# 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.

#### Tooling Stack

|  |  |
| --- | --- |
| Tool | Role |
| Prometheus | Scrapes kubelet, cAdvisor, sidecars, ingress, etc. |
| Grafana | Visualizes resource/latency/load trends |
| Envoy/Istio | Captures latency/load without app code |
| OpenTelemetry Collector | Optional: collect and forward latency spans or metrics |
| kube-state-metrics | Exposes current resource limits/requests for all pods |
| Node Exporter / Kubelet | Provides node- and pod-level resource usage |

##### Summary

|  |  |  |
| --- | --- | --- |
| **Metric** | **Source** | **Description** |
| CPU/Memory Usage | cAdvisor, kubelet | Native Kubernetes metrics from nodes |
| CPU/Memory Limits | kube-state-metrics | Declared values in pod specs |
| Latency (P95/P99) | Envoy / Istio / NGINX | Observed at proxy/ingress layer |
| Request Rate | Envoy / NGINX / Traefik | Measured at entry point or sidecar |
| Concurrency | Envoy / OpenTelemetry | Active connections or spans |

To collect metrics effectively for Kubernetes resource optimization without modifying your application code, the best choice is to use a combination of Kubernetes-native telemetry + Envoy/Istio sidecars.

### Best Overall Choice (Recommended Setup)

1. **Resource Usage + Limits**

Use Kubernetes-native telemetry via Prometheus

|  |  |  |
| --- | --- | --- |
| **Metric Type** | **Source** | **Prometheus Metric** |
| CPU usage | kubelet + cAdvisor | container\_cpu\_usage\_seconds\_total |
| Memory usage | kubelet + cAdvisor | container\_memory\_working\_set\_bytes |
| CPU/Memory limits | kube-state-metrics | Kube\_pod\_container\_resource\_limits\_cpu\_cores  kube\_pod\_container\_resource\_limits\_memory\_bytes |

**How to Set Up**:

* Install **Prometheus Operator**
* Enable scraping from:
  + kubelet on each node (exposes cAdvisor metrics)
  + kube-state-metrics for resource limit specs

**Why This Works**:

* No code changes
* Works for all containers automatically
* Provides base metrics for reward shaping in RL or input features for ML

1. **Latency Metrics (P95, P99, Avg)**

Use Sidecar Proxies — Preferably Envoy via Istio

|  |  |  |
| --- | --- | --- |
| **Method** | **Best Choice** | **Why** |
| Sidecar proxy | **Envoy / Istio** | Fine-grained, per-service latency histograms |
| Ingress controller | NGINX (Optional) | Coarse-grained, only for external traffic |
| OpenTelemetry sidecar | Advanced option | Flexible, but more setup/overhead |

**How to Set Up with Istio/Envoy**:

* Deploy Istio with Prometheus scraping enabled
* Metrics auto-exposed from sidecars:
  + envoy\_cluster\_upstream\_rq\_time\_bucket
* Use PromQL for percentiles:

histogram\_quantile(0.95, rate(envoy\_cluster\_upstream\_rq\_time\_bucket[5m]))

**Why This Works**:

* High-resolution latency per service/microservice
* No app instrumentation needed
* Built-in integration with Prometheus

**3. Load Metrics (Request Rate + Concurrency)**

Use Envoy Sidecar Metrics (again)

|  |  |
| --- | --- |
| **Metric** | **Prometheus Metric** |
| Request rate | rate(envoy\_cluster\_upstream\_rq\_total[1m]) |
| Concurrency | envoy\_http\_downstream\_rq\_active |

**Alternative if using Ingress**:

* Use nginx\_ingress\_controller\_requests and related metrics.

**Why This Works**:

* Tracks load per microservice (not just per cluster)
* Real-time load insight = smarter scaling/resource decisions

### Best Stack for Metric Collection

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric Category** | **Tool/Source** | **Method** | **Why Best** |
| **CPU/Mem usage & limits** | Prometheus + kubelet + kube-state-metrics | Native Kubernetes metrics | Accurate + zero app change |
| **Latency** | Istio/Envoy Sidecars | Prometheus + PromQL | High-fidelity + per service |
| **Request rate** | Envoy/Istio | Prometheus metrics | Load-aware, fine-grained |
| **Concurrency** | Envoy/Istio | Prometheus metrics | Useful for burst handling |

Here’s a detailed explanation of why other metric collection methods are less suitable compared to the recommended setup (Kubernetes-native metrics + Envoy/Istio sidecars), especially for our adaptive resource optimization system:

**App-Level Instrumentation (Manual Prometheus Exporters)**

**Why It’s Bad:**

* **Intrusive:** Requires modifying the source code of every microservice.
* **Inconsistent metrics:** Different teams might use different libraries or formats.
* **Latency blind spots:** Developers may not expose correct P95/P99 latency or expose average only.
* **Maintenance overhead:** Every change in metric structure = redeploy + retest.

**Why It's Used Sometimes:**

* Offers internal business logic insights (e.g., queue sizes, cache hits) — but not needed for infra-level optimization like resource tuning.

**Custom Sidecar Exporters (Written by Us)**

**Why It’s Bad:**

* **Reinventing the wheel:** Tools like Envoy, Istio, and OpenTelemetry already solve this.
* **Hard to scale:** We’d need to handle logging, scraping, exporting, and aggregation logic manually.
* **Low community support:** No ecosystem or built-in dashboards; difficult to debug and extend.

**OpenTelemetry Sidecars Only**

**Why It’s Not Ideal Alone:**

* **High complexity:** Needs configuration for receivers, exporters, and pipelines.
* **Latency and throughput metrics may need manual setup:** Unlike Envoy where these are exposed by default.
* **Heavier agent:** May add more CPU/memory overhead than Envoy.
* **Requires additional observability backend:** For distributed tracing (e.g., Jaeger, Tempo), unless simplified.

**When it’s better:**

If we want full observability with traces + logs + metrics, OpenTelemetry is great — but overkill just for resource tuning.

**Ingress Controller Metrics Only (e.g., NGINX)**

**Why It’s Bad for Internal Optimization:**

* **Only tracks external traffic:** No visibility into internal service-to-service calls (e.g., microservice A → B).
* **Coarse-grained:** Metrics are not per-service. We can’t differentiate between slow services or pinpoint bottlenecks.
* **Lacks resource context:** Doesn’t correlate latency with CPU/memory usage like sidecars do.

**Logs-Based Metrics Extraction (e.g., from Fluentd/ELK)**

**Why It’s Bad:**

* **High latency:** Logs take time to ship and index.
* **Hard to query for percentiles:** Logs don’t naturally form histograms like Prometheus metrics do.
* **Expensive and heavy:** ELK stack is resource-intensive and overkill for real-time feedback loops.

**Cloud Vendor-Specific Monitoring (e.g., AWS CloudWatch, GCP Stackdriver)**

**Why It’s Bad:**

* **Vendor lock-in:** Not portable across clouds or on-prem.
* **Higher cost:** We may be billed per metric or per scrape.
* **Slower scrape intervals:** Often slower than Prometheus's 15–30s scrapes, reducing responsiveness.

### Why the Recommended Stack Wins

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature/Need** | **App Instrumentation** | **NGINX Ingress** | **OpenTelemetry** | **Istio/Envoy Sidecar** | **K8s Native + Prometheus** |
| No app code changes | No | Yes | Yes | Yes | Yes |
| Per-service latency (P95/P99) | Yes (if done well) | No | Config heavy | Yes | No (needs sidecar/mesh) |
| Internal traffic visibility | No | No | Yes | Yes | Yes |
| Low setup complexity | No | Yes | No | Yes | Yes |
| Real-time metrics for RL/ML feedback | Limited | No | Delayed | Yes | Yes |

* Istio/Envoy sidecars + Prometheus + Kubernetes-native metrics give us the best balance of visibility, non-invasiveness, and real-time monitoring.
* Avoid alternatives unless we have strong use cases (e.g., OpenTelemetry for deep tracing or app instrumentation for business metrics).

### Istio/Envoy

Istio and Envoy are related but distinct components, and together they represent one integrated method for observability and traffic management in Kubernetes.

**Here's how they relate:**

|  |  |
| --- | --- |
| Component | Role |
| Envoy | A high-performance proxy that runs as a sidecar next to each microservice. It collects telemetry, handles traffic routing, and enforces policies. |
| Istio | A service mesh control plane that manages Envoy proxies across our services. It configures and controls Envoy, and provides high-level APIs, security, and monitoring. |

**In simple terms:**

* Envoy is the data plane — it sits next to your service and collects metrics like latency, request rate, errors, etc.
* Istio is the control plane — it tells Envoy *what to do* and *how to behave*.

**We can use:**

* Just Envoy: If we want a lightweight setup and only need latency metrics and routing without full mesh features.
* Istio (with Envoy): If we want a full service mesh: secure traffic, retry policies, fine-grained telemetry, tracing, etc.

**Recommendation:**

If we only need latency metrics and load data for resource tuning, you can deploy standalone Envoy sidecars per pod.

But if we want:

* Zero-trust security (mTLS)
* Traffic mirroring
* Advanced routing (A/B testing)
* Native Prometheus/Jaeger/Grafana integrations

→ Use Istio (with Envoy underneath).

We don’t strictly need Istio/Envoy, but it is highly recommended for our project — especially given our objective:

*Dynamically tune CPU/memory limits while keeping latency within 10% of baseline, without modifying the apps.*

**Why we should use Istio (with Envoy):**

|  |  |
| --- | --- |
| **Requirement** | **Why Istio/Envoy Helps** |
| **Latency metrics without app code changes** | Envoy sidecars emit per-request latency metrics to Prometheus — no app instrumentation needed. |
| **Consistent telemetry across services** | All services, regardless of language or framework, get uniform metrics (latency, errors, request rate). |
| **Works as drop-in for all microservices** | We don't need to change application code or expose custom metrics endpoints. |
| **Provides P95/P99 latency metrics** | Built-in Prometheus-compatible histograms for upstream/downstream latency. |
| **Great for ML model input** | Rich, high-frequency metrics are essential for ML/online learning accuracy. |
| **No code changes in legacy or 3rd party services** | Critical if we’re dealing with services you can't easily modify. |

**What happens if we don’t use Istio/Envoy?**

**We would need to:**

* Instrument every microservice to expose latency metrics (e.g., via Prometheus client libraries).
* Maintain metric consistency and correctness across services.
* Write custom exporters or wrappers to capture request durations.
* Possibly lose granularity or miss real-time request traces.

Since our core innovation is around dynamic resource tuning based on live latency/load feedback, and we want to avoid static CRP thresholds, then:

Yes, we should use Istio with Envoy — it gives us the cleanest, most powerful way to observe latency and load in real time, across all services, without touching the code.

### 2. Online ML Predictor Using DARE(Dynamic Adaptive Resource Estimater) Architecture

The Online ML Predictor based on the DARE model is a continuously learning and adapting module that eliminates the need for explicitly identifying CRPs (Critical Reduction Points). Instead, it intelligently learns patterns and predicts the optimal CPU and memory resource limits based on real-time observations of load and latency — while staying within SLA boundaries.

**Primary Goals**

* Predict the "next safe step" for adjusting CPU and memory limits under the current load.
* Estimate the risk of latency degradation for each candidate resource configuration.
* Continuously learn from live traffic, gradually improving predictions.
* Adapt to changing usage patterns, workloads, and traffic profiles without retraining from scratch.

#### Architecture Overview (DARE)

The DARE-based predictor has three key components:

1. **Trend Learners (TL)**

The Trend Learners (TL) component is the foundational intelligence layer of the DARE system. Its job is to model, track, and predict how microservice resource consumption evolves over time, under varying load conditions. It doesn’t just react to spikes or drops in usage — it proactively learns temporal trends and estimates safe operating zones, which is essential for stable and efficient resource management.

Trend Learners use online regression models or Exponential Moving Average (EMA)-enhanced predictors to learn the mapping:

(Current Load Conditions) ⟶ (Resource Requirements)

This includes:

* How CPU and memory usage scale with changes in incoming request rate.
* Whether the current limits are higher or lower than needed (overprovisioning vs underprovisioning).
* How fast or slowly the resource demand is trending (e.g., is memory steadily increasing, or spiking erratically?).

### Model Types:

The TL can be implemented using:

1. **Online Linear Regression**

A continuously updating linear model such as:

CPU\_usage(t) ≈ a₁ \* RequestRate(t) + b₁

Memory\_usage(t) ≈ a₂ \* RequestRate(t) + b₂

Weights a₁, a₂, b₁, and b₂ are updated at each time step using stochastic gradient descent (SGD) or recursive least squares.

1. **EMA-Augmented Models**

To smooth noise and provide more stable predictions:

EMA\_CPU(t) = α \* CPU\_usage(t) + (1 - α) \* EMA\_CPU(t-1)

These models:

* Smooth out short-term fluctuations
* Highlight sustained trends in usage
* Are lightweight, perfect for edge or in-cluster deployment

**Why Online Regression Models?**

**1. Incremental Learning in Real Time**

Unlike batch learning models (e.g., traditional linear regression), online regression models update their parameters continuously as new data arrives. This is essential for Kubernetes environments where:

* Workloads shift frequently (e.g., due to time of day or traffic surges),
* We can’t afford to retrain models offline repeatedly,
* We need low-latency model updates.

**Why it matters**: We get a constantly adapting model that keeps pace with live system behavior.

**2. Lightweight & Efficient**

Online regressors (like SGDRegressor in scikit-learn, or River's online models) are memory-efficient, require low compute, and can run:

* On the same node as the application,
* In a sidecar container,
* Or in a lightweight central controller.

**Why it matters**: Ideal for resource-constrained or distributed environments typical of Kubernetes.

**3. Interpretability**

Online linear models are easy to interpret:

* We can see the coefficients for features like request rate or CPU usage,
* We know exactly how each metric influences predicted future usage or resource requirements.

**Why it matters**: This helps with debugging, tuning, and trust in the system — especially in production.

**4. Direction + Rate of Change (i.e., Trend)**

These models can naturally learn both:

* **Direction** of usage (↑ or ↓),
* **Speed** (e.g., 50m CPU increase per 10 rps).

**Why it matters**: It helps in proactive decisions like: “Reduce memory soon — usage is slowly dropping.”

**Why Exponential Moving Average (EMA)?**

**1. Smooths Out Noisy Metrics**

Usage data in Kubernetes (especially for latency or memory) is often spiky due to:

* GC events,
* JIT warm-ups,
* Bursty traffic.

EMA applies exponential decay, giving more weight to recent data, and less to old data:

EMA(t) = α \* x(t) + (1 - α) \* EMA(t-1)

where α is a smoothing factor.

**Why it matters**: Reduces false triggers caused by outliers or short-lived spikes.

**2. Very Low Computational Cost**

EMA is cheap to compute — just one multiplication and one addition per new data point.

**Why it matters**: Perfect for real-time systems with tight performance budgets.

**3. Captures Short-Term Trends Quickly**

We can tune α:

* A **high α** (e.g., 0.8) makes EMA sensitive to recent changes (good for reactive tuning),
* A **low α** (e.g., 0.2) makes it smoother and more stable (good for gradual trends).

**Why it matters**: We can match the EMA’s sensitivity to your desired aggressiveness in tuning.

**Why These Models?**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Online Regression Models** | **EMA-Enhanced Predictors** |
| Adaptive | Updates with each sample | Reacts to recent changes |
| Lightweight | Low CPU/memory | Extremely low overhead |
| Interpretable | Clear coefficients | Less interpretable |
| Captures trends | Learns trend slopes | Smooths and highlights |
| Handles noise | Somewhat susceptible | Very effective |
| Suitable for K8s feedback | Yes | Yes |

These models are not just simple — they’re deliberately minimal, real-time-compatible, and production-safe.

**Why Not Other models?**

It’s not that other models are inherently bad — it's that many popular ML models are not suitable for real-time, adaptive Kubernetes environments due to specific trade-offs in latency, compute cost, data requirements, interpretability, and retraining complexity.

**1. Deep Learning Models (e.g., LSTM, CNN, Transformers)**

**Why they’re problematic:**

|  |  |
| --- | --- |
| **Issue** | **Explanation** |
| **High computation** | Require GPUs or significant CPU cycles — not ideal for lightweight, real-time adaptation inside Kubernetes. |
| **Long training times** | Not suitable for continuous online learning unless we heavily engineer it. |
| **Require large datasets** | Kubernetes workloads may not generate enough data per service to justify such complexity. |
| **Opaque (Black Box)** | Difficult to interpret why a decision was made — risky in production SLAs. |
| **Overkill** | Predicting CPU/memory usage trends doesn’t require thousands of parameters. |

Use these only for long-term forecasting or batch training offline, not live tuning.

**2. Random Forests / Gradient Boosted Trees (e.g., XGBoost, LightGBM)**

**Why they’re problematic:**

|  |  |
| --- | --- |
| **Issue** | **Explanation** |
| **No true online training** | These are batch models — we need to retrain them completely for new data. |
| **Model staleness** | We can’t incrementally adapt to new trends or behaviors in live traffic. |
| **Heavy retraining cost** | Even small updates require full-tree regeneration and memory access. |
| **Latency** | Inference is fast, but updates are slow and non-incremental. |

Great for offline modeling and initial bootstrapping, but not live systems.

**3. Bayesian Models (e.g., Gaussian Processes)**

**Why they’re problematic:**

|  |  |
| --- | --- |
| **Issue** | **Explanation** |
| **Poor scalability** | Gaussian Processes scale poorly with the number of data points (O(n³)). |
| **Heavy memory footprint** | Too costly to maintain posterior distributions for every microservice. |
| **Complex implementation** | Too sophisticated for quick resource tuning tasks. |

Best used for offline exploration (e.g., Bayesian Optimization), not real-time adjustments.

**4. K-Nearest Neighbors (KNN)**

**Why they’re problematic:**

|  |  |
| --- | --- |
| **Issue** | **Explanation** |
| **No real learning** | KNN stores data but doesn’t generalize. Requires all historical data at inference. |
| **Memory heavy** | Needs to store all previous data points. |
| **Slow inference** | Each prediction requires computing distances to all stored points. |

Unsuitable for resource-constrained, latency-sensitive environments.

Why Lightweight Online Models Win?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model Type** | **Online Learning** | **Real-Time Compatible** | **Lightweight** | **Interpretability** | **Suitable for K8s?** |
| Online Linear Regression | Yes | Yes | Yes | Yes | Yes |
| EMA | Yes | Yes | Yes | Partial | Yes |
| Deep Learning | No | No | No | No | No |
| Random Forest / XGBoost | No | Fast inference | No | Yes | No |
| Gaussian Process | No | No | No | Yes (but complex) | No |
| KNN | Lazy learner | No | No | Yes (but not scalable) | No |

Just because a model is powerful doesn’t mean it’s practical. In Kubernetes environments, the *best* model is the one that:

* Learns online,
* Adapts quickly,
* Uses minimal resources, and
* Provides transparent decisions.

That’s why Online Regression + EMA is a smart, pragmatic, production-ready choice.

**Use Online Regression Alone if:**

* We want a model that learns relationships between inputs (e.g., usage, request rate) and outputs (e.g., required resources).
* We care about predictive accuracy, not just smoothing.
* We need the model to adapt trends over time, e.g., how resource needs grow with traffic.

**Best for:**

* Learning complex but linear (or piecewise linear) patterns.
* Making forward-looking predictions.
* Detecting longer-term workload drift.

**Use EMA Alone if:**

* We want a lightweight, blazingly fast, and noise-resistant trend estimator.
* We don’t need to model explicit relationships — just want to track if usage is trending up/down.
* We're running in very resource-constrained environments (e.g., edge devices, tiny sidecars).

**Best for:**

* Simple smoothing of metrics (e.g., average usage trends).
* Detecting short-term anomalies or bursts.
* Low-overhead setups where prediction quality is secondary to stability.

**Use Both Together if:**

We want to combine the strengths of both:

|  |  |
| --- | --- |
| **Layer** | **Purpose** |
| **EMA** | Smooth noisy metrics before feeding them into your model — stabilizes learning. |
| **Online Regression** | Learns meaningful relationships from the smoothed inputs. |

**Best for:**

* Production-grade adaptive systems.
* Mitigating noise while still capturing causal structure.
* Systems that face noisy metrics + dynamic traffic.

Raw CPU usage → EMA smoother → Online regressor → Predict safe CPU limit

**Recommendation Based on our Use Case**

For Kubernetes resource tuning under varying load with SLA constraints, I recommend:

Use both EMA + Online Regression:

* EMA smooths our metric stream (to avoid reacting to noise),
* Online regression maps smoothed trends to safe resource predictions.

#### Inputs:

The TL takes as input the raw usage and limit metrics, which are either collected through a Prometheus exporter or pulled directly from Kubernetes metrics APIs:

* CPU\_usage(t): Actual CPU usage in millicores
* Memory\_usage(t): Actual memory usage in MiB
* RequestRate(t): Number of incoming requests per second or minute
* CPU\_limit(t) and Memory\_limit(t): The current configured resource limits in Kubernetes

Optionally, we may also include:

* Time of day (to capture diurnal patterns)
* Recent request rate deltas (to spot load surges)

#### Outputs:

The TL provides two critical outputs:

**1. Estimated Future Usage:**

It forecasts what the CPU and memory usage are likely to be in the near future, e.g., 5–10 minutes ahead, under the assumption that the load trend continues.

This forecast supports proactive decisions like:

* “Usage is trending upward; we should avoid cutting resources now.”
* “Usage is declining and stable; we can try stepping down the limits.”

**2. Safe Resource Range Suggestions:**

Based on the forecasted usage and past adjustment outcomes, the TL outputs a range of recommended CPU and memory values:

* Minimum Safe CPU (e.g., 300m)
* Maximum Safe CPU (e.g., 500m)
* Minimum Safe Memory (e.g., 400Mi)
* Maximum Safe Memory (e.g., 600Mi)

These ranges provide flexibility for the decision engine to choose from — potentially selecting tighter or looser margins based on risk.

#### Use Case Example:

Let’s say at time t:

|  |  |
| --- | --- |
| **Metric** | **Value** |
| CPU Usage | 250m |
| CPU Limit | 500m |
| Request Rate | 300 rps |
| Memory Usage | 500Mi |
| Memory Limit | 800Mi |

The Trend Learner notices:

* CPU usage is stable and dropping by 10m per 5-minute interval
* Memory usage is slowly increasing by 15Mi per interval
* Request rate is stable

**Forecasted Next Usage:**

* CPU: 240m
* Memory: 515Mi

**Suggested Range:**

* CPU: 260–300m
* Memory: 520–600Mi

This tells the decision engine: "You’re overprovisioned. It’s safe to reduce CPU and memory limits slightly, but keep an eye on memory."

#### How TL Improves Adaptivity:

* **Learns per-microservice trends** — instead of assuming uniform behavior
* **Reduces risk of overreacting** to one-off spikes or dips
* **Enables intelligent step sizing** — instead of fixed increments, step sizes are informed by trend slopes

## Latency Risk Estimator (RE)

### Goal

Estimate the probability that a proposed CPU/memory configuration will violate latency SLAs (e.g., P95 latency exceeding 1.2× baseline), given current and past runtime conditions. This helps avoid risky downscaling decisions and improves predictive safety for resource optimizations.

### Conceptual Role in the System

The RE acts like a safety evaluator or early warning system in the optimization loop. Whenever a new CPU/memory configuration is proposed — by an online ML predictor— the RE evaluates “How risky is this change?”

### Input Features

The RE uses historical and current context to evaluate the safety of a proposed change. Key features:

|  |  |
| --- | --- |
| **Feature** | **Description** |
| ΔCPU and ΔMemory | Change in CPU/memory limit from previous step |
| CPU usage and Memory usage | Before and after the resource change |
| Request rate | Load level when the change occurred |
| Latency (P95 or P99) | Observed latency outcome |
| Latency ratio | Latency afterBaseline Latency\frac{\text{Latency after}}{\text{Baseline Latency}} |
| SLA Violation? | Boolean label: was latency > x × baseline? |

### Step-by-Step Working

**1. History Bucketization**

* Group past actions into buckets based on:
  + Load range (e.g., 200–300 rps)
  + Resource change (e.g., CPU reduced by 100m)
  + Starting limits (e.g., from 600m → 500m)
* Store outcome: was SLA violated? (1 = Yes, 0 = No)

This forms a structured memory like:

"Reducing 600m → 500m at 300 rps → SLA violated 3 out of 5 times → Risk = 0.6"

**2.Query Current Proposed Action**

* When a new action is proposed (e.g., reducing CPU by 100m under 320 rps):
  + RE searches for similar past actions in the same load & config bucket.

3.**Risk Score Computation**

* If similar past actions exist:
  + Risk Score=Total Number of Observations / Number of SLA Violations

**More formally:**

Let:

* A= a specific resource adjustment action (e.g., CPU reduced by 100m)
* L= load level (e.g., request rate bin)
* V(A,L)= number of times action A under load L resulted in SLA violation
* N(A,L)= total number of times action A was taken under load L

Then:

Risk(A,L)=V(A,L) / N(A,L)

This gives a probabilistic risk score ∈[0,1], where:

* 0.0 = completely safe historically
* 1.0 = always resulted in SLA violations

**If we want to add smoothing (Laplace smoothing to handle low sample count), we can use:**

Risk(A,L)=(V(A,L)+α) / (N(A,L)+β)

Where:

α and β are small constants (e.g., α=1, β=2) to avoid division by zero and reduce overfitting to rare events.

* If no exact matches:
  + Use interpolation or nearest-neighbor search across similar buckets.
  + If completely novel → assign a conservative default risk (e.g., 0.5).

**4. Output**

* A risk score ∈ [0, 1] indicating confidence that SLA will be violated.
  + 0 = historically safe
  + 1 = historically always fails

**Example:**

|  |  |
| --- | --- |
| **Input** | **Example Value** |
| Load | 320 rps |
| CPU change | 600m → 500m |
| Memory unchanged | 400Mi |
| RE finds | 5 similar changes: 3 caused SLA violation |
| Risk Score | 3 / 5=0.6 |

Implementation Notes

* **Backend:** Python, Go, or Rust microservice.
* **Storage:** In-memory DB (Redis), or light-weight embedded DB (SQLite) to store buckets and outcomes.
* **Update strategy:** Append-only or exponential decay (recent events weigh more).
* **Integration:** The tuner queries RE before applying a config change. If risk > threshold (e.g., 0.7), tuner either:
  + Cancels the action,
  + Reduces change magnitude,
  + Defers decision to ML model or Bayesian optimizer.

**Benefits**

|  |  |
| --- | --- |
| **Feature** | **Benefit** |
| Historical risk scoring | Avoids blindly repeating harmful configurations |
| Action-aware | Tailors risk to specific resource deltas |
| Load-aware | Understands that safe limits vary by request rate |
| Latency-aware | Prioritizes real-world SLA outcomes |
| Plug-and-play | Can be used alongside any tuner or controller |

### Decision Engine (DE) – *The Optimization Brain*

Decision Engine (DE) component, which serves as the intelligent controller at the heart of a CRP-free adaptive Kubernetes resource tuner.

#### Purpose

The Decision Engine (DE) is responsible for synthesizing insights from upstream modules — Trend Learners (TL) and Latency Risk Estimator (RE) — to make safe, efficient resource adjustment decisions in real time.

It acts as the policy layer that chooses how and when to adjust CPU and memory limits, balancing:

* Resource minimization
* Latency compliance
* SLA risk avoidance

#### Inputs

At each decision step (e.g., every N minutes), the DE receives:

1. **From Trend Learners (TL):**
   * Predicted safe ranges of CPU and memory limits for current load and usage.
   * Optional confidence intervals or slopes of usage trends.
2. **From Risk Estimator (RE):**
   * A risk score r∈[0,1] for each possible adjustment action under current load.
   * Risk scores are historical-probabilistic assessments of SLA violations for each configuration step.
3. **From Observed System State:**
   * Current CPU/memory usage and limits
   * Current P95 latency
   * Current request rate

#### Decision Logic

The DE follows a constrained optimization strategy, typically framed as: Select the lowest CPU/memory configuration within the TL's suggested range, such that the estimated risk score from RE is below a safety threshold τ (e.g., 0.2).

#### Generate Candidates with Dynamic Step Sizes

**Goal**

Construct a candidate set of CPU and memory limits that:

* Stay within the safe bounds predicted by the Trend Learner (TL)
* Use adaptive step sizes based on:
  + Latency sensitivity
  + Historical success/failure of prior steps
  + Usage trends
  + Confidence/variance in TL estimates

**Inputs Required:**

1. **Safe resource range from TL**:

|  |
| --- |
| {   "cpu\_min": 200,   "cpu\_max": 500,   "mem\_min": 256,   "mem\_max": 768 } |

**2. Confidence from TL** (optional):

* A confidence score (or variance estimate) on how accurate the trend prediction is. Higher confidence → can try larger steps.

**3. Recent delta-latency data**:

* How much latency changed for recent CPU/memory adjustments.

**4. Risk Estimator feedback**:

* Historical success/failure of past step sizes at current load.

**Step 1: Compute Sensitivity Score**

Compute how **sensitive latency is to resource changes** based on recent data:

Scpu​=∣ΔCPU∣ / ∣Δlatency∣​;         Smem​=∣ΔMemory∣ / ∣Δlatency∣​

Higher S = more sensitive → use smaller steps

**What does it mean?**

These formulas measure how much we had to change a resource to produce a given amount of latency change.

* A high sensitivity score (high Scpu​ or Smem​) means:
  + A small change in latency required a large resource change.
  + In other words, latency is highly sensitive to resource adjustments.
  + We should reduce resource changes in smaller steps (to avoid large latency spikes).
* A low sensitivity score means:
  + A small resource change caused a large latency shift.
  + The system is not very sensitive to resource adjustments.
  + We can safely make larger step changes in CPU/memory during tuning.

**Step 2: Determine Dynamic Step Sizes**

Now choose dynamic step sizes based on:

* Latency sensitivity
* TL confidence
* Historical safety

**Example heuristic:**

|  |  |  |
| --- | --- | --- |
| base\_step\_cpu = 50  # baseline base\_step\_mem = 64  # baseline  # Inverse-sensitivity scaling step\_cpu = base\_step\_cpu / (1 + S\_cpu) step\_mem = base\_step\_mem / (1 + S\_mem)  # Confidence scaling (0.0 to 1.0) step\_cpu \*= (0.5 + TL\_confidence)  # e.g., high confidence allows larger steps step\_mem \*= (0.5 + TL\_confidence)  # Risk penalty (penalize recent risky actions) step\_cpu \*= (1 - recent\_risk\_penalty) step\_mem \*= (1 - recent\_risk\_penalty)  **Step 3: Generate Candidates**  Generate candidates within TL's bounds using the computed step sizes:   |  | | --- | | candidates = [] cpu\_range = np.arange(cpu\_max, cpu\_min - step\_cpu, -step\_cpu) mem\_range = np.arange(mem\_max, mem\_min - step\_mem, -step\_mem)  for cpu in cpu\_range:     for mem in mem\_range:         candidates.append((cpu, mem)) |   Optionally, prioritize configurations near:   * 80% of previous usage * Historical best efficiency * Where risk score was lowest  Example Output  |  | | --- | | [   { "cpu": 500, "mem": 768 },   { "cpu": 450, "mem": 768 },   { "cpu": 450, "mem": 704 },   { "cpu": 400, "mem": 704 },   { "cpu": 400, "mem": 640 } ] | |

2. **Filter Safe Options:**

* For each candidate ci​∈candidates, query the RE:  
   Risk(ci​,Loadt​)<τ

If no options satisfy the constraint, relax the candidate space *conservatively* (e.g., try higher limits or add smoothing to RE).

1. **Select Optimal Adjustment:**

* From the filtered candidates, choose:



(i.e., lowest resource cost within safe zone)

1. **SLA Violation Handling:**

* If current latency Lt​>x⋅L0​, then:
  + Temporarily increase limits in small increments (e.g., +50m CPU), even if RE scores them risky.
  + Record this reversal for RE’s learning.

1. **Conservative Escalation:**

If risk scores are consistently high or TL/RE disagree, the DE defaults to higher allocations, preventing SLA breaches at the cost of short-term overprovisioning.

6. **Cool-Down Window:**

After a recent SLA breach or limit change, the DE may before making new changes, avoiding oscillations.

**Output**

The DE produces a decision action:

|  |
| --- |
| {   "service": "password-generator",   "new\_cpu\_limit": "350m",   "new\_memory\_limit": "512Mi",   "risk\_score": 0.12,   "reason": "Lowest cost within safe risk threshold" } |

This output:

* Passed to a **controller/operator** for applying via Kubernetes API.
* Logged for audit and retraining of RE.
* Visualized in dashboards for explainability.

### Post-Decision Output Handling

Once the Decision Engine (DE) selects a resource configuration (CPU and memory limits) based on inputs from the Trend Learner (TL) and Risk Estimator (RE), the selected action must be executed, tracked, and made transparent for analysis and improvement.

**1. Passed to Controller/Operator for Application via Kubernetes API**

* The selected resource configuration (e.g., cpu: 300m, memory: 512Mi) is handed off to a custom Kubernetes controller or operator.
* This controller:
  + Calls the Kubernetes API server using a client (e.g., client-go or controller-runtime).
  + Updates the target Deployment or StatefulSet with the new resource requests/limits.
  + Ensures safe rollout: e.g., waits for readiness, monitors post-deployment metrics.
* This process is idempotent and reconciles continuously — ensuring the cluster state always matches the decision engine’s output.

*Example:*

|  |  |
| --- | --- |
| resources:   requests:     cpu: "300m"     memory: "512Mi"   limits:     cpu: "400m"     memory: "512Mi"  **2. Logged for Auditing and Retraining the Risk Estimator (RE)**  Every decision made by the DE is logged along with:   * Timestamp * Microservice ID * Previous and new CPU/memory configurations * Current load (request rate) * Observed latency and risk score * Result (did it violate the SLA or not?)   These logs serve two key purposes:   1. **Auditability:** Enable engineers to trace and understand past decisions — essential for debugging and compliance. 2. **Learning Feedback:** Used to retrain or fine-tune the RE, especially if it's using historical success/failure data to estimate SLA violation risk (e.g., updating the violation probability of past actions).   *Log structure (JSON or Prometheus annotation):*   |  | | --- | | {   "timestamp": "2025-05-28T13:45:00Z",   "service": "hash-generator",   "prev\_cpu": "500m",   "new\_cpu": "400m",   "latency": "112ms",   "request\_rate": 250,   "sla\_violated": false,   "risk\_score": 0.18 } | |

**3. Visualized in Dashboards for Explainability**

To improve transparency and operator trust, decisions and their impact should be visualized in dashboards, e.g., via Grafana, Kibana, or custom UIs.

Visualizations include:

* Current vs previous CPU/memory settings
* Latency over time with SLA thresholds
* Risk score trends per service
* Resource efficiency heatmaps
* Alerts for SLA violations and reversal triggers

This explainability is critical for:

* Real-time monitoring of decisions
* Debugging unexpected behavior
* Continuous tuning and stakeholder reporting

*Example Panels in Grafana:*

* Line graph of latency (P95) vs time with overlay of resource changes
* Bar chart of risk score per action
* Timeline of controller actions (like a CI/CD pipeline)

### **Thresholds & Policies**

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Description** | **Typical Value** |
| τ | Maximum acceptable risk score | 0.2 |
| δ | Step size for resource adjustment (CPU/mem) | 25–100m |
| Ltarget | Max allowed latency (e.g., P95) | x × baseline |
| Cool-down time | Delay after adjustments before next decision | 5–15 mins |

These parameters can be configured per microservice or learned over time

To configure parameters per microservice or learn them over time, you can use a hybrid of static configuration and adaptive learning.

**Goal**

Make your optimization system adaptive per microservice, so that each one learns its own:

* Latency threshold behavior
* Safe adjustment step sizes
* Sensitivity to CPU and memory changes
* Risk tolerance

**1. Manual Configuration (Initial Phase)**

Start by statically configuring key parameters per microservice, either in a config file or via annotations:

|  |
| --- |
| annotations:   optimizer.step.cpu: "50m"   optimizer.step.memory: "128Mi"   optimizer.latency.threshold.multiplier: "1.2"   optimizer.risk.tolerance: "0.2" |

Each microservice can define:

* Initial step sizes
* Acceptable latency increase (e.g., 1.2× baseline)
* Maximum risk tolerance for SLA violations (e.g., 20%)

Store this in:

* A ConfigMap or values.yaml (Helm)
* A custom Kubernetes CRD (e.g., ResourceOptimizationProfile)
* A lightweight YAML/JSON registry used by your controller

**2. Learning Parameters Dynamically (Adaptive Phase)**

Once running, the system should **learn and tune these parameters per microservice** using observed outcomes.

**A. Online Learning of Sensitivity to Change**

For each microservice:

* Track every resource adjustment
* Observe resulting latency and utilization
* Estimate sensitivity:



Microservices with **high sensitivity** → smaller step sizes

Microservices with **low sensitivity** → larger, more aggressive steps

**B. Dynamic Step Size Tuning**

Update step sizes over time:

|  |
| --- |
| if recent\_adjustments\_were\_safe():     step\_size\_cpu \*= 1.2  # increase step else:     step\_size\_cpu \*= 0.8  # reduce step |

This gives us adaptive hill-climbing, tuned per microservice.