AutoLimit - A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices

Hasan Masum - 0424052018
Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology
masumbuetcse18@gmail.com

Mahdee Mushfique Kamal - 0424052064
Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology
mahdee.m.kamal@gmail.com

***Abstract*—The efficient management of resources in microservices while maintaining Service Level Objectives (SLOs) is crucial for modern applications. Overprovisioning, while reducing the risk of SLO violations, often leads to resource wastage and increased operational costs. This paper presents a bi-level resource management strategy, extending the work of Autothrottle, to address these challenges by optimizing memory allocation while meeting SLOs. At the microservice level, we introduce Captain, a lightweight resource controller that adjusts CPU allocations based on CPU throttles, a metric highly correlated with latency. At the higher level, Tower, the centralized feedback controller, dynamically sets performance targets using a contextual bandit algorithm, adjusting for varying workloads. On each Captain, we set heuristic-based rules to scale up or scale down the memory limit of services. Through this optimization, we reduce the memory limit by 50.6% - 60.3%. Our contribution aims to reduce resource wastage while ensuring SLO compliance, providing an adaptive, scalable solution for resource management in SLO-targeted microservices.**

***Index Terms*—SLO, Resource Management, CPU Throttle Ratio, Memory Limit**

I. Introduction

Resource management of SLO-targeted microservices is an important topic of research in distributed computing systems. A microservice is an architectural style that structures an application as a collection of small, autonomous services, each designed to handle a specific business function. These services communicate with each other using lightweight protocols, often over a network. Over the past few years, there has been a significant shift from monolithic architectures to microservices, driven by their ability to support flexible, scalable, and resilient application designs [1]–[5]. By decoupling functionalities, microservices enable teams to develop, deploy, and scale different parts of an application independently.

To ensure a seamless end-user experience, many latency-sensitive, user-facing applications impose Service Level Objectives (SLOs). An SLO is a specific, measurable target that defines the expected level of service performance, such as response time, availability, or throughput, often agreed upon between service providers and consumers. However, SLO violations can lead to poor user experiences, reduced customer satisfaction, and potential financial or reputational losses for service providers. These violations often arise due to inadequate resource allocation, unpredictable workloads, or failures in the underlying infrastructure.

To mitigate the risk of SLO violations, many cloud providers and organizations resort to overprovisioning their infrastructure. This approach involves allocating significantly more resources—such as CPU, memory, and storage—than what is typically required to handle peak demand or unexpected workload spikes. While this strategy reduces the likelihood of SLO breaches, it comes at a substantial cost.

The primary issue with overprovisioning is resource wastage. Allocating idle or underutilized resources leads to inefficiencies, increasing operational costs without delivering proportional benefits [6], [7]. For instance, servers or virtual machines often operate at a fraction of their capacity during non-peak hours, contributing to unnecessary energy consumption and maintenance costs. Previous studies have shown that substantial savings can be achieved by harvesting or reclaiming excess resources for co-located applications in multi-tenant environments [8]–[13].

This research seeks to address these challenges by proposing strategies to minimize resource usage while avoiding SLO violations. By optimizing resource allocation and management, we aim to improve the efficiency and sustainability of microservice-based distributed systems.

Previous efforts have proposed various strategies to address these challenges. For example, Sinan [14] employs a machine learning (ML)-based approach to achieve QoS-aware resource management for cloud microservices, showcasing the potential of ML techniques to dynamically allocate resources while maintaining service-level objectives. Adaptive resource-efficient deployments across the cloud-edge continuum have been studied in works like Fu et al. [15]. Similarly, Zhou et al. [16] and Wen et al. [17] focus on QoS-aware and SLO-aware resource configurations for serverless workflows. Other notable contributions include ERMS [18], which emphasizes efficient resource management with SLA guarantees, and GRAF [19], which utilizes graph neural networks for proac-

We are thankful to Dr. Abdullah Adnan Sir for his guidance.

tive resource allocation. Furthermore, autoscaling mechanisms with QoS assurance, as demonstrated by Hossen et al. [20], and fine-grained intelligent frameworks such as FIRM [21] provide insights into optimizing microservice performance.

The gaps in state-of-the-art research are as follows. *ML-based solutions* have gained attention due to their ability to dynamically adjust resource allocations based on patterns learned from data. However, they come with their own set of challenges:

- *Training and retraining costs* are substantial, especially for complex applications. For instance, training a model for a 28-microservice application can take over *14 hours*, making it impractical for real-time use.
- There is *no strong correlation* between end-to-end system performance and per-service resource usage, making it difficult to accurately predict and optimize overall system behavior based on individual service metrics.

As a result, solutions based on Boosted Trees, Convolutional Neural Networks, or Graph Neural Networks remain ineffective, highlighting the necessity for a lightweight online algorithm that requires less training time. On the other hand, existing solutions for resource management in microservices can be broadly categorized into two types:

- *Centralized control with a global view of service topology*, where a central controller manages resources across all microservices based on the complete topology of the system. While this approach can offer comprehensive management, it often introduces scalability challenges and lacks adaptability to dynamic, real-time conditions.
- *Per-service heuristics with operator-defined rules*, which rely on predefined strategies for managing individual services. These solutions often fail to adapt to evolving workloads or complex service interactions, leading to suboptimal resource allocation.

These gaps highlight the need for more efficient, adaptive resource management strategies that can overcome the limitations of both centralized control and per-service heuristics-based approaches.

To address this issue, Wang et al. introduced Autothrottle, a practical bi-level tool to manage resources for SLO-targeted microservices. Autothrottle introduces an innovative approach to resource management for SLO-targeted microservices by combining local and centralized control mechanisms. The primary contribution of Autothrottle lies in its use of a *bi-level resource management strategy*, which efficiently balances the need for real-time, low-overhead adjustments with the goal of meeting end-to-end SLOs. At the microservice level, *Captain*, the lightweight resource controller, ensures that each microservice adheres to its performance target by adjusting CPU allocations through OS APIs, such as CPU quotas in Linux's cgroups. This fine-grained, local control is based on an unconventional metric—*CPU throttles*, which represent the number of times a service exceeds its CPU quota within a given time period. By tracking CPU throttles, Autothrottle benefits from a metric that is both inexpensive to sample at high frequency and highly correlated with latency, making it an effective proxy for SLO compliance. These characteristics enable *Captain* to make timely, cost-effective adjustments without introducing significant overhead. On a higher level, *Tower*, the centralized SLO feedback controller, aggregates information from across the application and learns to identify optimal performance targets. Using a *contextual bandits* algorithm, *Tower* dynamically determines the most cost-effective performance targets that satisfy the SLO, adjusting for varying workloads measured by requests per second (RPS). This centralized learning mechanism allows Autothrottle to adapt to changing application demands while minimizing resource wastage. Overall, the contribution of Autothrottle is twofold: first, it introduces a highly efficient and scalable method for managing resource allocation at the microservice level, and second, it integrates a lightweight online learning algorithm to optimize performance targets for the entire system, ensuring that SLOs are met in a cost-effective manner.

However, Autothrottle doesn't manage memory allocation. In their implementation, a single service can consume as much memory as possible. To address this issue, we extend the work of Autothrottle. We implement a heuristic-based memory management approach. Here, given a target *memory ratio* and the *Captain* auto-scales to conserve it's memory limit. This optimization achieves a memory limit reduction of 50.6% to 72.3%.

The rest of the paper is organized as follows. In Section II, we discuss the work of commonly used or prominent works in detail. In Section III, we provide the methodology and implementation of our work. In Section IV, we discuss the results. Finally, Section V presents the conclusion.

II. Background and Related Work

In this section, we review the methodologies and implementations of several widely used tools and frameworks for resource management in microservices, focusing on Kubernetes AutoScaler, the machine learning-based tool Sinan, and the bi-level tool Autothrottle.

A. Kubernetes Default Autoscalers

Kubernetes offers a default autoscaling mechanism for managing resource allocation across microservices. The K8s-CPU autoscaler locally maintains the average CPU utilization for each service, using a user-specified CPU utilization threshold (e.g., 50%) to determine when to adjust resource allocation. The autoscaler works by periodically measuring each service's CPU utilization at fixed intervals.

In the standard K8s-CPU configuration, the autoscaler measures CPU utilization every 15 seconds (denoted as $m = 15$ seconds). If the CPU utilization exceeds the threshold, the system adjusts the CPU limit by setting it to the largest value allocated in the last 300 seconds (denoted as $s = 300$ seconds), thus ensuring that the service has sufficient resources to handle high demand periods. This approach, while effective for basic CPU scaling, may not capture rapid fluctuations in resource demands and can lead to overprovisioning, especially when workloads change unexpectedly.

A faster variant, K8s-CPU-Fast, reduces the measurement period to $m = 1$ second and shortens the allocation window to $s = 20$ seconds. This modification allows the autoscaler to respond more rapidly to changes in CPU usage, enabling more dynamic scaling. However, the increased frequency of measurements and the shorter time window for allocation also introduce the potential for more frequent and possibly unnecessary resource adjustments, which could impact system performance and efficiency.

While the K8s-CPU autoscalers provide basic resource scaling, they rely heavily on fixed thresholds and historical averages, which may not always align with the dynamic nature of modern microservice workloads. Additionally, they primarily focus on CPU utilization, potentially overlooking other critical factors such as memory usage, latency, or application-specific SLOs. Consequently, these default autoscalers may not be sufficient for complex, latency-sensitive microservices that require more nuanced resource management strategies.

In memory-based autoscaling, the system checks the average memory utilization of all the pods in a deployment and increases or decreases the number of replicas based on the utilization levels. If the average memory utilization is above a specified threshold, the system will add more replicas to handle the increased load. Conversely, if the average memory utilization is below the threshold, the system will remove replicas to save resources.

For REST microservices, memory utilization is a critical metric, as these services tend to be memory-intensive. The autoscaling of memory utilization for REST microservices can be managed by setting the desired memory utilization percentage and the minimum and maximum number of replicas in the deployment. Memory-based autoscaling is particularly suitable for memory-intensive applications, as it provides a more accurate representation of resource utilization compared to CPU-based scaling. By directly accounting for memory usage, it helps optimize resource allocation for applications that are heavily I/O bound. However, memory-based autoscaling can be more complex to set up, requiring a deeper understanding of an application's memory needs. Additionally, memory utilization is harder to monitor and measure than CPU usage, making it more challenging to ensure that autoscaling is functioning effectively.

B. Sinan

Sinan is a Machine Learning-based and QoS-Aware Resource Management system for Cloud Microservices. It predicts end-to-end latency and the probability of QoS violations using machine learning models, based on the system's state and historical data. The operation of Sinan involves the following steps:

- *Query Request*: The process begins when a query request is sent from the Server Cluster to the Centralized Scheduler.
- *Docker Info*: The Server Cluster sends Docker-related information to the Centralized Scheduler, providing details about the containerized environment.
- *Model Inputs*: The Centralized Scheduler processes this information and sends the relevant model inputs to the Prediction Server, which utilizes CNN and XGBoost technologies.
- *Predictions*: The Prediction Server returns predictions back to the Centralized Scheduler. These predictions include the end-to-end latency and probability of QoS violations.
- *Resource Allocations*: Based on the predictions received, the Centralized Scheduler makes decisions about resource allocation and sends these instructions back to the Server Cluster.

The system architecture is illustrated in Figure 1.

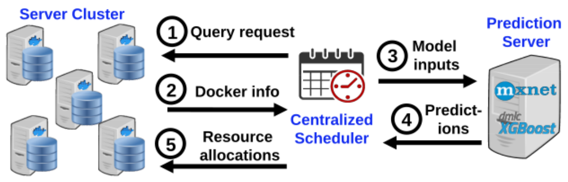


Fig. 1: System Architecture of Sinan.

Data Collection: Sinan consists of a Convolutional Neural Network (CNN) model for predicting short-term tail latency and a Boosted Trees (BT) model for predicting the probability of Quality of Service (QoS) violations in the long term. The CNN handles immediate predictions, and the BT model addresses future QoS violations, enabling Sinan to adjust resources accordingly to meet QoS requirements.

Sinan employs a data collection agent to gather training data efficiently. The collected data includes historical resource utilization and performance metrics, which are used to train the machine learning models. This data collection process is guided by an algorithm designed to explore the resource allocation space effectively, focusing particularly on boundary regions where QoS violations may arise. Here

- X_{RH} represent the resource usage history (a 3D tensor)
- X_{LH} represent the latency history (a 2D matrix)
- X_{RC} represent the resource configuration for the next timestep (a 2D matrix)

The data collection process continuously monitors resource utilization (e.g., CPU, memory, network usage) and latency over time. This historical data is crucial for training the CNN and BT models.

Convolutional Neural Network (CNN) Model: The CNN model is designed to predict the *end-to-end tail latency* (y_L) for the next timestep, given the resource utilization history, latency history, and the resource allocation for the next timestep. The input to the CNN consists of the following components:

The CNN processes this input through several convolutional layers, which learn dependencies between the microservice tiers over the time window. The output of the convolutional layers is a latent representation L_f, which is then passed through fully-connected layers to predict the tail latency y_L for the next timestep.

The output is the predicted end-to-end tail latency for the next timestep:

$$y_L = \text{CNN}(X_{RH}, X_{LH}, X_{RC})$$

Boosted Trees (BT) Model: The BT model is used to predict the probability of a *QoS violation* (p_V) further into the future. This is a binary classification problem, where the BT model predicts the likelihood of a QoS violation occurring based on the latent representation L_f generated by the CNN and the resource allocation for the next k timesteps.

Let:

L_f represent the latent variable extracted by the CNN,

The BT model outputs the probability of a QoS violation p_V at timestep k:

$$p_V = \text{BT}(L_f, X_{RC})$$

Prediction and Resource Adjustment: At runtime, Sinan uses the predictions from both models to dynamically adjust the resources allocated to each microservice to maintain QoS:

- The CNN model predicts the *immediate latency* y_L for the next timestep, which is used to adjust resource allocation to minimize latency.
- The BT model predicts the *probability of a QoS violation* p_V for a given resource configuration over the next k timesteps.

Figure 2 illustrates the hybrid model of Sinan.

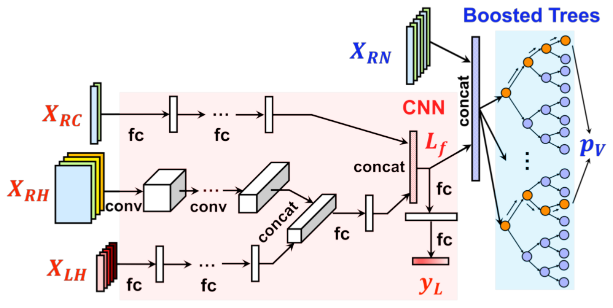


Fig. 2: Sinan's hybrid model

The two-stage approach in Sinan, combining CNN for short-term latency prediction and BT for long-term QoS violation prediction, allows for efficient and accurate resource management in cloud microservices. The system adjusts resources dynamically based on real-time predictions, and retrains its models to adapt to changing application scenarios, ensuring the optimization of both performance and resource efficiency.

To further enhance its performance, Sinan incorporates several advanced methodologies:

Resource Allocation Space Exploration: Sinan utilizes a Multi-Armed Bandit (MAB) process for resource allocation exploration, with a balance factor of $\alpha = 20\%$. This allows efficient exploration of configuration spaces to identify optimal allocations.

Online Scheduler: The scheduler dynamically adjusts resources with actions such as:

- *Scale down -1:* Reduce resources by one unit.
- *Scale down to least of k rounds:* Gradually reduce resources to the minimum observed configuration over k rounds.
- *Hold:* Maintain the current allocation.
- *Scale-up +1:* Incrementally increase resources by one unit.
- *Scale-up all:* Significantly increase resources across all tiers.
- *Scale-up victims:* Allocate additional resources to tiers with the most significant QoS violations.

Transfer Learning: Sinan employs transfer learning techniques to leverage knowledge from a local cluster to optimize resource allocation in the cloud environment, enhancing generalization and adaptability.

Results: Sinan demonstrates a significant improvement in performance compared to existing methods. Specifically, it achieves a $25.9\% - 59.0\%$ improvement in resource efficiency and QoS maintenance over PowerChief [22] and AWS AutoScaleOpt.

C. *AutoThrottle*

Autothrottle is a bi-level learning-assisted framework designed for resource management in microservice applications with Service Level Objective (SLO) guarantees. The framework consists of two components: a global controller called Tower and per-service controllers named Captains. The Tower utilizes contextual bandits, a lightweight form of online reinforcement learning, to determine suitable performance targets based on observed workloads, CPU allocations, and end-to-end latencies. These targets are communicated to the Captains as CPU throttle ratios. Autothrottle implements the performance target with CPU throttle ratio - the fraction of time a microservice is stopped by the underlying CPU scheduler. A low CPU throttle ratio indicates underutilization, whereas a high throttle ratio suggests overutilization. The design is motivated by a strong correlation between CPU throttle and service latencies revealed by their correlation test.

Autothrottle's bi-level design decouples application-level SLO feedback from low-level resource control by delegating fine-grained resource adjustments to Captains. This architecture reduces the complexity of global decision-making and avoids the overhead of aggregating resource metrics while enabling rapid responses to dynamic workload fluctuations. Figure 3 illustrates the top level system architecture.

Per-service controllers - Captains: The Captain is the per-service controller responsible for dynamically scaling CPU resources for microservices to meet the target CPU throttle ratio assigned by the Tower. By continuously monitoring CPU metrics and adjusting quotas in response to workload

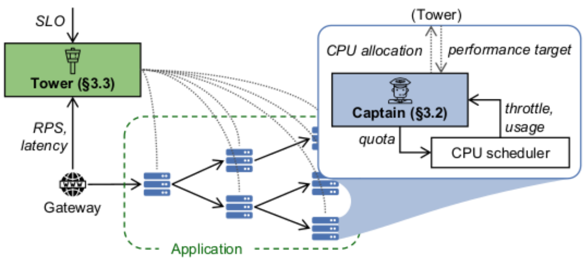


Fig. 3: Autothrottle System Diagram

changes, the Captain ensures efficient resource utilization while preventing Service Level Objective (SLO) violations.

Each Captain operates in periodic intervals, typically N Continuous Fair Scheduler (CFS) periods, where it gathers resource metrics and makes decisions on whether to scale up or down the CPU allocation. The control logic is based on CPU throttle ratio measurements, CPU usage statistics, and a dynamically adjusted margin parameter to prevent resource overreaction.

The decision-making process of Captain comprises the following components:

- **Multiplicative Scale-Up:** When the measured CPU throttle ratio exceeds a defined threshold ($\alpha\times$ the target ratio), Captain increases the CPU quota proportionally to the difference between the measured and target ratios. This approach ensures that under-provisioned microservices rapidly receive additional resources to handle demand surges, thereby avoiding potential SLO violations.
- **Instantaneous Scale-Down:** When the CPU throttle ratio is below the target, Captain leverages historical CPU usage over the most recent M periods to propose a new CPU quota. The proposed quota is calculated as the maximum CPU usage plus a dynamically tuned margin times the standard deviation of usage. This approach helps reclaim excess CPU allocation while avoiding unnecessary fluctuations.
- **Rollback Mechanism After Scaling Down:** Since accidental CPU scale-downs can result in SLO violations, Captain implements a fast rollback mechanism. Following each scale-down, Captain monitors the throttle ratio during every period within the next N periods. If the ratio exceeds the $\alpha\times$ target threshold, Captain immediately restores the previous higher quota and adds an additional allocation to compensate for potential delays caused by the erroneous scale-down.

Algorithm 1 and Algorithm 2 outline the Captain's scaling and rollback operations in detail. These algorithms ensure that the system can efficiently adapt to changing workloads while maintaining reliable and responsive service behavior.

Application-level controller - Tower: In Autothrottle, the Tower serves as the global controller responsible for computing and dispatching target CPU throttle ratios for each microservice. These target ratios act as guidelines for the

Algorithm 1 Captain: scaling up and down

```
    ▷ executes every $N$ periods
1:  throttleCount ← throttle count during last $N$ periods
2:  throttleRatio ← throttleCount/$N$
3:  margin ← max(0, margin + throttleRatio − throttleTarget)
4:  if throttleRatio $> \alpha\times$ throttleTarget then
        ▷ multiplicatively scale up
5:      quota ← quota $\times(1+$ throttle.Ratio $-\alpha\times$ throttleTarget$)$
6:  else
        ▷ instantaneously scale down
7:      history ← CPU usage history in the last $M$ periods
8:      proposed ← max(history) + margin × stdev(history)
9:      if proposed $\leq \beta_{\max}\times$ quota then
10:         quota ← max($\beta_{\min}\times$ quota, proposed)
11:     end if
12: end if
```

Algorithm 2 Captain: rollback mechanism

```
    ▷ executes every period for $N$ periods after each scale-down
1:  lastQuota ← CPU quota before scale-down
2:  throttleCount ← throttle count since scale-down
3:  throttleRatio ← throttleCount/$N$
4:  if throttleRatio $> \alpha\times$ throttleTarget then
        ▷ revert to the previous (higher) quota before scale-down
        ▷ with an additional allocation equal to the quota difference
5:      quota ← lastQuota + (lastQuota − quota)
6:      margin ← margin + throttleRatio − throttleTarget
7:  end if
```

per-service Captains, enabling them to make in-situ resource adjustments. By delegating resource control to Captains, the Tower avoids the latency overhead typically associated with distributed tracing and logging while maintaining a global perspective on application-level Service Level Objective (SLO) compliance and resource usage.

The Tower operates at a lower frequency compared to Captains, typically updating target throttle ratios once every minute. This longer interval allows tail request latencies and average CPU usage to stabilize after resource adjustments, thus simplifying decision-making into a "one-step" problem. Unlike full-fledged reinforcement learning (RL) approaches that must account for long-term consequences, the Tower only needs to compute the optimal CPU throttle targets for the current interval, without considering historical decisions.

Given this "one-step" nature, the Tower employs contextual bandits, a lightweight and efficient class of online reinforcement learning algorithms that excel at decision-making in scenarios where each action's impact is confined to the immediate outcome.

1) Primer on Contextual Bandits: Contextual bandits are well-suited for real-time decision-making in resource management scenarios. Similar to multi-armed bandits, they focus on selecting the best action at each step to minimize cumulative cost. However, they differ by considering the system's current state, known as the context, to inform decisions.

Contextual bandits are more lightweight than full RL algorithms, as they do not require extensive offline training or frequent retraining. Instead, they learn incrementally from observed data, making them ideal for online learning in dynamic environments.

A common approach for solving contextual bandit problems is to train a cost-prediction model that estimates the cost of taking each action within a given context. Due to the partial observability of contextual bandits (only the cost of the selected action is observed), counterfactual estimation techniques are often employed to estimate the costs of unselected actions, improving sample efficiency and decision accuracy.

2) Realizing Contextual Bandits in Tower: The Tower's contextual bandit algorithm operates with a step size of one minute, aiming to select the action that incurs the lowest cost given the observed context.

a) Context: The Tower selects the average Requests Per Second (RPS) observed during the last interval as the context. This choice is motivated by the strong correlation between RPS and the optimal CPU throttle target. Other metrics, such as CPU usage, are excluded from the context as they are merely byproducts of applying a throttle target to an RPS.

b) Action Space: The Tower searches for a ladder of CPU throttle targets as potential actions. By default, the action space consists of nine throttle targets, ranging from 0 to 0.3.

c) Reduction of Action Space: Microservice-based applications can contain thousands of services, leading to a combinatorially large action space if each service were assigned a unique throttle target. To address this, the Tower clusters microservices into two classes based on their average CPU usage and assigns an action to each class, reducing the action space to 81 possible combinations. The k-means clustering algorithm is employed for this purpose.

d) Cost Function: The cost function used by the Tower is defined as follows:

When the SLO is met, the cost is based solely on the total CPU allocation, normalized linearly to the range [0, 1].

When the SLO is violated, the cost is based solely on the tail latency, normalized to the range [2, 3] to emphasize the higher priority of SLO violations.

The choice of these normalization ranges is empirically validated, although they may not represent the optimal configuration.

e) Noise Reduction for Costs: To address the high noise in cost measurements, the Tower buffers recent samples and groups them based on the context and action. When a new sample is observed, the median cost of its group is used for model updates, significantly reducing noise and stabilizing the learning process.

f) Exploration Strategy: To balance exploration and exploitation, the Tower employs a neighbor-based exploration strategy. Given a sorted ladder of CPU throttle targets $r_1 < r_2 < ... < r_9$, if the best action is (r_i, r_j), the neighbors (r_i, r_{j-1}), (r_i, r_{j+1}), (r_{i-1}, r_j), and (r_{i+1}, r_j) are explored with equal probability, subject to boundary conditions. This approach ensures efficient exploration without compromising the learning process.

By adopting contextual bandits and the design choices outlined above, the Tower achieves lightweight, real-time resource management for microservice applications, minimizing CPU allocations while ensuring SLO compliance.

Results: Autothrottle achieves significant CPU resource savings while consistently maintaining application SLOs across different workloads. In Social-Network, it saves up to 49.85% (49.7 cores) over K8s-CPU-Fast and 25.93% (17.5 cores) over K8s-CPU, achieving a P99 latency of 178 ms with only 77.5 cores. In comparison, K8s-CPU requires 115.5 cores for 177 ms latency, and K8s-CPU-Fast requires 93.9 cores for 171 ms. For Hotel-Reservation, where requests traverse only three microservices, the resource savings are less pronounced due to the application's simplicity. In a 21-day real-world evaluation using Social-Network on a 160-core cluster, Autothrottle achieves an average CPU saving of 12.1 cores per hour compared to K8s-CPU, with a peak saving of 35.2 cores. Additionally, it drastically reduces SLO violations, recording only 5 instances compared to K8s-CPU's 71 violations. On a large-scale 512-core cluster, Autothrottle demonstrates up to 28.24% (150 cores) savings over K8s-CPU and at least 5.92% (24 cores) over K8s-CPU-Fast while keeping P99 latency within the 200 ms SLO limit. These numbers underscore Autothrottle's efficiency in CPU allocation and application performance stability.

III. AutoLimit Methodology and Implementation

In this section, we provide our methodology to incorporate memory feature in Autothrottle and implementation details.

A. AutoLimit

The motivation for our work involves a limitation of Autothrottle. In their implementation the pod is created without any memory limits specified. The pod may use the node's full memory capacity without restrictions. This phenomemom can be observed by checking the 'memory_limit_in_bytes' value in the 'sys/fs/cgroup/memory' folder. In their implementation, the memory limit is set to 9223372036854771712 which is the page size used in Linux. It means "no memory limit" is set. We applied a heuristics based rule to auto scale-up and scale-down this memory.

Our methodology introduces dynamic memory management to Autothrottle by implementing a watermark-based scaling system. The CaptainScaler class monitors pod memory usage against configurable watermarks (90% high, 70% low) and adjusts limits accordingly. When memory usage exceeds the high watermark, the system increases the memory limit by factoring in a 30% headroom buffer (memory_headroom = 1.3) to prevent resource contention. Conversely, when usage falls below the low watermark, the limit is reduced to optimize resource utilization. The system enforces a minimum memory threshold of 64MB and requires a difference of at least 1MB before applying limit changes, preventing rapid oscillations. This approach ensures efficient memory utilization while protecting against unbounded consumption that could occur in the original Autothrottle implementation.

B. Implementation

We evaluated our implementation in two different cloud environments: Microsoft Azure and Cloud Lab BUET. In the Azure setup, we used a 6-core cluster (3x 2-core VMs). The three VMs each featured a 2-core Azure B2als_v2 (AMD) processor and 4 GB RAM. For resource provisioning, we used Terraform. In our Cloud Lab BUET setup, we utilized three virtual machines running Ubuntu 22.04.5 LTS (x86_64) on OpenStack Nova 27.4.0. Each VM was equipped with an Intel Xeon (Icelake) processor with 32 cores at 2.0 GHz and approximately 126 GB of memory (128805 MiB). We configured VM1 as the control plane node and designated VM2 and VM3 as worker nodes. The VMs were equipped with Virtio GPU for basic display capabilities. Much effort went into updating the codebase of Autothrottle from Kubernetes version v1.20 to v1.32. The Captain and Tower implementation, workload generation, and evaluation were performed using Python. We also have a Bash script to streamline the entire setup and evaluation process. The complete implementation can be found at https://github.com/hmasum52/autoalloc.

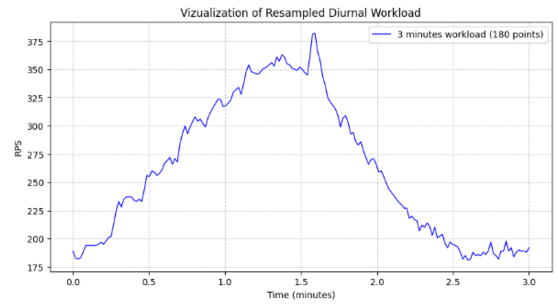
IV. Evaluation

In this section, we explain the methodology used for evaluation and present the results obtained from experiments.

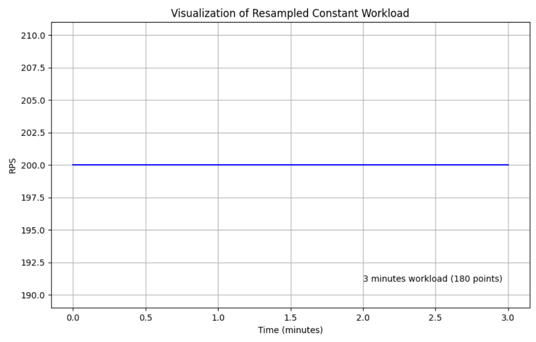
A. Methodology

We deploy an SLO-targeted microservice application, Hotel-Reservation, from DeathStarBench [23]. This application is representative of real-world microservices, with stateless services (e.g., business logic), data services (e.g., key-value stores), and gateways. In Appendix A, we list the services. Deployments are managed by Docker and Kubernetes. Parent-child service communications are through popular RPC frameworks such as gRPC and Thrift.

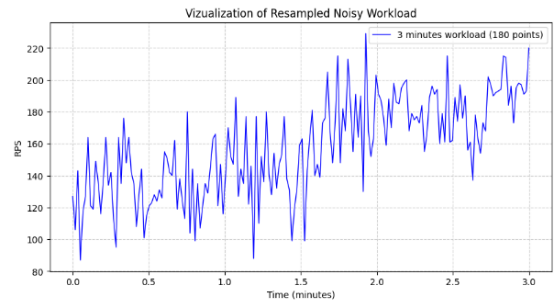
We generate workloads with Locust [24], which is configured to mix application requests to stress as many services as possible. Locust replays workload traces to reproduce RPS (requests per second). The first set of traces captures hourly RPS patterns, commonly observed in production environments: Puffer's streaming requests [25], Google's cluster usage [26], and Twitter tweets [27]. We used four types of workloads: diurnal, constant, noisy, and bursty. A diurnal workload follows a daily cycle, peaking during specific times of the day, such as web traffic that rises during the daytime and falls at night. A constant workload remains steady, with no fluctuations over time, like consistent data processing rates. A noisy workload is marked by random, unpredictable variations in demand, often due to external disruptions. Finally, a bursty workload consists of brief periods of intense activity, followed by quieter intervals, commonly seen in events like flash sales. To reduce Microsoft Azure costs and experiment duration, we set the duration to 3 minutes and reduced the RPS by a factor of 100. Figure 4 illustrates these patterns.



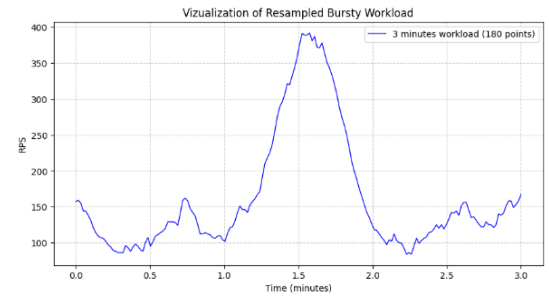
(a) Diurnal Workload



(b) Constant Workload



(c) Noisy Workload



(d) Bursty Workload

Fig. 4: Workload Types: Diurnal, Constant, Noisy, and Bursty

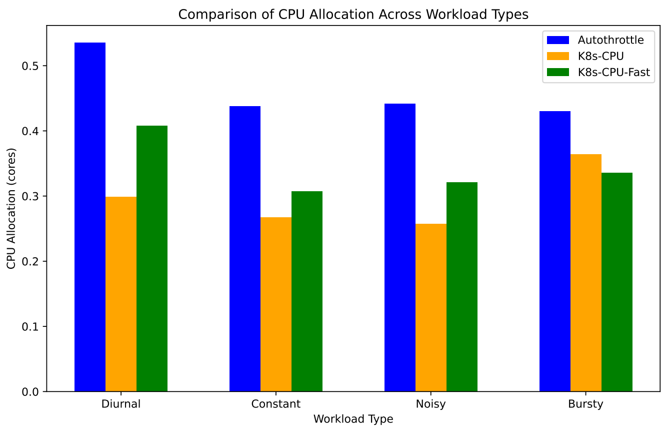


Fig. 5: Experiment 1 Result

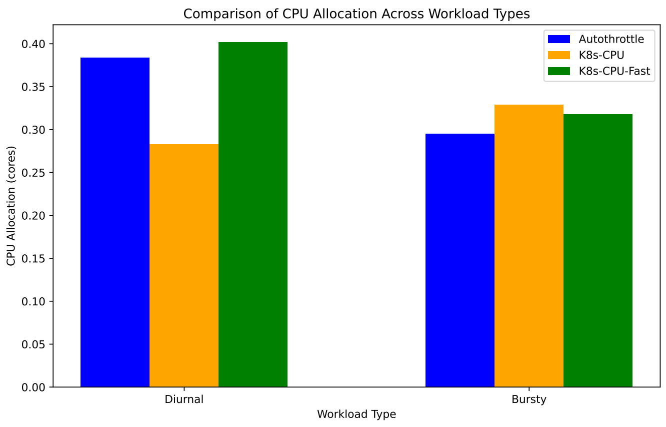


Fig. 6: Experiment 2 Result

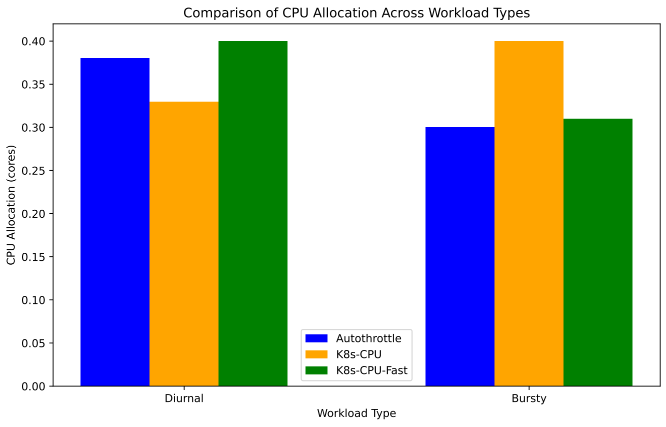


Fig. 7: Experiment 4 Result

B. Results

1) Reproduce original paper: We attempted to reproduce the results from the original Autothrottle paper through three experiments with varying training configurations. The goal was to demonstrate Autothrottle's ability to achieve efficient resource utilization compared to Kubernetes' Vertical Pod Autoscaler (K8s-CPU and K8s-CPU-Fast). In Experiment 1 (Figure 5), we set an SLO of 100ms P99 latency with minimal training: 2 warm-up periods (1 random exploration, 1 normal learning) and a duration of 1 hour 55 minutes. The results show Autothrottle using higher CPU allocations (approximately 0.42 cores) compared to K8s approaches. This suboptimal performance can be attributed to insufficient training time (online learning), which prevents Autothrottle from learning optimal resource allocation patterns. Experiment 2 (Figure 6) incorporated improved training parameters: a more relaxed SLO of 2s P99 latency, extended warm-up period of 6 cycles (3 random, 3 learning), and a duration of 1 hour 57 minutes. With this configuration, we observed Autothrottle's intended behavior in the bursty workload pattern, where it achieved lower CPU allocation (0.3 cores) compared to both K8s-CPU (0.33 cores) and K8s-CPU-Fast (0.32 cores). This demonstrates that with proper training, Autothrottle can learn to allocate resources more efficiently. In Experiment 4 (Figure 7), we further extended the training parameters with 8 warm-up cycles (4 random, 4 learning) and a longer duration of 3 hours 10 minutes, maintaining the 2s P99 latency SLO. For diurnal workloads, while Autothrottle showed higher allocations, it demonstrated improved efficiency in bursty workloads with an allocation of 0.35 cores compared to K8s-CPU's 0.4 cores. These experiments highlight the crucial role of proper training in Autothrottle's performance. While our initial experiment with limited training showed suboptimal results, extended training periods in subsequent experiments enabled Autothrottle to achieve its designed goal of more efficient resource utilization, particularly in bursty workload scenarios. This underscores the importance of adequate training time and appropriate SLO settings for Autothrottle to effectively learn and optimize resource allocation patterns.

2) Memory Management Results: We evaluated our proposed AutoLimit memory management system against the original Autothrottle implementation. While Autothrottle doesn't set any memory limit, we used a fixed predefined memory limit of 256 MB to compare with our approach of dynamic memory scaling based on actual usage patterns. In our implementation, we added detailed logging of memory-related metrics. The logs capture CPU limits, memory usage, memory limits, and memory usage ratios at each adjustment interval. To analyze these logs, we used regex-based parsing as shown below in Appendix B. The results are detailed in Table I.

Our analysis of memory allocation logs from two worker nodes across multiple experimental runs highlights the effectiveness of our heuristics-based scaling system in optimizing resource utilization.

TABLE I: Comparison of Memory Limits and Savings (Initial memory limit 256 MB)

VM	Average Memory Limit	Memory Limit Saving
Worker-2 (Exp 1)	101.77 MB	60.2%
Worker-3 (Exp 1)	126.49 MB	50.5%
Worker-2 (Exp 2)	79.91 MB	72.3%
Worker-3 (Exp 2)	125.10 MB	51.1%

As Autothrolle's implementation was done for cgroup version one, in the initial experiments, use used two worker node (with ubuntu 20.04 LTS) both having cgroup version one which provides the metrics *memory.usage_in_bytes* and *memory.limit_in_bytes*. We utilized this two available metrics to update the limit dynamically based on memory usage. In our experiment, Worker Node 2 maintained an average memory limit of 101.77MB, while Worker Node 3 averaged 126.49MB. These allocations resulted in significant memory savings of 60.2% and 50.5%, respectively, compared to Autothrottle's fixed 256MB allocation.

As latest version of ubuntu comes with cgroup vesion 2 which have the metrics *memory.current* (equivalent to *memory.usage_in_bytes*) and *memory.max* (equivalent to *memory.limit_in_bytes*) we migrate our code to use these two metrics for our dynamic memory limit changing. For our second experiment, we used worker node having cgroup version two (ubuntu 22.04 LTS). In the second set of experiments, Worker Node 2 exhibited an even lower average memory limit of 79.91MB, while Worker Node 3 remained consistent at 125.10MB. The corresponding memory savings increased to 72.3% and 51.1%, reinforcing the efficiency of our approach. The variation in memory limits between worker nodes (~80MB to 126MB) illustrates our system's adaptability to fluctuating workload patterns and resource demands. This dynamic scaling capability represents a substantial advancement over Autothrottle's rigid 256MB allocation, enabling memory savings of up to 72.3%. Crucially, these optimizations are achieved without degrading application performance, ensuring that each node receives precisely the resources it requires.

V. Conclusion

In this paper, we presented AutoLimit, an extension of Autothrottle that introduces dynamic memory management for microservices while maintaining SLO compliance. Our implementation demonstrated significant memory savings, reducing allocation by 50.5% to 72.3% compared to Autothrottle's static allocation approach. Through a heuristics-based scaling system with configurable high and low thresholds, AutoLimit effectively balances resource efficiency with performance requirements. The system's ability to maintain consistent performance while significantly reducing memory usage validates our approach to dynamic resource management. Several promising directions exist for future work. These include extending evaluations with longer-duration and memory-intensive workloads to better understand the system's behavior under sustained pressure. Additionally, developing a bi-level approach for memory management—similar to Autothrottle's CPU management strategy—could enhance efficiency, potentially incorporating online learning techniques for more sophisticated threshold adjustments.

References

[1] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 149–161.

[2] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[3] A. Gluck, "Introducing domain-oriented microservice architecture," *Uber Engineering Blog*, p. 45, 2020.

[4] G. Santoli, "Microservices architectures: Become a unicorn like netflix, twitter and hailo," *Presentation Slides, Mar*, vol. 31, 2016.

[5] J. Cloud, "Decomposing twitter: Adventures in service-oriented architecture," *QCon New York*, 2013.

[6] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 301–312, 2014.

[7] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni *et al.*, "Morpheus: Towards automated {SLOs} for enterprise clusters," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 117–134.

[8] P. Ambati, Í. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety *et al.*, "Providing {SLOs} for {Resource-Harvesting}{VMs} in cloud platforms," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 735–751.

[9] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang *et al.*, "{PerfIso}: Performance isolation for commercial {Latency-Sensitive} services," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 519–532.

[10] S. A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi, "Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments," in *Proceedings of the ACM symposium on cloud computing*, 2019, pp. 272–285.

[11] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 450–462.

[12] Y. Wang, K. Arya, M. Kogias, M. Vanga, A. Bhandari, N. J. Yadwadkar, S. Sen, S. Elnikety, C. Kozyrakis, and R. Bianchini, "Smartharvest: Harvesting idle cpus safely and efficiently in the cloud," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 1–16.

[13] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 724–739.

[14] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 167–181. [Online]. Available: https://doi.org/10.1145/3445814.3446693

[15] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2022.

[16] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–14. [Online]. Available: https://doi.org/10.1145/3567955.3567960

[17] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022, pp. 1868–1877.

[18] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: Efficient resource management for shared microservices with sla guarantees," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2022, p. 62–77. [Online]. Available: https://doi.org/10.1145/3567955.3567964

[19] J. Park, B. Choi, C. Lee, and D. Han, "Graf: a graph neural network based proactive resource allocation framework for slo-oriented microservices," in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 154–167. [Online]. Available: https://doi.org/10.1145/3485983.3494866

[20] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical efficient microservice autoscaling with qos assurance," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 240–252. [Online]. Available: https://doi.org/10.1145/3502181.3531460

[21] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: an intelligent fine-grained resource management framework for slo-oriented microservices," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[22] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars, "Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 133–146.

[23] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[24] Locust, "Locust: An open source load testing tool," https://locust.io, 2025, accessed: 2025-02-04.

[25] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein, "Learning in situ: a randomized experiment in video streaming," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 495–511.

[26] J. Wilkes, "Google cluster data – 2019 traces," https://github.com/google/cluster-data/blob/master/ClusterData2019.md, 2020, accessed: 2025-02-04.

[27] Twitter, "Twitter data for academic research," https://developer.twitter.com/en/use-cases/do-research/academic-research/resources, 2022, accessed: 2022-01-01.

APPENDIX A
HOTEL RESERVATION BENCHMARK

Services: Total Services: 17

Workload details: Search: 60%, Recommend: 39%, Reserver: 0.5% , Login: 0.5%

High CPU Usage Group:
- Frontend
- Geo
- Profile
- Rate
- Reservation
- Search

Low CPU Usage Group:
- Memcached (3)
- Mongodb (6)
- Consul
- Jaeger

Distribution of Services in Different VMs

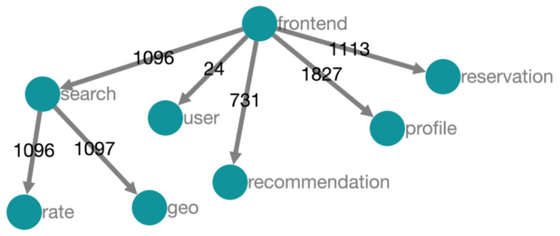


Fig. 8: Services in hotel reservation microservice

- Master Node (autothrottle-1):
  - Kubernetes control plane only
- Worker Node 1 (autothrottle-2):
  - Consul
  - Jaeger
  - Frontend
  - Profile service stack (service + MongoDB + memcached)
  - Rate service stack (service + MongoDB + memcached)
  - Reservation service stack (service + MongoDB + memcached)
- Worker Node 2 (autothrottle-3):
  - Geo service stack (service + MongoDB)
  - Recommendation service stack (service + MongoDB)
  - Search service
  - User service stack (service + MongoDB)

APPENDIX B
MEMORY IMPLEMENTATION RESULT CALCULATION

Experiment Set 1

```
Worker node 2:

$ ggrep -oP "^\d{2}:\d{2}:\d{2}
captain_scaler:
new memory limit \K[\d.]+"
worker-daemon-worker-2.log |
awk '{ sum += $1 }
END { print "Average = ", sum/NR }'
> Average = 101.768
```

```
 Worker node 3:

$ ggrep -oP "^\d{2}:\d{2}:\d{2}
captain_scaler:
new memory limit \K[\d.]+"
worker-daemon-worker-3.log |
awk '{ sum += $1 }
END { print "Average = ", sum/NR }'
> Average = 126.494
```

Experiment Set 2

```
Worker node 2:

$ ggrep -oP "captain_scaler:
```

```
new memory limit \K[\d.]+"
worker-daemon-worker-2.log |
awk '{ sum += $1 }
END { print "Average = ", sum/NR }'
> Average = 79.9068
```

Worker node 3:

```
$ ggrep -oP "captain_scaler:
new memory limit \K[\d.]+"
worker-daemon-worker-3.log |
awk '{ sum += $1 }
END { print "Average = ", sum/NR }'
> Average = 125.102
```