Dynamic Resource Allocation for Microservice Applications

Sriyash Caculo, Vishal Rao

What is this work trying to solve?

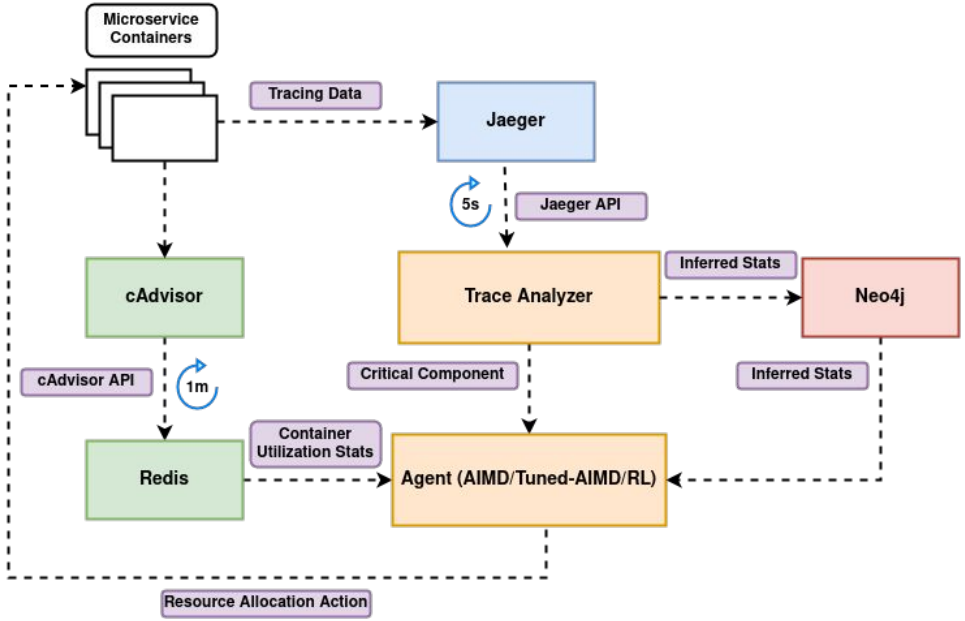
- Microservices have sub-ms SLO requirements
- There is constant interference on servers as they are often servicing multiple application
- The microservice bottlenecks are constantly changing

∴ *We need to monitor the application and constantly react in real time.*

Approach: efficiently allocate resources based on whether the application is violating it's SLO in real time

The Instrumentation Infrastructure

- Jaeger
  - Traces packets through the application
- cAdvisor
  - Monitors container utilization metrics
  - API queried and data stored in Redis
- Neo4j
  - Stores inferred metrics (Ex. SLO-Retainment)



Critical Component Extraction

What is the Critical Path?

The path on which the packet spends the most amount of time.

What is the Critical Component?

The microservice which if scaled up, can improve performance and prevent the current SLO violation.

Metrics to find the Critical Component

Relative Intensity

PCC between the Critical Path Time and the time spent in each microservice on that path

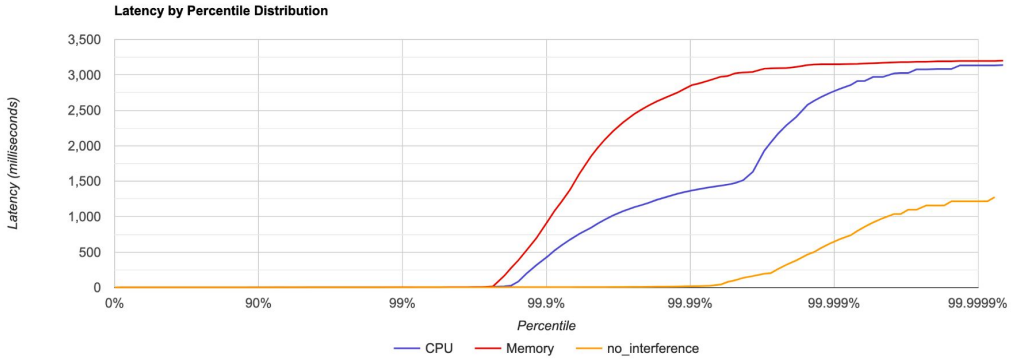
Congestion Intensity

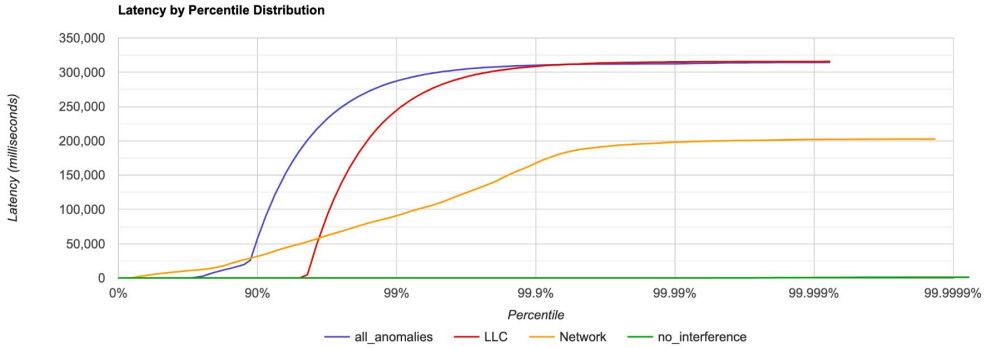
(P99 Latency/P50 Latency) of the time spent performing an operation in a microservice

Anomaly Injection

The injected anomalies artificially create resource scarcity and contention.

Network and LLC anomalies seem to cause a larger spike than the CPU and memory anomalies.





SLO Mitigation Mechanism

Once the critical component is identified, one of the following policies is used to mitigate the violation by provisioning more CPU-shares to that microservice.

- Vanilla AIMD
  - SLO Violation → 2x CPU-Share ; Else → -1 CPU-Share
- Tuned AIMD
  - SLO violation → Allocate CPU-shares in proportion to the extent of the violation; Else → -1 CPU-Share
- RL Agent
  - Learn to allocate resources based on the SLO retainment, CPU Utilization and Rate Ratio

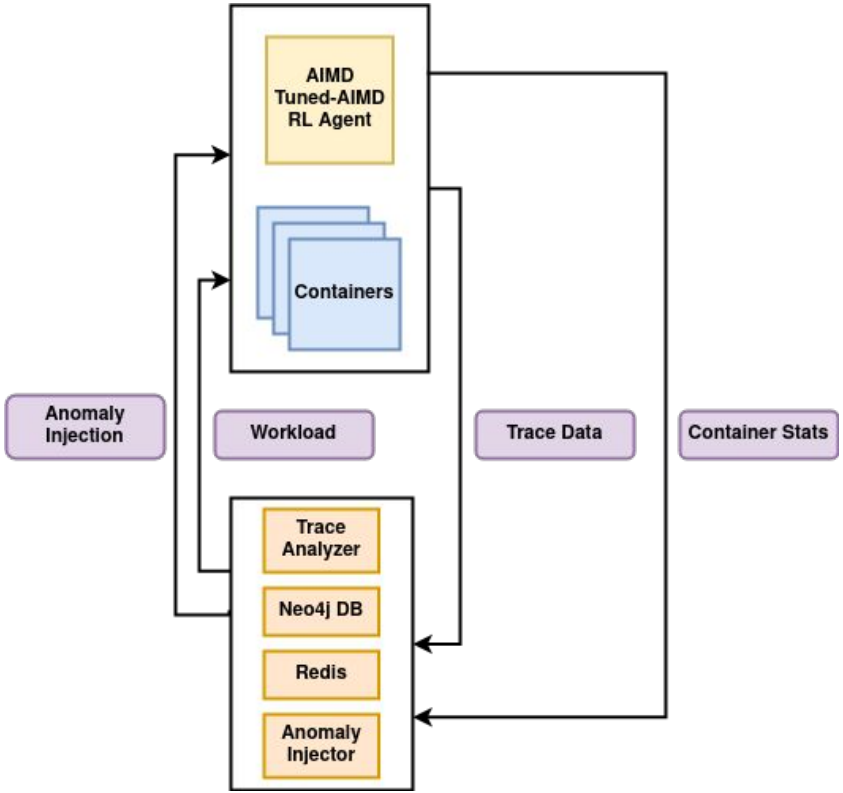
Experimental Setup

Hardware on both nodes -

- 20-Core Skylake C620
- 187 GB DDR4 RAM
- 640 KB L1; 10 MB L2 and 13 MB L3 Caches

Constant 10-minute workload of 300 Requests/Second

Each container is allocated a baseline of 2 CPU-Shares

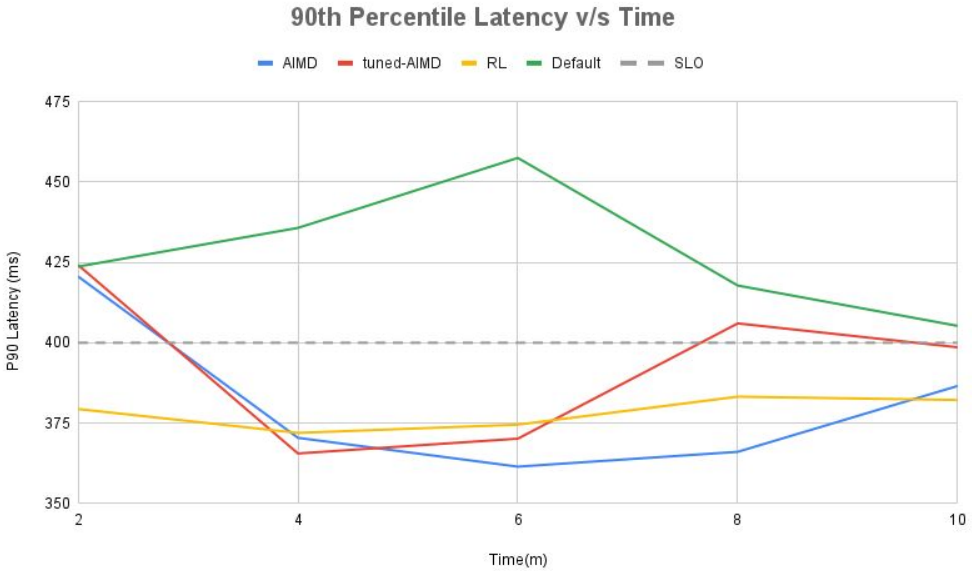


SLO Mitigation in Action

Observations

- Default is complete a function of the anomaly injector and the resultant interference.

- Tuned-AIMD is closest to SLO Latency
- RL has almost constant latency
- All policies have comparable latencies



How does the Adaptive AIMD work?

- When an SLO violation occurs, The critical component needs to be scaled up
- Allocate CPU-share as a function of SLO-Retainment
    - Lower SLO-Retainment → More CPU-share (SLO-Retainment = SLO-Latency/Current-Latency)

Compared to Vanilla AIMD -

- It uses ~11.5% lesser CPU-shares. (*Least Among the 3 Policies*)
- It will require lesser re-allocations.
- It will converge to a stable allocation faster.

Understanding the RL Agent Plot

Trained for approx 8 hrs.

The reward mechanics of the RL Agent depends on -

- SLO Retainment
- Current CPU Utilization
- Rate Ratio

We see in the results that it maintains the lowest latency. This is because during training, it found the of allocation sweet spot (60% of Max CPU-Shares for the 2 most common critical components).

(The loss function is pretty low at this allocation)

It allocates ~10% lesser CPU shares than vanilla AIMD

Possible Improvements

- Train RL agent longer - May result in more efficient utilization of resources.
- Inject multiple anomalies instead of just the CPU anomaly to reduce predictability
- Support dynamic allocation of all resources.

Key Takeaways

- Real Time monitoring is hard but extremely useful if tuned properly.

- RL does not necessarily need to be used in this use-case.
  - Finely tuned policies can offer similar performance at lower computational costs.
  - These policies also do not require pre-trained models