# 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).