- | --- |
| Signal | Description | Source |
| cpu\_request, memory\_request | New resource values to apply (e.g., 200m, 512Mi) | Decision Module |

**Key Concepts**

Resources in Kubernetes

Each container has two resource specs:

* requests: guaranteed amount (scheduler uses this)
* limits: maximum allowed amount (runtime enforces this)

PALEO typically modifies requests (limits can optionally be updated too).

**Using the Kubernetes API**

Access Method:

* RESTful API calls to Kubernetes
* Via official client libraries:
  + Python: kubernetes (pip install kubernetes)
  + Go: client-go
  + Node.js: @kubernetes/client-node

**What Happens Under the Hood?**

* A PATCH request is sent to the Deployment resource.
* Kubernetes notices the template change → triggers rolling update:
  + Creates a new ReplicaSet
  + Gradually terminates old pods and spins up new ones with new resources
  + Ensures zero downtime (if configured with proper readiness/liveness probes)
* Pods get rescheduled if the new requests can’t be satisfied on the current node (e.g., not enough memory)

**Updating Stateful Workloads**

For stateful workloads (e.g., StatefulSets or single pods), we can:

* Update StatefulSet → same rolling logic applies
* Or directly PATCH Pod → but only works for limits, not requests, and will not persist (because the pod is ephemeral)

For most cases, update the controller spec (Deployment, StatefulSet, Job).

**Rate-Limiting & Safety**

You must rate-limit and validate updates to prevent churn and instability:

|  |  |
| --- | --- |
| Feature | Description |
| min\_change\_delta | Only patch if CPU/memory changed > threshold (e.g., 50m, 64Mi) |
| cooldown\_period | Wait time before next enforcement (e.g., 5–10 mins) |
| graceful\_rolling\_update | Ensures that patch causes safe pod replacement |
| max\_parallel\_pods | Controls how many pods update at once |

**Intelligent Enforcement Strategies**

We can choose different patch strategies depending on app needs:

|  |  |  |
| --- | --- | --- |
| Strategy | Description | When to Use |
| Rolling Update (default) | Standard Deployment patch triggers rolling pod restarts | Most web services, microservices |
| Immediate Pod Patch | Direct patch to pod resource spec | Debug or ephemeral services |
| HPA/VPA CRD Injection | Patch into HPA or VPA custom resource definitions | If hybrid autoscaling is used |
| Custom Controller | Manage updates via CRD and operator logic | For platform-level abstractions (e.g., Knative) |

**What If Enforcement Fails?**

Our system should:

* Catch and retry errors (e.g., 409 Conflict, 500 Timeout)
* Monitor rollout status (kubectl rollout status)
* Use Prometheus metrics to observe pod readiness & health
* Fall back to previous working configuration if latency spikes post-deployment

|  |  |  |  |
| --- | --- | --- | --- |
| Task | How it’s done | Tools/APIs | Notes |
| Apply resource change | Patch Deployment | K8s API via client.AppsV1Api().patch\_namespaced\_deployment | Triggers rolling update |
| Track impact | Observe new pod latency & usage via Prometheus | Use Observation Module | For online learning feedback |
| Handle errors | Retry, log, rollback if needed | Wrap with try/catch | Use kubectl rollout status or API |

## Resource Optimization Objective

**Objective Function**

We’ve defined an optimization goal as a weighted sum of two competing factors: resource cost and predicted latency.

Objective=α⋅(cpu+memory)+β⋅predicted\_latency

Where:

* α: weight (or penalty) assigned to resource usage cost
* β: weight (or penalty) assigned to predicted latency
* cpu: allocated CPU (typically in millicores)
* memory: allocated memory (typically in MiB)
* predicted\_latency: estimated latency for a given allocation (usually P95 in ms)