Dynamic Resource Allocation for Microservice Applications

April 24, 2022

Sriyash Caculo • Vishal Rao

---

1 Project Motivation

Due to the many advantages, the most popular internet-based services such as Netflix [3] and Amazon [2] have adopted the microservice-based architecture. These microservice-based applications are deployed on a large number of servers to take advantage of the resources provided by a cluster of hosts. The SLOs for microservice-based applications are very strict, requiring each microservice to conform to sub-ms latencies and resource contentions can cause latency spikes that can violate these SLOs. Thus, strategic resource allocation to microservice application becomes very important. We focus on the framework, FIRM [4], developed to automate the task of resource management. The high-level milestones that we originally set out with are listed below.

1. Bring up a functional instance of FIRM.
2. Attempt to reproduce the results obtained for the AIMD policy as a sanity check.
3. Hypothesise a better policy that would improve performance.
4. Tweak code to see if the new policy is valid and improves Performance.˘
5. Compare the policy's performance and that of the RL agent used by FIRM.

2 Approach

FIRM requires setup of kubernetes for container orchestration; a microservice application (social network) and a workload (compose-post) to measure performance; a tracing framework (tracer-grapher, cAdvisor, Prometheus) to capture microservice latencies, execution paths and system utilization statistics; a load generator (wrk2) to inject load onto the microservice application; an anomaly injector that artificially creates resource scarcity and contentions; an SVM for SLO detection critical path extraction; and finally an RL agent for SLO violation mitigation.

Initial effort was directed towards setting up the existing FIRM repository. However, this effort was abandoned due to the complexity of the codebase. The codebase was then used as reference to build our own instrumentation infrastructure as well as the resource allocation mechanisms. We use docker instead of kubernetes and implement our own pipeline to mimic that of tracer-grapher[1]. We still use cAdvisor to monitor container statistics.

2.1 Tracing Framework

The tracing infrastructure shown in Fig.1 has three parts - The Jaeger Tracing Framework, The cAdvisor per-container metrics collector and the Neo4j database which is used to store inferred metrics in memory. The social network application's codebase is instrumented with Jaeger which traces packets as they move through the different modules. Jaeger collates and reports the amount of time spent by the packet in each operation of every microservice.

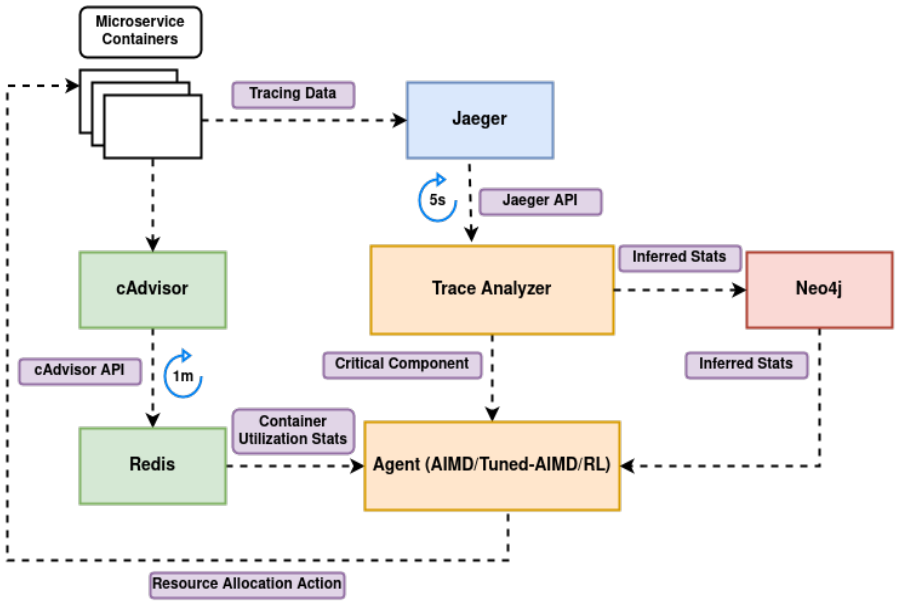


Figure 1: Instrumentation Infrastructure

The traces generated by Jaeger can be queried using an API and we continuously query a 20 second window of traces so that we can spot the recent trends. This helps us to figure out the potential microservices that should be scaled up in order to mitigate SLO violations. To quantify the SLO violation, we use SLO-Retainment value, which is the ratio between the current P90 latency and the required P90 latency. If this ratio is greater than 1, it means that there is an SLO violation. The traces sampled in the last 20-second window are aggregated to compute to current P90 latency and detect SLO violations. Using the traces, the SLO-refinement value, request arrival rate and rate ratio (`curr_arrival_rate/prev_arrival_rate`) are calculated for each 20s window and these values are written to the Neo4j database every 5 seconds.

Google's cAdvisor is a container (and host) monitoring tool which monitors the resource utilization statistics of every container and exposes them through an API. This data is queried every minute and stored into a Redis database which is queried by the RL agent to aid in deciding what actions to take. The resource consumption of the host is also reported by cAdvisor and stored in Redis.

2.2 Critical Component Extraction

We use Critical Component Extraction to find microservice candidates which, if scaled up, could potentially mitigate the current SLO violation. This is done in real time as these candidates can change depending on the previous scale up decision and the nature of the workload. We use the two metrics to find potential candidates -

- **Congestion Intensity**: The ratio of the P99 Latency and P50 Latency of the time spent by a packet while performing an operation in a microservice. If this ratio is high, it means that there is a high amount of congestion and allocating more resources to this microservice may benefit performance.

- **Relative Importance**: The critical path is defined as the path on which the packet spends most of the time. We infer this path across all sampled requests in that last 20 seconds and compute the Pearson Correlation Coefficient (PCC) between the Critical Path Time and the time spent in each microservice along this path. This is used to find the contribution each microservice makes towards explained the total variance of the critical path latency across the last 20 seconds. The larger the PCC, the more that microservice contributes to the total CP variance. Hence, allocating more resources to this microservice may benefit performance.

The FIRM approach uses an SVM to classify microservices as Critical Components based on these two metrics but we decided to use a weighted sum of these two factors as the codebase was slightly complex.

2.3 SLO mitigation mechanism

For critical components with SLO violations we allocated resources using one of three mechanisms every 5 seconds. These are: AIMD, tuned-AIMD and an RL mechanism.

- **AIMD**: In this approach, a naive AIMD algorithm is used to allocate and deallocate resource. If an SLO violation is detected, the CPU allocated to the critical component is doubled, else a finite amount of resource is taken away.

- **Tuned-AIMD**: This approach is an optimization over the vanilla AIMD mechanism. Resources are allocated on an SLO violation in proportion to the extent of the SLO violation. Hence this is a more conservative allocation approach that results in more efficient usage of resources.

- **RL**: This approach uses an RL agent that is trained to learn how to allocate resources to the critical component based on the SLO-Retainment value, the rate ratio and the container and host utilization stats.

3 Challenges

- **Source code setup and dependency management**

  We faced many issues with the compilation of the source code and the setting up of the FIRM framework which is why we decided to create our own pipeline to collect and compute all the statistics.

- **Tuning the Jaeger Polling Frequency and Look-back time**

  Jaeger requires a little time to process the traces it collects so while querying the API we had to figure out the minimum amount of time from when we could start looking back. If we picked a sooner time, the data processing would be incomplete and the query to return erroneous data. If we looked back too far, we would we reacting to traces that happened too far back and loose out on accuracy, possibly computing a stale SLO-retainment value.

We also had to tweak the API polling frequency. If the frequency was too high, then there would be too much pressure on the network and if it was too loo, we would again be susceptible to computing stale values.

- **Memory bandwidth and cache allocation problems**

  Intel's CAT allows customization in the amount of memory bandwidth and cache available to a given CPU. Our goal was to allocate these resources to individual microservices. Since microservices are not pinned to a set of CPUs, it was unclear how memory and cache could be partitioned among microservices. Thus, we decide to only inject CPU anomalies and work with CPU allocations to mitigate SLO violations

4 Evaluation

4.1 Experimental Setup

We setup a two-node server cluster on CloudLab for conducting our experiments. The setup required specific hardware (c220g5 nodes) that had support for Intel Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) Technology. The workload generator, anomaly injector, Redis DB and Neo4j DB were deployed on Node 0 as we did not want these to interfere with the application and resource allocation agents which were deployed on Node 1.

4.2 Anomaly Injection

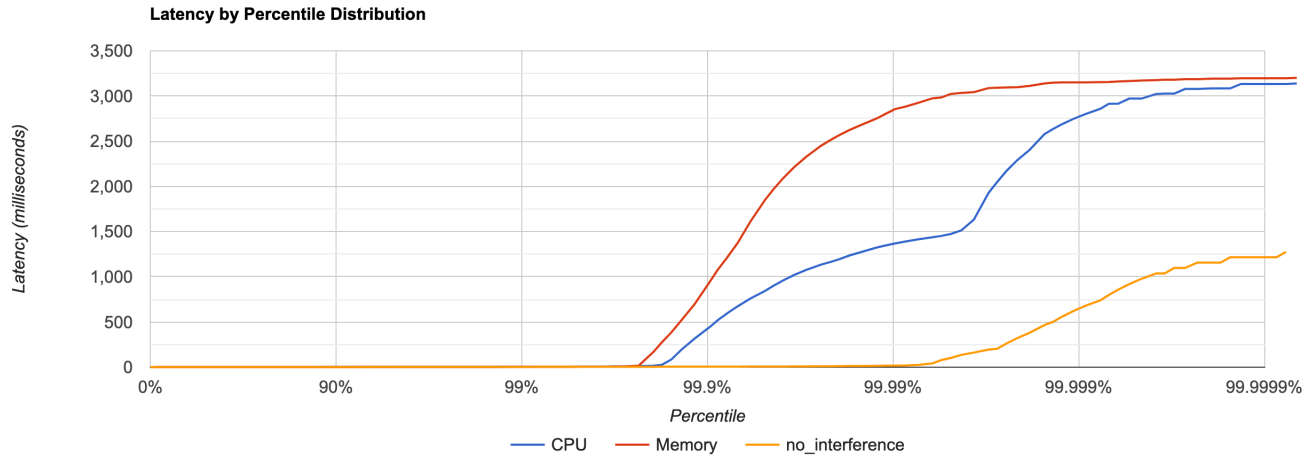
We have deployed the social-network workload, are able to generate load to it using wrk2 and inject anomalies using the anomaly injector.

We used the performance anomaly injector to inject the following types of anomalies:

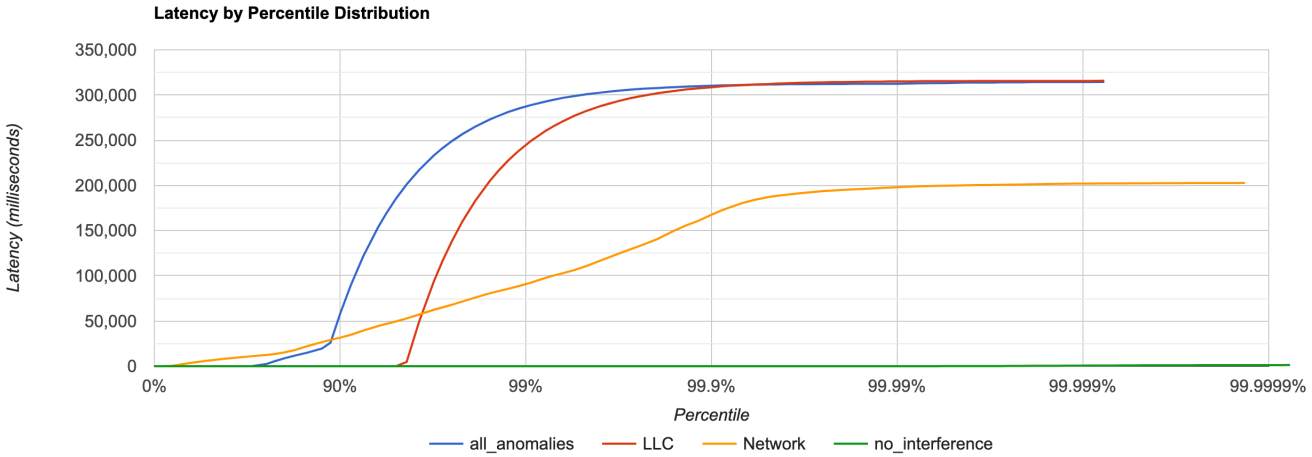
- **CPU utilization.** CPU stressor test that exhausts a specified level of CPU utilization.
- **Memory bandwidth.** Generates memory bandwidth contention.
- **LLC bandwidth and capacity.** Performs capacity and bandwidth interference using streaming accesses and intensity of accesses respectively.
- **Network bandwidth.** Generates ingress and egress traffic to create network interference.

To examine the working of the anomaly injector, we plot the latency distribution reported by the workload on injecting anomalies versus baseline. The results are plotted as a Cumulative Distribution Function (CDF) in Figures 2a and 2b. In Figure 2a, we note that on injecting CPU and Memory anomalies, a spike in latency is observed at very tail end of the CDF at around 99.9%. The LLC and Network anomalies on the other hand, introduce far greater interference as seen in Figure 2b. Additionally, we run a script that randomly picks and injects anomalies, the results of which are also plotted in Figure 2b and labeled as 'all anomalies'. We observe that up to the 90% mark, the 'all anomalies' CDF shows a slightly better trend than the Network CDF. We hypothesize that since random anomalies are injected, the 'all anomalies' run need not perform as bad as the worst of the

rest. Beyond the 90% mark, the 'all anomalies' run has the highest latency when compared with the rest.



(a) Percentile Distribution for the CPU and Memory anomalies



(b) Percentile Distribution for the LLC and Network anomalies

Figure 2: Latency CDFs for different Anomalies

4.3 SLO Violation Mitigation

While evaluating the performance of the different SLO mitigation techniques, we generated a constant workload of 300 Requests/second for 10 minutes and recorded latencies at every 2 minute interval. We used 4 different configurations - Default, AIMD, tuned-AIMD and RL. We determined that an SLO of 400ms for the P90 latency was reasonable in the presence of the CPU anomaly, which was injected for approximately injected for 2s every 10s. For all the experiments we only alter the CPU allocation policy as explained in §3. Each container is allocated 2 CPU-Shares by default and is scaled up according to the different policies. The resulting P90 latencies are plotted in Fig. 3. The gray dashed line represents the SLO latency.

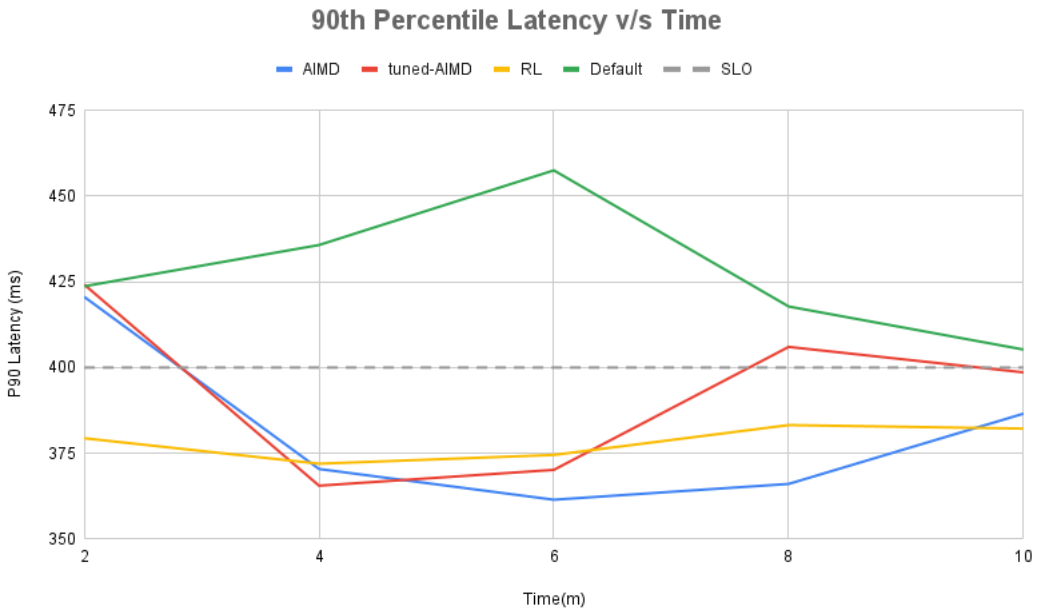


Figure 3: P90 Latencies over the course of the workload

Default

The is the baseline, where each container is given 2 CPUs and the anomalies are injected at random. We see the graph has no visible trend. The variance in the graph is completely attributed to the anomalies injected and the resulting interference.

AIMD

This plot represents the trend of the naive AIMD policy explained in §2.3 which doubles the CPU-share allocated the critical component when an SLO violation is detected and takes away 1 CPU-share if there is no violation. This policy comfortably maintains the P90 Latency below the SLO at the cost of over-allocation of resources. We observed that by the end of the 10 minute run, the two microservices that were most commonly identified as the critical components had been allocated the maximum possible CPU-share. The average CPU-shares allocated per container at the end of the 10 minute run was ≈ 2.967 Therefore, we developed the adaptive AIMD policy described below.

Tuned-AIMD

This policy, as explained in §2.3, allocates CPU to the critical component in proportion the extent of the SLO violation. We observe that this plot takes has a higher variance than the RL agent but still maintains the latency at an acceptable level. The benefit of this policy was that it allocated the least amount of CPUs among all three policies (11.5% lesser than vanilla AIMD). The CPU-share to be allocated was calculated as a function of the SLO-retainment value we see that it comes very close to the SLO latency. This policy allocated the maximum possible CPU-share to the most common critical component and allocated only 1 more CPU-share than the baseline to the second most commonly inferred critical component. The average CPU-shares allocated per container at the end of the 10 minute run was ≈ 2.63, which is an improvement over the vanilla AIMD considering the fact that

there are 30 containers. We would also like to point out that it will require lesser re-allocations than vanilla AIMD and will converge to a stable allocation faster.

RL Agent

The RL agent was trained for roughly 8 hours and did a very good job of maintaining the latency below the SLO. The reward was calculated based on the CPU utilization, SLO-retainment and rate ratio. We observed that at the end of training, since we factor in CPU utilization and only test of workload, the RL agent was able to find a sweet spot of allocating 60% of the maximum CPU share to the two most commonly inferred critical components to minimize it's loss functions. This is why we saw that while testing, the agent starts off by performing this allocation and not making any changes as it has mitigated the SLO violation. We feel like if we had trained for longer, it may have slowly started taking away CPU-shares as well but did not have the time to do so. The average CPU-shares allocated per container at the end of the 10 minute run was ≈ 2.67 (10% lesser than vanilla AIMD).

5 Progress Summary

In the last update, we had setup the social network application, the workload generator and the anomaly injector. We had profiled the workload in the presence of different anomalies and were working on the tracing pipeline. In light of the complexity of performing the pipeline setup, we decided to bring up our own tracing pipeline.

The present status of the work is as follows -

- We have successfully developed our own pipeline that mimics the one described in paper and have used the FIRM codebase as our main reference.
- We have built a dynamic critical component extractor which will work for any application instrumented with Jaeger.
- We have implemented an adaptive AIMD policy, by making a simple change, that is more efficient than the vanilla AIMD in terms of resource efficiency.
- We have also been able to retrofit the RL agent of the FIRM paper with our tracing pipeline, although it only performs CPU-specific allocation decisions.

Key Takeaways

A couple of takeaways we have from this project are -

- There is a need to implement low-overhead efficient Real-Time monitoring frameworks as the bottlenecks constantly change in real-time and we need a way to react to the SLO violations as soon as possible.
- RL is not neccessary for this use-case. Finely tuned policies can offer similar performance at lower computational costs. These policies also do not require pre-trained models and will consume lesser resources as compared to a model that needs to perform an action after it has been trained.

- Fine-Tuning of the monitoring framework is extremely important as it can help optimize the pipeline in terms of resource requirements as well as reaction-times.

6 Division of Work

- Vishal : Worked on the bring up of the tracing infrastructure and metrics collection (including Jaeger, cAdvisor, Neo4j). Developed the critical path and critical component extraction algorithms. Retrofitted the RL based SLO mitigation mechanism into the tracing framework using the FIRM codebase as reference.
- Sriyash : Worked on the debugging the social network's compose-post workload from DeathStarBench. Developed the anomaly injection and resource (CPU) allocation scripts. Implemented the AIMD SLO-mitigation mechanism as described in the FIRM repository and implemented the tuned-AIMD policy.

Both Sriyash and Vishal worked on the final tuning of the complete pipeline before obtaining the final metrics as we both were very interested in the complete workflow.

References

[1] Tracer-grapher, https://github.com/clement-casse/trace-grapher.

[2] Staci Kramer. Gigaom—the biggest thing amazon got right: The platform, 2011.

[3] Tony Mauro. Adopting microservices at netflix: Lessons for architectural design. *Recuperado de https://www/. nginx. com/blog/microservices-at-netflix-architectural-best-practices*, 2015.

[4] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. {FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 805–825, 2020.