# Objective

To intelligently and autonomously optimize CPU and memory resource limits for multiple microservices deployed in a Kubernetes environment, such that:

Application latency remains within acceptable bounds, defined as no more than certain percentage of degradation from the service-specific baseline latency under normal load.

Resource configurations adapt dynamically and continuously, without relying on statically defined Critical Reduction Points (CRPs) or manual tuning thresholds.

# Expanded Sub-Objectives

1. Resource Efficiency

* Reduce overprovisioning of CPU and memory requests/limits.
* Automatically discover and apply the minimum safe resources required to meet Service-Level Objectives (SLOs), thereby minimizing infrastructure costs.

1. SLA Adherence

* Continuously ensure that latency does not exceed a certain amount of the baseline latency observed at initial overprovisioned levels.
* Latency-aware adjustments must be consider individual behavior and load profiles.

1. Dynamic & Continuous Adaptation

* Adjust CPU and memory resource limits at runtime based on real-time telemetry (e.g., latency, request rates, usage).
* Avoid one-time profiling or offline stress testing; instead, leverage on-the-fly observations for decision-making.

1. Online Learning Without CRPs

* Replace CRP-based static analysis (which identifies “safe-to-reduce” breakpoints) with adaptive feedback control that responds continuously to observed performance impacts.
* Learn the performance/resource tradeoff implicitly over time through interaction with the system and feedback loops.

1. Service and Multi-Stage Optimization

* Support microservices with potentially diverse performance/resource characteristics.
* Allow the system to generalize and specialize across services without manual tuning heuristics.

## Overall Strategy

Instead of relying on statically identified Critical Reduction Points (CRPs) - which are brittle, labor-intensive to compute, and specific to individual services - this strategy embraces a dynamic, learning-driven control approach. It continuously adjusts resource limits based on real-time feedback, learned performance patterns, and SLA-aware policies.

This approach centers on three pillars:

**1. Online Learning of Performance–Resource Relationships**

Rather than conducting offline profiling or defining "safe" thresholds (e.g., the lowest CPU limit before latency spikes), we let the system learn how resource allocations affect performance in real-time:

* Continuously observe metrics, including:
  + CPU and memory usage
  + CPU and memory limits
  + P95/P99 Latency
* Build lightweight predictive models (e.g., regression trees, online linear models, or streaming learners likeRiver) that estimate:
  + How changes in CPU/memory limits are likely to impact latency  
    What resource setting is likely to meet the latency target under current load
* These models are continuously retrained as new data arrives, allowing the system to adapt to workload shifts, code changes, or traffic anomalies without human intervention.

**Example:** If the model observes that reducing memory from 512Mi to 384Mi causes only a 2% latency increase during normal load, it registers this as a safe trade-off and continues testing smaller limits.

**2. SLA-Aware Dynamic Feedback Control**

This layer acts like a closed-loop controller: it makes adjustments to resource settings and observes the impact, treating the system as a black box.

* At regular intervals (e.g., every 5–10 minutes), the controller:
  + Proposes small, controlled changes (e.g., reduce CPU limit by 100m)
  + Observes resulting performance impact (especially latency)
  + Evaluates change with respect to defined SLA boundaries (e.g., "latency must remain within 20% of baseline")
* **Positive feedback loop**:
  + If latency remains within acceptable bounds, and usage stays below limit → further reduce resources.
  + If latency improves or remains stable → keep reducing in smaller steps.
* **Negative feedback loop**:
  + If latency spikes beyond the SLA threshold → revert to previous settings and reduce step size.
  + If utilization becomes dangerously close to limits (e.g., CPU throttling begins) → increase limits slightly.

This forms a hill-climbing optimization loop with built-in safety mechanisms:

* The system climbs "down" in resource usage until it hits a boundary (latency breach).
* Then it rolls back slightly and tries smaller or orthogonal adjustments (e.g., reduce memory instead of CPU).

**3. Reward-Based Control System (Reinforcement Learning Agent)**

To further enhance the system’s intelligence, a Reinforcement Learning (RL) agent can be introduced. The agent learns from trial and error over time, gradually optimizing its policy to achieve long-term goals:

* **State Space**:
  + Current CPU and memory limits
  + Current CPU and memory usage
  + Observed P95 latency
  + Request rate or concurrency
  + Historical deltas (e.g., last adjustment and result)
* **Action Space**:
  + Increase, decrease, or hold CPU/memory limits (individually or jointly)
  + Adjust in small, medium, or large steps
* **Reward Function** (carefully crafted to balance goals):
  + **Positive reward** for resource savings (smaller limits, especially if underutilized)
  + **Negative reward** for SLA violations (latency > 1.2× baseline)
  + **Moderate penalty** for being too conservative (underutilization + no cost benefit)
  + **Bonus** for stability and minimal oscillations (reducing thrashing)
* The RL agent can be implemented using frameworks like Stable-Baselines3 (PPO/DDPG agents) or Ray RLlib, and it can be:
  + **Trained in a simulated environment** (using historical data or performance models)
  + **Fine-tuned in production**, continuously improving from live feedback

Over time, the RL agent develops sophisticated, service-specific optimization policies that outperform static heuristics or hand-coded strategies.

**Example**: The RL agent may learn that during peak traffic hours, reducing CPU leads to SLA violations, but memory can be trimmed — and vice versa at night.

## Summary of Overall Strategy

By combining these three components, the system forms a self-improving, SLA-aware, CRP-free resource optimizer that:

* Adjusts limits with high granularity and safety
* Learns from its own behavior
* Improves over time, even under evolving workloads
* Eliminates the need for manually defined "safe reduction points"

This architecture mirrors human intuition: observe → act → learn → refine — but at machine speed and scale.

### **Data Collection Module**: Real-Time Observability Backbone

To enable dynamic resource optimization, our system must continuously collect key runtime metrics - even from services that aren’t natively instrumented for Prometheus. This module acts as the data foundation for all learning, feedback, and control components.

**Goals**

* Observe CPU and memory usage vs. limits
* Measure latency per request (P95 or average)
* Monitor request load (rate, concurrency)

All without requiring internal application changes or native Prometheus instrumentation.

**Core Metrics to Collect**

1. CPU and Memory Usage + Limits

What to track:

* CPU usage: in millicores (mCPU)
* Memory usage: in bytes (working set, RSS)
* CPU limit and Memory limit per container

How to collect without app instrumentation:

* Use Kubernetes-native telemetry:
  + cAdvisor (via kubelet) collects container-level usage metrics.
  + Prometheus scrapes from kubelet, kube-state-metrics, or node-exporter.

Key Prometheus metrics:

* Container\_cpu\_usage\_seconds\_total
* Container\_memory\_working\_set\_bytes
* Kube\_pod\_container\_resource\_limits\_cpu\_cores
* Kube\_pod\_container\_resource\_limits\_memory\_bytes

These allow us to:

* Detect overprovisioning (usage << limit)
* Identify throttling risks (usage ~ limit)
* Feed usage-efficiency into RL/ML reward functions

1. Latency per Request (P95 or average)

**Problem**: Our applications do not expose latency metrics.

**Solution**: Use sidecar proxies, service mesh, or ingress controllers to externally observe and emit latency metrics - no code changes required.

Approaches:

**Option A: Envoy Proxy / Istio Sidecar**

* Add Istio or standalone Envoy as a sidecar proxy to each service.
* Envoy tracks per-request latency and emits metrics in Prometheus format.

*Sample Envoy metric:*envoy\_cluster\_upstream\_rq\_time\_bucket

*Then compute latency percentiles using PromQL:* histogram\_quantile(0.95, rate(envoy\_cluster\_upstream\_rq\_time\_bucket[5m]))

**Option B: Ingress Controller (NGINX, Traefik)**

* If all external traffic flows through a single ingress (e.g., NGINX), enable its Prometheus module to export:
  + Request durations
  + Response codes
  + Request counts
* This provides coarse-grained latency and load data without touching internal services.

**Option C: OpenTelemetry Collector Sidecar**

* Deploy an OpenTelemetry collector as a sidecar alongside each app.
* It can sniff traffic and record:
  + Latency
  + Throughput
  + Errors
* Pushes data to Prometheus or another backend (e.g., Tempo, Jaeger)

1. Load Monitoring (Request Rate and Concurrency)

What to track:

* Request rate (requests per second)
* Concurrency (in-flight requests)

**Collection methods**:

I. Sidecar proxies (Envoy, Istio) again prove helpful here:

* Track how many requests per second go to each backend.
* Track how many are active simultaneously.

Example Prometheus metrics:

* rate(envoy\_cluster\_upstream\_rq\_total[1m])
* envoy\_http\_downstream\_rq\_active

Ii. Ingress metrics:

* If requests go through a single entrypoint (e.g., NGINX):
  + nginx\_ingress\_controller\_requests
  + nginx\_ingress\_controller\_request\_duration\_seconds\_bucket

These load metrics let the optimizer correlate latency/resource usage with traffic intensity, enabling intelligent scaling and adaptation.