## Background & Motivation

Modern Kubernetes deployments often start with generously provisioned CPU and memory resources to ensure service reliability. However, this overprovisioning results in significant resource wastage, increasing operational costs without delivering proportional performance benefits. This inefficiency becomes particularly pronounced in microservice architectures, where each service may have distinct performance-resource relationships. Existing autoscaling strategies like HPA/VPA and static provisioning heuristics fail to capture the intricate latency behaviors under multi-axis resource reduction (CPU, memory, or both).

## Objective:

To design an intelligent, latency-constrained resource tuning strategy that dynamically reduces CPU and memory *requests* and *limits* for each service, starting from overprovisioned conditions. The reduction continues until the service latency increases by no more than a certain percentage from the baseline (measured under full resource availability).

## Experimental Scope

Your research involves four distinct Java and Go-based microservices under constant or semi-constant request load:

* Prime Verifier
* Echo
* Hash Generator
* Random Password Generator

Each service is evaluated under:

1. CPU-only reductions
2. Memory-only reductions
3. Combined CPU and memory reductions

Latency, CPU usage, and memory usage are tracked continuously during gradual reductions.

## Key Findings & Observations

1. **Resource Sensitivity Profiles**

      CPU-Sensitive Services

* Prime Verifier and Hash Generator, both implemented in Java and operating on computationally intensive tasks (e.g., number-theoretic checks or hashing operations), demonstrated *high sensitivity to CPU limit reductions*.
* As CPU limits were decreased, these services rapidly hit execution bottlenecks due to thread starvation and slower context switching.
* This led to abrupt and steep latency spikes, revealing that their runtime environments (especially the JVM) rely heavily on CPU headroom for efficient garbage collection, JIT compilation, and multithreading.
* These services exhibit "tight CPU coupling" - their performance scales linearly or super-linearly with available CPU until a critical point is reached.

Memory-Resilient Services

* Services such as Echo (in Go) and Password Generator (in Java) showed remarkable resilience to memory limit reductions - latency remained flat across multiple reduction steps.
* These applications tend to have lightweight memory footprints and rarely experience high allocation churn. In the case of Go services, garbage collection is efficient and predictable.
* Only when memory limits were aggressively reduced to within 5 - 10% of the service's actual memory usage did we begin to observe increased latency or OOM-adjacent behavior.

Combined Reductions: A Nonlinear Story

* When CPU and memory were reduced simultaneously, certain services - especially those running on the JVM - exhibited nonlinear performance degradation.
* For instance, Hash Generator began showing GC pauses and stuttering behavior even when memory usage was below limits, due to synchronized pressure on heap space and compute capacity.
* This confirms that resource reduction effects are not additive - they interact in complex ways, especially under concurrent load and runtime overheads.

1. **Latency Behavior Dynamics**

Cliff-Edge Phenomenon

* A consistent observation across all services was the emergence of latency cliffs - a phenomenon where latency remains stable through gradual reductions but suddenly increases drastically when passing a Critical Reduction Point (CRP).
* This behavior is akin to a nonlinear tipping point, often hidden in CPU scheduling, garbage collection frequency, or cache eviction behaviors.
* The cliff indicates a fragile zone where performance collapses with even a small further reduction. These zones vary per service and must be empirically discovered.

Latency-Resilient Curve

* When plotting latency against resource limits, services like Echo exhibited a flat latency-resilience curve, indicating strong tolerance to resource reduction. These services operate well below bottleneck thresholds, possibly due to I/O-bound workloads or stateless design.
* Conversely, Hash Generator showed a steep curve with an early inflection point - an indicator that its workload is highly sensitive to resource throttling, likely due to algorithmic and JVM-specific behavior (e.g., memory allocation rate, GC intervals).

1. **Resource Usage Patterns**

Underutilized Baselines

* Initially, all services were deployed with generous CPU and memory requests/limits, far exceeding actual usage under steady-state load.
* This led to resource waste, as most services maintained low utilization ratios (40–60% CPU, <50% memory).
* These baselines serve as the reference points for controlled resource reduction experiments and reveal the potential for aggressive optimization.

Buffered Memory Usage

* Especially in Go-based services (e.g., Echo), memory usage was observed to be remarkably flat even as limits were reduced.
* This is attributed to the Go runtime's adaptive memory allocator and incremental garbage collection, which maintain low residency despite reductions.
* However, once memory limits approached actual usage (~95%), even minor drops led to latency spikes or runtime warnings, indicating internal GC stress or heap resizing delays.
* Java services, in contrast, showed more volatility due to GC-induced memory churn as the heap shrinks under constrained environments.

## Research Challenges

1. Dynamic Profiling of Critical Reduction Points (CRPs)

* One of the core difficulties in resource optimization under latency constraints is the need to accurately identify the Critical Reduction Point (CRP) - the threshold at which further reduction in CPU or memory resources causes a nonlinear degradation in service latency or reliability.
* Why it’s challenging:
  + Nonlinearity and Variability: The location of the CRP is not fixed and may vary across deployments, workloads, time-of-day patterns, and runtime states (e.g., warmed-up vs cold JVM).
  + Service-Specific Behavior: Each service - depending on its language runtime (Java, Go), concurrency model, and workload (CPU-bound, memory-bound, I/O-bound) - has unique CRP characteristics.
  + Noise and Volatility: Identifying a CRP often requires observing trends amid transient fluctuations in latency, which can be noisy due to background GC, load bursts, or OS scheduling quirks.
  + Minimizing Disruption: Profiling must be done dynamically and in-place - we must extract sensitivity information without restarting or heavily disrupting production services.
* Effective CRP detection demands online, fine-grained telemetry combined with adaptive probing algorithms that reduce limits incrementally while continually checking SLA compliance.
* Purely static heuristics or offline benchmarks are insufficient.