TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices

Jinwoo Park
KAIST
Daejeon, Republic of Korea
jinwoo520528@kaist.ac.kr

Jaehyeong Park
KAIST
Daejeon, Republic of Korea
woguddlrj676@kaist.ac.kr

Youngmok Jung
KAIST
Daejeon, Republic of Korea
tom418@kaist.ac.kr

Hwijoon Lim
KAIST
Daejeon, Republic of Korea
hwijoon.lim@kaist.ac.kr

Hyunho Yeo
Moloco
Redwood City, California, USA
hyunho.yeo@moloco.com

Dongsu Han
KAIST
Daejeon, Republic of Korea
dhan.ee@kaist.ac.kr

ABSTRACT

Microservice has become a de facto standard for building large-scale cloud applications. Overload control is essential in preventing microservice failures and maintaining system performance under overloads. Although several approaches have been proposed, they are limited to mitigating the overload of individual microservices, lacking assessments of interdependent microservices and APIs.

This paper presents TopFull, an adaptive overload control at entry for microservices that leverages global observations to maximize throughput that meets service level objectives (i.e., goodput). TopFull makes adaptive load control on a per-API basis, exercises parallel control on each independent subset of microservices, and applies RL-based rate controllers that adjust the admitted rates of the APIs at entry according to the severity of overload. Our experiments on various open-source benchmarks demonstrate that TopFull significantly increases goodput in overload scenarios, outperforming DAGOR by 1.82x and Breakwater by 2.26x. Furthermore, the Kubernetes autoscaler with TopFull serves up to 3.91x more requests under traffic surge and tolerates traffic spikes with up to 57% fewer resources than the standalone Kubernetes autoscaler.

CCS CONCEPTS

• **Networks, Computer systems organization → Cloud computing**; • **Software and its engineering → n-tier architectures**; • **Computing methodologies → Machine learning approaches**.

KEYWORDS

Microservices, Overload Control, Quality of Service, Resources Optimization, Applied Machine Learning, Cloud Computing

ACM Reference Format:
Jinwoo Park, Jaehyeong Park, Youngmok Jung, Hwijoon Lim, Hyunho Yeo, and Dongsu Han. 2024. TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3651890.3672253

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672253

1 INTRODUCTION

Microservice has become a popular architectural choice for cloud-based applications, widely adopted across various enterprises such as Amazon [2], Netflix [9], and Airbnb [4]. However, microservice applications are vulnerable to unexpected load caused by various factors such as sudden traffic increase, success disaster [1], retry storm by misbehaving clients, under-scaled microservices, instance failures, or cloud provider issues [15]. They often result in violations of Service Level Objectives (SLOs) and, in extreme cases, cause service outages [3, 5–8]. For example, Amazon experienced a service outage due to a massive traffic surge on Prime Day [5], while Zoom and Microsoft faced service outages due to a rise in demand [6, 7]. Both large-scale and micro traffic changes are not uncommon [43], and services must have built-in mechanisms to handle them gracefully.

Implementing overload control in microservices is vital to ensure consistent performance across diverse overload scenarios and to compensate for the limits of autoscalers. Although microservices use autoscalers to handle demand changes, they alone are not silver bullets for managing overloads in microservices. Despite extensive research on autoscalers [35, 38–41, 46–48, 51], there is a fundamental limitation—autoscalers take several seconds to minutes to provision additional resources in response to demand changes. This may result in a transient, but severe degradation of *goodput*, the rate of successful responses that satisfy latency SLOs.

An ideal overload controller must simultaneously satisfy two criteria: First, it should rapidly mitigate catastrophic failures triggered by traffic surges, thereby minimizing the number of requests that experience SLO violations, while also being carefully calibrated to avoid excessive throttling. Second, in scenarios where multiple APIs are served, the controller must ensure cost-effectiveness by maximizing goodput, while also maintaining business priorities through the careful implementation of selective API throttling.

However, current overload controllers [26, 44, 53] are primarily effective in fulfilling the first of the two requirements. This limitation arises from their inability to perform a holistic scheduling that takes into account the impact of all APIs on every microservice within an application. Figure 1 elucidates this shortfall using an illustrative example. It depicts a scenario where two APIs (API_1 and API_2) receive traffic at a rate of 10 thousand requests per second (rps), while microservices M_A and M_B have a serving capacity of $10k$ and $3k$ rps, respectively. In an ideal situation, these microservices

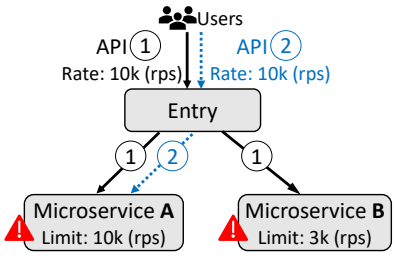


Figure 1: Overload scenario where API 2 is vulnerable to performance loss from starvation.

could distribute their capacity to serve API_1 and API_2 at rates of $3k$ and $7k$ rps, respectively, achieving a total goodput of $10k$ rps. Unfortunately, traditional approaches, which implement traffic shading at the microservice level, only manage to attain a total goodput of 80% of the potential capacity. Given that at M_A, both API_1 and API_2 are throttled equally to $5k$ rps, and considering the additional throttling of API_1 to $3k$ rps at M_B, the effective throughput for API_1 is reduced to $3k$ rps. This results in significant inefficiency, as responses for $2k$ rps of API_1 traffic are initially processed at M_A but ultimately discarded since the responses are incomplete and partially constructed.

To this end, we introduce TopFull, an overload control system that is capable of meeting both critical requirements: rapid response to traffic surges and overall cost efficiency. Nevertheless, the development of such an advanced overload control framework is nontrivial, due to two crucial challenges:

- Determining the optimal throttling strategy for interdependent APIs and microservices is significantly complex. The load management on an API influences the efficacy of load control for other competing APIs, creating a circular dependency. Moreover, establishing the acceptable admission rates for an API requires taking into account all the overloaded microservices along the API's execution path.
- It is difficult to control multiple APIs concurrently and discern the specific impact of each control. Each API may traverse dozens to hundreds of microservices, with some of these APIs sharing overloaded microservices along their execution paths. The aggregate goodput resulting from the adjustment of multiple APIs is complicated by the distinct execution paths of each API and the resource characteristics of each overloaded microservice.

TopFull addresses the challenges by introducing novel system designs. First, TopFull searches for the optimal balance across APIs by selectively managing APIs through API-wise load control. TopFull resolves overloads by systematically choosing a target overloaded microservice and tightening the corresponding APIs to avoid creating a circular dependency. Meanwhile, TopFull confirms every microservice along the API's execution path is not suffering overload when rate-increasing the API. In this way, we safely search for maximal utilization of a bottleneck microservice while avoiding incomplete and partially constructed responses. Second, TopFull enables parallel independent load controls by splitting the microservices into smaller subsets. We cluster APIs and their associated overloaded microservices based on whether they are shared among these APIs. Each cluster thus forms a smaller but independent group, mutually unaffected by others' load control at the moment. This allows us to conduct load control in parallel at each cluster safely. Finally, TopFull adopts a Reinforcement Learning (RL)-based rate controller that can adjust the aggressiveness of API rate throttling by diagnosing overload severity with end-to-end performance metrics of goodput and percentile latencies. The RL-based rate controller shows rapid responsiveness towards overloads and resource allocations from an autoscaler.

We evaluate TopFull with a full system implementation. Our evaluation on open-source microservices benchmark applications, Train Ticket [12] and Online Boutique [21], shows that TopFull outperforms recent overload controls, serving 1.82x higher average goodput than DAGOR and 2.26x greater than Breakwater during overload scenarios. TopFull also enhances the robustness of microservices with autoscaler compared to the autoscaler standalone, improving the goodput by 1.38-3.91x under overload and enduring traffic surge with 50-57% less resources. Moreover, TopFull is capable of handling various overload scenarios, including adaptation to internal instance failures.

In summary, we make the following key contributions:

- **Execution path-aware top-down load control** : TopFull is the first API-wise overload controller that utilizes the complete execution paths of multiple APIs and the characteristics of microservices along those paths to maximize goodput.
- **Scalable overload control framework:** The clustering of external APIs captures granular clusters within complex microservices, enabling parallel load control.
- **Flexible overload mitigation policy:** We adopt an RL-based rate controller, which provides an adaptive rate control policy without human-expert engineering.

This work does not raise any ethical issues.

2 MOTIVATION

Concurrent load control causes starvation. Existing overload control approaches for microservices [26, 44, 53] perform load control on individual microservices that are experiencing overloads. They effectively mitigate overload at microservices, but their load control decisions themselves lead to starvation of certain user requests that could possibly be served. This is because a successful response requires that it be served by all the microservices in its execution path, while the existing works for a microservice fail to consider information about requests that are destined to be dropped at other overloaded microservices.

Starvation occurs even in an overload scenario that involves a very simple topology, as illustrated in Figure 1. Consider the topology with two microservices and two different APIs. API_1's execution path includes microservices M_A and M_B, and API_2's execution path includes microservice M_A. Let us assume that both microservices M_A and M_B are suffering from overload.

Existing overload controls perform load control at each microservice without considering whether the API's execution path includes other overloaded microservices. At M_A, it will limit API_1 and API_2 until the overload is relieved at M_A. Meanwhile, at M_B, it will limit API_1 until the overload is relieved at M_B. If M_B is serving fewer

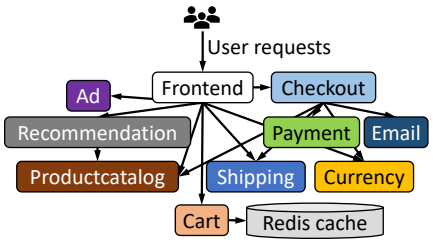


Figure 2: A microservice topology of the open-source benchmark called Online Boutique [21].

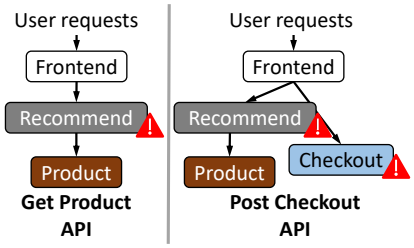


Figure 3: Execution path of Get Product API and Post Checkout API under overload scenario.

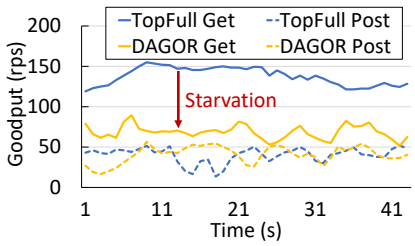


Figure 4: Concurrent load control causes starvation. Get Product (solid yellow line) is being starved with DAGOR.

API_1 requests than API_1 requests served in M_A, this leads to the following problem. M_A will waste resources on constructing partially processed responses for API_1, which will ultimately be rejected in M_B. At M_A, API_2 is starved by API_1 due to the suboptimal load control decision. On the other hand, the opportunity lies in enhancing the goodput of API_2 by allocating more resources to it within M_A, considering that the goodput of API_1 is limited in M_B.

Empirical demonstration. We demonstrate that this starvation occurs in real applications using the Online Boutique microservice application with DAGOR [53], an existing overload control framework. During overload control at individual microservices, the admitted rates among APIs are shed equally by DAGOR. It informs the upstream microservice about its current admission level, but lacks path-wise execution information. Thus, DAGOR cannot consider APIs' availability in other microservices, which possibly results in wasted resources on building partially processed responses that are ultimately rejected at other microservices. We generate an overload scenario and observe that such load control causes starvation, serving 65.5% less requests compared to our API-wise load control. The overload is generated by increasing the load of *Get Product* and *Post Checkout* API requests which cause overload at individual microservices, *Recommend* and *Checkout* as depicted in Figure 3. The goodput is measured by collecting the number of successful responses that finished within 1 second, a pre-determined latency SLO. Figure 4 shows the goodput degradation due to starvation from concurrent load control (DAGOR). Our globally managed load control (TopFull) serves 1.9x more *Get Product* requests while serving the same amount of *Post Checkout* requests compared to the DAGOR.

Stavation is easily triggered and frequent. We observe that the overload scenarios vulnerable to starvation occur commonly in the real world. When an API experiences overload at multiple microservices in its execution path, it triggers possible starvation upon other APIs sharing those microservices. This is because the existing approaches load-control the requests indiscriminately at each microservice regardless of the type of API, resulting in suboptimal resource utilization as in Figure 1. In Online Boutique application, we generated a traffic surge in an API one at a time upon *postcheckout*, *getproduct*, *getcart*, *postcart*, and *emptycart*. It is observed that any traffic surge in a single API always generates multiple overloaded microservices, the overload scenario vulnerable to starvation. Among the five APIs we tested, it creates 3.4 overloaded microservices on average.

We also analyze 23,481 microservices from Alibaba trace data [34] that include microservice resource utilization data and traces that depict API execution paths. At a given time, we classify microservices as "overloaded" when their CPU utilization is over 0.8. We identify cases like Figure 1, in which an API is involved in multiple overloaded microservices while having more than one contending API at the overloaded microservice. We observe that 44.4% of APIs among those involved in overloaded microservices were potentially vulnerable to starvation.

3 GOAL AND APPROACH

Goal. Our goal is to identify the optimal target rate limits of external APIs that maximize the total number of responses served within SLOs. This overload control problem can be formulated as follows:

$$\begin{aligned} &\text{Maximize } G(x_1, x_2, \ldots, x_n) \\ &\text{s.t. } W(x_1, x_2, \ldots, x_n) \le A(r_1, r_2, \ldots, r_m), \\ &\quad x_1, x_2, \ldots, x_n \ge 0. \end{aligned} \tag{1}$$

G is the total number of good responses within SLOs for each external API, where x_i represents the target rate limit of the load of the external API of the user interface i. Note that unlike existing approaches [26, 44, 53] that control the load at every microservice, TopFull only controls the load of external user-facing APIs. W and A are functions that represent the complex interdependencies of the microservices system. W is the actual load that the underlying microservices experience as a result of the workload of external APIs, and A is the compound ability of the microservice system to service workloads given r_m, the resources allocated at the microservice m. The inequality refers to the condition that microservices do not experience overload, hence avoiding performance degradation.

Adaptive API-wise load control. We find the optimal rates among APIs sharing an overloaded microservice by API-wise adaptive load control that considers the underlying microservices' bottleneck signals. It is common to adaptively adjust the admitted rates of APIs' requests in search of the maximal utility while resolving an overload of a microservice. However, conventional approaches suffer from possible starvation since they indiscriminately load

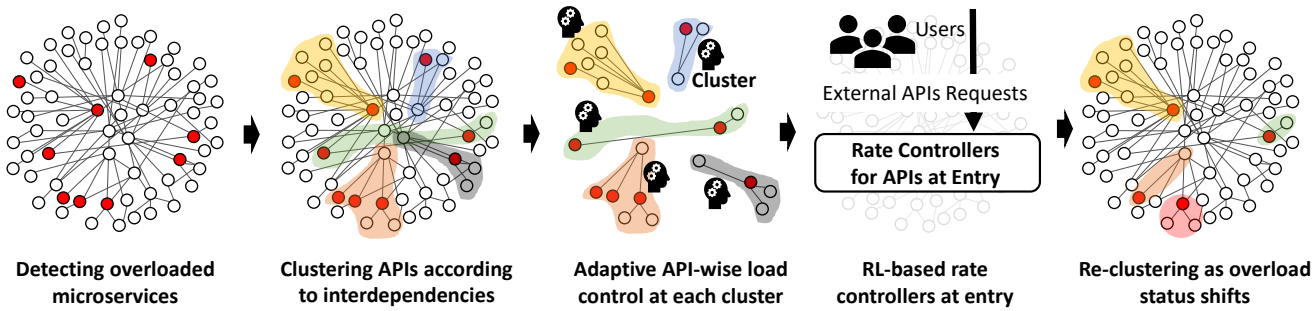


Figure 5: Abstract visualization of TopFull execution flow of load control.

control APIs sharing a bottleneck microservice. On the other hand, we differentiate load control decisions among the corresponding APIs according to the presence of other overloaded microservices in their execution path. When rate limiting, we reduce the rates of corresponding APIs equally to resolve the overload. After the overload is resolved, when we rate-increase the APIs, we inspect the existence of other overloads in their execution path and then only increase the APIs that no longer contain overloaded microservices. In this way, we avoid wasting the bottleneck microservice's resources on processing partially processed responses, which will be eventually rejected by other overloaded microservices.

Enabling parallel independent load control. To accelerate the overall load control process, we split microservices into smaller subsets and carry out load control in a parallel manner. Interdependencies between APIs and complex microservices make it difficult to control multiple APIs concurrently and discern the specific impact of each load control. Therefore, we decompose the original overload control problem into smaller sub-problems. Sub-problems are identified by finding independent subsets of constraints toward control variables, which correspond to overloaded microservices and APIs, respectively. In particular, each subset is made from clustering interdependent external APIs according to APIs' execution paths and the current status of overloaded microservices. A cluster is defined as a unique grouping of APIs, each interconnected by their reliance on the same set of overloaded microservices. These APIs are distinct in that they do not share their overloaded microservices with any other APIs outside the cluster. To form such clusters, we initially identify a group of APIs that utilize each overloaded microservice. If an API appears in multiple groups, we aggregate multiple groups into a single cluster, otherwise a group forms a single cluster. Therefore, once the clusters are formed, load-controlling APIs in a cluster do not have any effect on the other overloaded microservices in different clusters.

RL-based rate controller. We adopt an RL-based rate controller for rapid response to overloads and changes in the microservices resource allocations. For each set of APIs to load control, we apply our RL-based rate controller that decides its multiplicative step size of rate control. RL-based rate controller observes the current good-put and end-to-end percentile latencies, then performs a subsequent action that is applied to rates of the candidate API(s). The RL-based rate controller is responsible for rate-limiting APIs to resolve an overload, and also for recovery of rate-limited APIs when microservices are underutilized. The RL-based rate controller is trained in a Sim2real transfer learning scheme, where it is pre-trained with a simulator and then further specialized in the target real-world application.

4 TOPFULL DESIGN

TopFull consists of three system components: adaptive API-wise load control, clustering APIs for parallel load control, and an RL-based rate controller. The online end-to-end execution of load control takes place in the following order: clustering of external APIs and performing adaptive API-wise load control in each cluster with our RL-based rate controller. TopFull invokes this process when overload is detected and repeats the steps until it is resolved. Note that clustering happens first, and then load control takes place in parallel at each cluster. A load control cycle of the process is illustrated in Figure 5.

4.1 Adaptive API-wise load control

This module aims to resolve overload in a set of microservices while avoiding starvation, thereby maximizing overall goodput. Similar to the convention, we adaptively rate-limit APIs sharing a microservice to resolve overload and rate-increase APIs to find bottleneck microservices' maximum utilities. However, to avoid starvation, we differentiate our load control decisions among APIs regarding the existence of other overloaded microservices in their execution paths to guarantee APIs' end-to-end response. TopFull systematically resolves the overloaded microservice one at a time by rate-limiting the APIs that utilize the overloaded microservice. When TopFull increases the rates for previously rate-limited APIs, it does so after identifying the presence of overloaded microservices along their execution paths at the current moment, choosing the APIs that are not currently experiencing any overloads. By the iterative procedure of adaptive API-wise load control, TopFull resolves overloads while ensuring that bottleneck microservices are utilized to their full capacity.

We demonstrate the behavior of API-wise load control between interdependent APIs among multiple overloaded microservices. Without loss of generality, let's assume two overloaded microservices as shown in Figure 6, where Service A is the current target microservice to resolve overload, and Service B is the next subsequent target to resolve overload. Moreover, the overload scenario

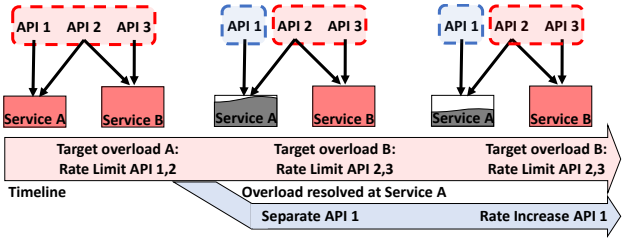


Figure 6: An illustration of API-wise load control.

Algorithm 1 Load control execution

```
1: function AdjustRate(Candidate_APIs, Action_RL)
2:     if isPositive(Action_RL) then
3:         targets ← Candidate_APIs.HighestPriority
4:     else
5:         targets ← Candidate_APIs.LowestPriority
6:     for each t in targets do
7:         t.rate ← t.rate · (1 + Action_RL)
```

vulnerable to starvation is reproduced by the presence of an API that uses both overloaded microservices, represented as API 2 in Figure 6. In conventional approaches, API 2 requests would likely be partially processed in either Service A or Service B, starving other APIs that utilize those microservices. In the beginning, APIs 1 and 2 will be rate limited by the rate controller to resolve overload in the current target Service A. Once Service A no longer experiences overload, API 1 now no longer involves overloaded microservices in its execution path, thus handled separately by a rate controller for possible recovery. On the other hand, API 2 is not a candidate for a rate increase since it is still involved in overloaded Service B. Next, Service B is now chosen as the target microservice and the rate controller will rate-limit APIs 2 and 3 to mitigate overload. During the process, Service A will be underutilized by further rate-limiting of API 2 which happens to share Service A. The separate rate controller for API 1 detects the change and adaptively recovers the rate limit of API 1. Eventually, all the microservices are relieved of overload and reach full utilization without building partially processed requests that cause starvation of other APIs.

To carry out API-wise load control, API execution paths are collected through a distributed tracing tool. We also gather the resource utilization status of microservices every second. The resource utilization is used as a signal indicating the overload status of underlying microservices. Moreover, we iteratively choose the overloaded microservice utilized by the fewest APIs as the target microservice for overload mitigation. This is because these particular microservices are less susceptible to being impacted by subsequent load control targeting other microservices. Although we adaptively restore rate-limited APIs in case of underutilization of bottleneck microservices, it is better to avoid excessive load shedding in the first place whenever possible.

Respecting the business priority. TopFull is capable of respecting the business priority by rate-limiting the lowest business priority API among the set of candidate APIs that utilize the overloaded microservice. The business priority is pre-determined values assigned to API types according to business logic or the operator. Business priority enforces APIs with higher business priority are assured to be processed prior to others. For example, DAGOR [53] performs admission control based on business priority at each microservice. In TopFull, we apply rate control decisions according to Algorithm 1. Given the target microservice, APIs that utilize the target microservice are identified as candidate APIs. Then, if the action is to decrease the rate limit, the action is applied to the API with the lowest business priority. Otherwise, when the action is to increase the rate limit, the action is applied to the API with the highest business priority. As a result, in the presence of the business priority, the load is distributed in a way to maximize the highest business priority API among the APIs that utilize the target microservice.

4.2 Clustering APIs for parallel load control

The purpose of clustering is to execute load control at each cluster in parallel, with a guarantee that the load control decision affects only the APIs within a cluster. To achieve this, TopFull clusters APIs exclusively so that they do not share any overloaded microservices with APIs outside the cluster. We detect overloaded microservices when the resource utilization of a microservice exceeds a predetermined threshold. By examining APIs' execution paths, we group APIs that share an overloaded microservice into a cluster. The equation below formally states this.

$$\begin{aligned} &\text{API } i,\ \text{API } j \in C_k \\ &s.t.\ \exists \text{OM} \in \text{API } i \cap \text{API } j \end{aligned} \tag{2}$$

API i and API j belong to the same cluster C_k, if any overloaded microservice, OM, exists on both of their execution paths.

Each cluster represents a smaller sub-problem having independent subsets of constraints. There exist no overloaded microservices that are shared between execution paths of two APIs in different clusters. For example, even if API 1 and API 3 do not directly share any overloaded microservices, they are clustered together if there exists API 2 that shares overloaded microservices with both APIs. Clustering allows TopFull to perform load control on each cluster in parallel, making it more scalable.

Re-clustering dynamically. TopFull re-clusters dynamically whenever the global set of overloaded microservices changes. The set of overloaded microservices changes for various reasons such as external workload transitions, additional resource allocation, or the result of overload controls. A cluster is transitive in nature. For example, after an overload on a microservice is resolved by TopFull, the cluster can be divided further if the set of APIs in the cluster no longer shares any overloaded microservices. The clusters are also aggregated to a larger cluster according to Equation 2 if an additional overload takes place.

APIs with branching execution paths. TopFull handles an API with a branching execution path as an API that is involved in every microservice in its possible execution paths. In microservices, some APIs are designed with branching execution paths that alter their operational flow within a microservice. The alteration is based on the specific content or parameters of the request. In certain microservices, the API request will branch out and follow a different

execution path. For such an API request, its actual execution path cannot be known at the entry point. However, the possible set of execution paths is predetermined, and we leverage this information during the clustering procedure.

4.3 RL-based rate controller

The goal of an RL-based rate controller is to utilize end-to-end performance metrics and decide the aggressiveness of rate control decisions for candidate API(s) at the entry to maximize the goodput. We demonstrate that a static load control policy that reacts to overload with fixed step size isn't effective (§6.2). Instead, an effective rate controller should make aggressive decisions in the initial phase of overload according to its severity and then finely adjust the rate-limit. In addition, it should recover from its previous rate limit decisions in response to resource allocation in the presence of an autoscaler (§6.3).

Existing approaches rely on local observations at individual microservices, such as resource utilization, processing delay, or queuing delay. They typically make static rate adjustments by simply classifying whether a microservice is overloaded or not. However, achieving end-to-end optimization requires more than simple heuristics based on local observations. Leveraging RL techniques capable of considering end-to-end performance metrics, such as goodput and end-to-end percentile latency, can provide significant benefits in this regard. RL-based rate controller can dynamically adjust its step size, unlike the static rate adjustments typically seen with heuristics, allowing for more flexible and adaptive optimization strategies.

TopFull employs an RL-based rate controller that observes the end-to-end performance metrics and adaptively adjusts the API's rate limit. Depending on the observed states, the RL determines the size of the multiplicative rate adjustment, which is applied to the API's rate limit. For training efficiency, TopFull employs Sim2real transfer learning scheme. The training involves the following steps: 1) Pre-training a base model from a simulated environment, 2) A pre-trained model is transferred to the target real-world application, 3) fine-tuning the transferred model at the target application by further training in the target application environment.

Sim2real transfer learning. Prior work on applying a machine learning approach to auto-scaling microservices [38, 39] directly uses the target microservices application to train a model. However, they are very costly and require a large amount of data collection through real-world interactions. Training involves an actual exchange of state, action, and next-state pairs between the real-world application and the RL model, and the model has to wait until the action takes effect in the real-world environment. Especially during the initial phase of training, the model produces random actions that are far from optimal and collects low rewards in most of the episodes, which makes training extremely cost-ineffective.

TopFull addresses the problem of costly training in real-world applications by adopting transfer learning in the domain of RL [29, 32, 45, 54]. We pre-train the RL agent for overload mitigation through a graph simulator that generates directed acyclic graphs (DAGs) representing simple execution paths in microservices. Then, in the specialized case, the RL agent is transferred to learn the application-specific behavior of the target real-world application. This transfer learning is possible because understanding the basic logic of overload control decisions helps RL to learn the base policy, whereas the remaining task is fine-tuning to the behavior of real-world microservices.

Simulator's design principle. The purpose of the simulator is to train the base policy for an RL-based rate controller—tightening the load thresholds when overloaded and loosening the load thresholds otherwise. Thus, we build a simple and lightweight graph simulator that simulates a graph of nodes, representing microservices.

In designing the graph simulator, we incorporate two major characteristics of microservices. First, client requests are completed by invoking a series of microservices, which is pre-determined as an execution path. Therefore, the end-to-end latency and good responses are aggregated through the microservices in the request's execution path. Second, to develop a robust policy toward the volatile performance of microservices experiencing overload, the latency and good responses at each node are added with random noise proportional to its scale of overload conditions.

The simulator contains DAGs, each DAG represents an API execution path, and each node in a DAG represents a microservice. Each node is assigned with latency and load capacity, which is randomly generated within a range. The node is classified as overloaded when requests exceed its load capacity. Depending on the fluctuation of the input workload, each node adjusts its latency and goodput following the rules stated below: 1) When overloaded, if a node experiences an increase in incoming rate, the node's latency increases and *goodput* decreases. 2) When overloaded, if a node experiences a decrease in incoming rate, the node's latency decreases and *goodput* increases. 3) When not overloaded, a node outputs low latency and *goodput* equals the incoming rate.

RL model design. The RL agent observes two state features that indicate the end-to-end performance metrics of the candidate APIs: 1) the ratio of *goodput* to the current rate limit, and 2) the end-to-end percentile latency of the candidate APIs. When multiple APIs are identified as candidate APIs, we calculate the ratio of the aggregate sum of goodput to the aggregate sum of the current rate limits for those candidate APIs. The highest end-to-end percentile latency among candidate APIs is used as the second feature.

The RL agent selects an action from the continuous space between -0.5 and 0.5, which is a multiplicative step size that adjusts the current rate limit at the entry. The action is applied to the target APIs chosen from the candidate APIs (Algorithm 1). The reward is the change in total *goodput* among APIs in the cluster, minus a penalty for violating the latency SLO.

$$Reward = \Delta Goodput - \rho \cdot \phi(Latency, SLO) \tag{3}$$

The first term in Equation 3 is the objective term to maximize the increase in total goodput. The penalty function $\phi(\cdot)$ in the second term outputs $max(0, Latency - SLO)$, and ρ is a penalty coefficient.

Base model training. In every training episode, the simulator randomly generates DAGs, node characteristics, and the incoming request rate. The number of DAGs and nodes consisting of the DAGs is given as hyperparameters, where we used 1-3 for the number of DAGs and 1-5 nodes for each DAG. We use Proximal Policy Optimization (PPO) [42] for training. PPO is consistent with our

Parameter	Value
Steps in episode	50
Learning Rate	5×10^{-5}
Kullback-Leibler coeff	0.2
Kullback-Leibler target	0.01
Minibatch size	128
PPO Clip Parameter	0.3

Table 1: Reinforcement learning training parameters.

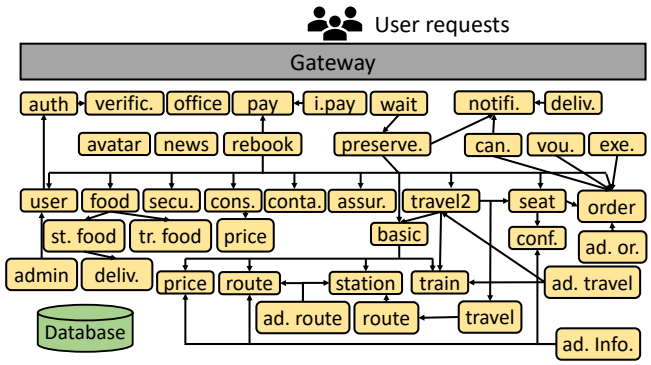


Figure 7: A microservice topology of the open-source benchmark called Train Ticket [12]

continuous action space, enabling fine-grained load control. During the training, we checkpoint the RL model every 50 episodes. We select the pre-trained model by validating the performance of the checkpointed RL models on a fixed set of scenarios in the simulator.

Specialization. The pre-trained base model is further trained in the target real-world application to learn application-specific characteristics. For each episode, we randomly generate workloads composed of different external APIs for the application. At each step, for a given set of APIs, an RL-based rate controller observes state features, makes rate control decisions, and then receives the reward. We also checkpoint RL models and select the best model by validating them on a fixed set of overload scenarios in the target application. The process of specialization from the pre-trained base model can be iterated upon to adapt to environmental changes, such as updates in microservices. In our evaluation, we quantify training time and cost.

5 IMPLEMENTATION

We implement our end-to-end load controller in three modules: distributed tracing collector, TopFull, and rate limiter. Distributed tracing collector tracks the resource utilization of each microservice using cAdvisor [17], and collects traces for each API request, including the execution path, individual microservice latency, and end-to-end response latency using Istio [14]. The execution paths for APIs are built from the data gathered from the distributed tracing collector. At each cluster, TopFull's RL-based rate controllers make load control decisions every 1 second. The rate limiter is attached at the entry and performs load control according to the given rate limit thresholds.

For load control, we use a rate limiter based on a token bucket algorithm. The rate limiter is implemented in Go. We use RLlib [23] for implementing the PPO algorithm. We adopt the default PPO settings [24] as illustrated in Table 1.

Baseline implementation and parameters. We implement DAGOR [53] and Breakwater [26] algorithms in Online Boutique. In DAGOR, every request is assigned a pre-determined business priority for API type and random user priority at the entry points. For every second, each pod sets a priority threshold according to a queuing delay and the number of incoming requests during the last second. The priority threshold is piggybacked to its upstream service. For Breakwater, it is implemented in each pod regarding gRPC exchange between pods as a client-server relationship. Each pod informs its token thresholds to the upstream pods, where upstream pods generate tokens following the thresholds.

Real-trace Demo implementation. To validate TopFull's performance with a large microservice application, we built a demo microservices following Alibaba microservice trace [34]. Our real-trace demo is composed of 127 microservices and 25 APIs with a total of 43 execution paths. Among 25 APIs, 8 APIs have branching execution paths of up to 6. In our overload experiments, 13 microservices are designed to be overloaded by imitating microservice utilization data from the trace. The execution path is realized by building connections between microservices with a simple RPC implementation. The microservices utilization is imitated by including sorting and arithmetic operations in their internal logic.

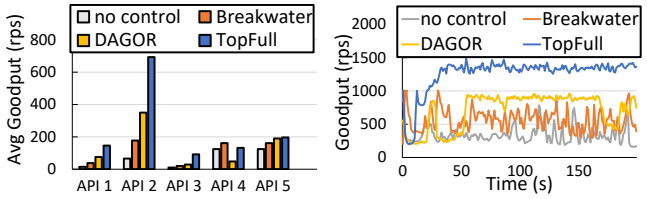
6 EVALUATION

We evaluate TopFull to answer the following questions:

- Does TopFull outperform existing overload control frameworks in microservices application? (§6.1)
- How much performance gain does each component of TopFull deliver? (§6.2)
- Does TopFull effectively prevent the performance degradation from transient overload with autoscaler? (§6.3)
- How effective is clustering and how much overhead cost does TopFull require? And how much benefit does transfer learning bring to the RL-based rate controller? (§6.4)

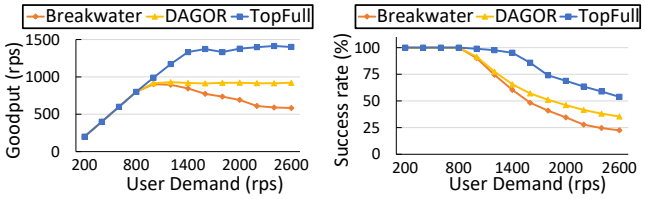
Experimental Setup. For the evaluation, we use a real-trace demo application and two open-source microservice applications; Train Ticket [12] and Online Boutique [21]. Train Ticket contains 41 microservices and Online Boutique consists of 11 microservices. Real-trace demo application contains 127 microservices. Kubernetes [22] is used for underlying container orchestration and the Kubernetes autoscaler [11] is used as the autoscaler baseline.

We deploy microservices on Azure cloud [16] environment and dynamically scale up to 10 VMs for Kubernetes worker nodes on demand. We conduct experiments on Azure VM D48ds_v5 [13] with 48 vCPUs of Intel(R) Xeon(R) Platinum 8370C. For traffic generation, we use Locust [20] and two additional Azure virtual machines (D48ds_v5) for large-scale traffic generation.



(a) Average goodput per API (b) Total goodput time line

Figure 8: TopFull, DAGOR, Breakwater, no control: goodput under overload.



(a) Total Goodput (b) Success Rate

Figure 9: Performance of TopFull, DAGOR, Breakwater according to user demand.

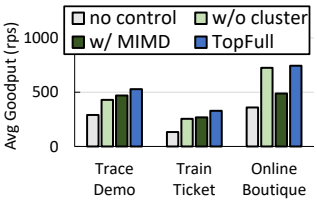


Figure 10: TopFull component-wise performance breakdown with different benchmarks.

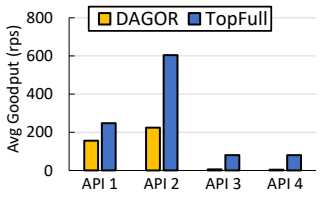


Figure 11: DAGOR vs. TopFull: average goodput per API when overload.

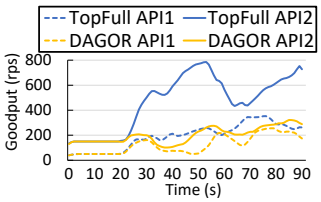


Figure 12: DAGOR vs. TopFull: goodput timeline of API 1 and API 2.

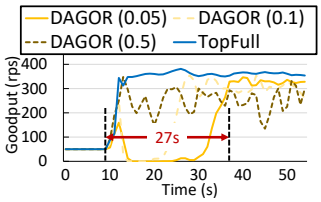


Figure 13: DAGOR vs. TopFull: adaptation speed with different parameters.

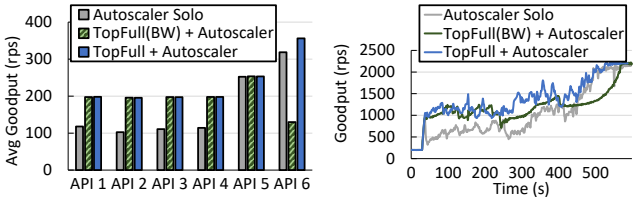
Benchmark Application Setup. We utilize open-source applications, Online Boutique, and Train Ticket. In Online Boutique, API 1, 2, 3, 4, 5 corresponds to *postcheckout*, *getproduct*, *getcart*, *postcart*, and *emptycart*, respectively. In Train Ticket, API 1, 2, 3, 4, 5, 6 corresponds to *high speed ticket*, *normal speed ticket*, *query order*, *query order other*, *query food*, and *query payment*, respectively. Across the benchmark applications, APIs are distinguished based on their URLs. Figure 7 shows the microservice topology of Train Ticket application.

In addition, We built a demo application from Alibaba trace data. Note that, our demo application is a partial representation of an actual trace facing overloads. From the data, at a given time t, we sampled among the microservices with CPU utilization higher than 0.8. We analyze every API involved in the sampled microservices and implement their entire execution path. The global topology of microservices is presented in Figure 20 in the Appendix A. Each blue node represents individual microservices and the connected arrow represents the calls between microservices. APIs are executed following the execution paths from the trace data.

6.1 End-to-end performance evaluation

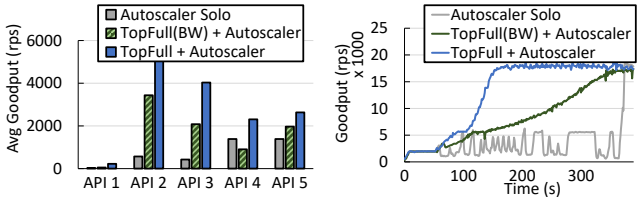
We compare the performance of TopFull and the other baselines by measuring the goodput per API under overload. Online Boutique is selected as a benchmark since it operates on gRPC [19], more compatible with Breakwater for remote procedure calls. The overload is generated from 2600 Locust [20] users invoking 1 request per second. Since Breakwater does not encompass a priority-based load control, we regarded all APIs as having the same business priority. Performance without overload control is also provided for comparisons. Figure 8 shows the average goodput per API and the total goodput during overload in an Online Boutique application. TopFull outperforms DAGOR by 1.82x and Breakwater by 2.26x on total average goodput under overload. In Figure 9, we compare the performance of each load control at different incoming request rates. Note that, once the user demands exceed the microservices resource capacity, the overload control maintains admitted rates regardless of additional traffic increase.

In Figure 9, we observe that TopFull and DAGOR show consistent performance with respect to the number of user demands, while Breakwater suffers from further performance degradation when user demands increase. This is because TopFull load control at the entry and DAGOR utilizes user priority-based admission control that enforces a consistent control standard at individual microservices. DAGOR's user priority is assigned randomly when a user request arrives at entry. The given priority value (integer between 0 and 127) is inherited by all the sub-requests that occur from the original user request. In DAGOR, each microservice sets different admission thresholds according to local overload metrics, however, each microservices admit the requests with a consistent priority standard. On the other hand, Breakwater's design is based on a single-tier client-server relationship, therefore requests are randomly shed at individual microservices. Assume that API 1 passes through microservices A and B, and that the load control rejects requests with probability p at both microservices due to overload. The probability of the request being served successfully becomes $(1-p)^2$. In such a multi-tier environment, either load control should be held at the entry similar to TopFull, or a design component, similar to DAGOR, that utilizes consistent standards across microservices is required.



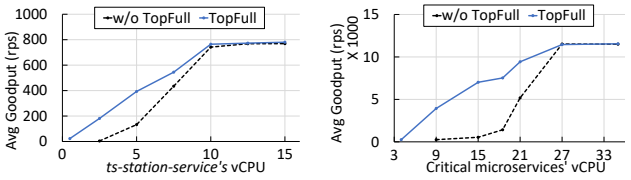
(a) Average goodput per API (b) Total goodput time line

Figure 14: Train Ticket: Performance under traffic surge.



(a) Average goodput per API (b) Total goodput time line

Figure 15: Online Boutique: Performance under traffic surge.



(a) Train Ticket (b) Online Boutique

Figure 16: Average goodput under traffic spikes according to allocated vCPUs

6.2 Component-wise benefit

Performance breakdown. To quantify the contribution of each design component, we measure the average total goodput without each component. To measure the effectiveness of the RL-based rate controller, we implement TopFull with a threshold-based multiplicative increase/decrease rate controller in replacement of the RL. Given the highest end-to-end percentile latency of the target APIs, it makes a 0.05 multiplicative decrease to the current target rate limit when the latency exceeds the SLO. It makes 0.01 multiplicative increase step to the target APIs, otherwise. To measure the contribution of external APIs clustering, we disable the initial clustering step, which results in solving the original problem without fragmenting into sub-problems. For comparison, we also experiment goodput of the microservices when there is no overload control at all.

The performance degradation from deactivating each component is shown in Figure 10 for real-trace demo application and both open-source applications. In real-trace demo, when TopFull employs MIMD instead of RL, the goodput decreases by 11.1%. TopFull without clustering of APIs represents the naïve sequential load control without parallel execution. In this case, the goodput degrades by 18.7%. In Train Ticket, when TopFull is used with MIMD component, the goodput decreased by 18.4%. TopFull without clustering, the goodput degrades by 22.5%. From these, we observe that TopFull's ability to react to overload in parallel brings significant benefit. In Online Boutique, the goodput decreased by 34.4% with MIMD. Without dynamic clustering of APIs, the goodput decreased by 2.6%. This degradation is much less compared to Train Ticket, as Online Boutique had a bottleneck microservice that is shared by most of the APIs. Since its APIs share an overloaded microservice most of the time, it cannot be broken down into many small pieces. Our analysis with real-world traces in Section 6.4 shows that this is very unlikely to happen, but many clusters are formed in real-world applications.

Overcoming starvation. We further analyze the benefit of adaptive API-wise load control designed to avoid starvation. We compare the load control behavior of DAGOR and TopFull with Online Boutique under overloads. Since DAGOR is designed to operate with business priority, we also compare per API performance after assigning APIs with different business priorities. Among API 1, API 2, API 3, and API 4, the former APIs are assigned a higher business priority than the latter APIs. Figure 11 shows the performance comparisons between DAGOR and TopFull. TopFull achieves 2.60x higher goodput on average. With DAGOR, we observe that APIs with lower business priority experience severe starvation. TopFull achieves higher goodput for APIs with lower business priority while guaranteeing as much goodput for APIs with high business priority. TopFull serves 1.58x more requests for API 1 with the highest business priority, and 7.55x more requests for API 2 with the second-highest business priority compared to the DAGOR. While the performance gain on API 1 is the result of our RL-based rate controller, the 7.55x gain from API 2 is due to DAGOR's severe performance degradation at API 2 which is starved by API 1. The API with the lowest business priority among the four APIs, API 4, is starved most by DAGOR's load control decisions, which results in TopFull serving 22.45x more compared to the DAGOR in terms of average goodput.

Figure 12 provides a detailed timeline of load control for both TopFull, and DAGOR. API 1 is *Post Checkout* API, and API 2 is *Get Product* API, sharing *Recommend* and *Product* microservice as illustrated in Figure 3. In local overload at *Product* microservice, DAGOR prioritizes business logic and sheds all the lower business priority API that passes *Product* microservice. On the other hand, TopFull manages the load between API 1 and API 2. TopFull also prioritizes API 1 and rate-limit API 2 at resolving overload at *Product* microservice, but when resolving overload at *Checkout* microservice, API 1 is rate-limited. In response, TopFull re-increases the rate-limit of API 2 to fully utilize the *Product* microservice. Note that although API 1 has higher business priority, the rate isn't increased when it is still involved in another overloaded microservice.

Benefit of RL-based rate controller. Figure 13 shows the adaptation speed of DAGOR with different step sizes and TopFull. The overload is generated with single *Post Checkout* API, focusing only on the effectiveness of the rate controller. TopFull takes 5s to reach

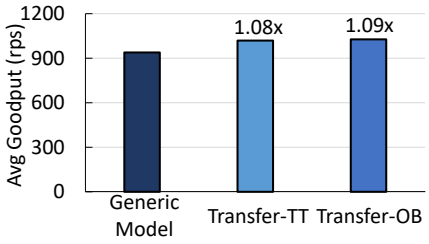


Figure 17: Average goodput of different RL models under traffic surge.

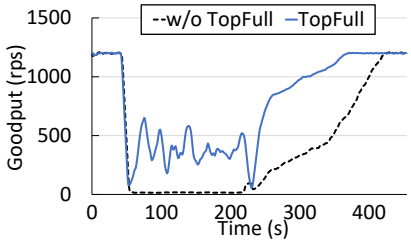


Figure 18: TopFull's adaptation toward temporary pods failures.

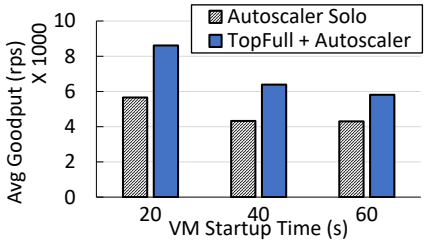


Figure 19: Average goodput with different VM startup time.

Rate Controllers	DAGOR (0.05)	DAGOR (0.1)	DAGOR (0.5)	TopFull (RL)
Convergence Speed	27 s	19 s	∞	5 s

Table 2: Convergence speed after overload between TopFull and DAGOR with different hyperparamters.

the maximal goodput whereas default DAGOR takes 27s to reach the maximal goodput after overload takes place. Such a difference is made since the RL-based rate controller makes fine-grained actions (from −0.5 to 0.5) which multiplicatively adjust the rate limit. TopFull observes the current status of microservices and takes aggressive actions if needed. In contrast, DAGOR only makes static decisions of 0.05 multiplicative decreases when microservice experiences overload, and 0.01 multiplicative increases when overload is no longer detected. The comparison of the convergence speed is provided in Table 2. The parameters can be tuned, but setting a bigger step size would harm the stability of the load control system while a smaller step size would make convergence even slower.

6.3 Performance evaluation with autoscaler

Performance under traffic surge. Microservices employ an autoscaler to cope with changes in traffic demand. However, autoscalers suffer from a fundamental limitation in that it takes several seconds to minutes to provision extra resources. We show how much performance gain TopFull can achieve under traffic surge when applied to the microservice application with autoscaler compared to the standalone autoscaler. We deploy a Kubernetes (k8s) autoscaler to the microservices for the experiments. TopFull is operated at the master node along with K8s autoscaler. The traffic surge is generated by increasing the number of users, which invokes multiple external APIs of the benchmark applications.

In addition, we include the comparison with TopFull(BW), which is TopFull with breakwater rate control algorithm instead of the RL-based rate controller. Breakwater increases the admitted rate additively proportional to the number of users when the measured delay is less than the target delay. It multiplicatively decreases the admitted rate proportional to the level of overload, which is represented as the difference between the measured delay and the target delay. We fine-tuned hyperparameters that influence the increase/decrease step size, ensuring that TopFull(BW) effectively handles the traffic surge. TopFull's RL-based rate controller shows a much faster adaptation speed to resource allocations from an autoscaler compared to the heuristic iterative algorithm.

As shown in Figure 14(a) and Figure 15(a), TopFull with the autoscaler achieves a higher average goodput at every APIs compared to the standalone autoscaler and TopFull(BW) with the autoscaler. Figure 14(b) and Figure 15(b) show that TopFull with the autoscaler consistently serves more goodput compared to the autoscaler alone throughout the time period of the traffic surge experiment. According to the resource allocation from the autoscaler, TopFull adaptively increases the rate limits of APIs to serve more goodput. The results also show that TopFull(BW) handles the initial traffic surge as much as TopFull but is slower at increasing the admitted rates when additional resources are allocated by the autoscaler. In Train Ticket, TopFull serves 1.38x higher average goodput during traffic surge compared to the autoscaler solo while using the same number of vCPUs. In Online Boutique, TopFull serves 3.91x higher average goodput during a traffic surge compared to the autoscaler solo. TopFull also serves 1.75x, and 1.19x higher average goodput compared to the TopFull(BW) in Train Ticket and Online Boutique, respectively. Online Boutique showed significant performance degradation during the traffic surge because *Recommendation* microservice's pods completely failed at the initial traffic surge. Although the autoscaler provided more *Recommendation* pods, they kept failing until enough pods are allocated at once. In microservices, without an overload control system, such pod failures can occur when liveness and readiness probes fail due to sudden overload.

Resource saving under traffic surge. Performance degradation due to traffic spikes can be alleviated by either load control or overprovisioning. We show the potential resource saving of TopFull by comparing the performance in terms of average goodput with and without TopFull while varying the degree of overprovisioning for critical microservices. For the traffic spikes, we generate a temporary load increase that lasts for two minutes. We evaluate the performance gain of microservice applications by introducing TopFull.

Figure 16(a) and Figure 16(b) show the average goodput according to the number of allocated vCPUs on critical microservices for Train Ticket and Online Boutique, respectively. The more vCPUs are allocated in advance, it represents the more overprovisioning in microservices. If sufficient vCPUs are overprovisioned to handle sudden traffic spikes, TopFull shows no additional gain, yet it requires many unnecessary vCPUs to be provisioned all the time. In Train Ticket, TopFull shows the same or higher average goodput with up to 50% fewer vCPUs compared to Train Ticket without

TopFull. TopFull achieves 2.98x higher average goodput compared to Train Ticket without TopFull when 5 vCPUs allocated. In Online Boutique, TopFull shows the same or higher average goodput with up to 57% fewer vCPUs compared to Online Boutique without TopFull. TopFull achieves 12.96x higher average goodput compared to Online Boutique without TopFull when 15 vCPUs allocated.

6.4 TopFull Deep Dive

Scalability and effectiveness of clustering. The clustering makes TopFull scalable because the load control is applied to each cluster in parallel. However, it is truly scalable only if we can divide the problem into many small clusters. We verify this using the Alibaba trace data [34] that contains a trace of 23K microservices. Among the overloaded microservices, 59% of them do not share any overlapping APIs with the other while the remaining 41% have overlapping APIs with very few microservices forming an average of 2.38 microservices that share any common APIs. At a given time, only up to 68 microservices are overloaded among 23,481 microservices, experiencing CPU utilization over 0.8. From our clustering, the initial problem with 68 overloaded microservices (constraints) is divided into 57 independent clusters with each sub-problem containing 1.19 constraints on average.

Online deployment overhead cost. During one iterative cycle of load control from TopFull, the overhead lies in mainly two parts: 1) building clusters, and 2) RL-based rate controllers making a load control decision for each cluster. Clustering requires an iteration over overloaded microservices to test the membership of the microservice in any API execution path, which is a simple hash table lookup. Our RL model is lightweight, having two-dimensional state space and one-dimensional action space. We exclude the cost of monitoring microservices resource utilization which is often collected as default for maintenance and autoscaling. In Train Ticket application with 41 microservices, clustering consumes 1.26×10^6 cycles on average. A single RL inference of our trained model takes 2.33×10^6 cycles. In our experiment setting, one designated CPU core from Azure VM D48ds_v5 [13] with Intel(R) Xeon(R) Platinum 8370C is capable of making load control for approximately 15000 microservices with 1000 independent API clusters for parallel RL decisions.

Training cost benefit from transfer learning. We evaluate the training cost of TopFull's RL-based rate controller and estimate the benefits in training time and cost by adopting the transfer learning scheme. It took 6 hours of training for the generic model to learn 48,000 episodes on our local machine with GeForce GTX 1080. Then, we further trained the generic model in the real-world microservice application for additional 800 episodes which took 12 hours of real-world sampling time. It takes at least one master node, one worker node, and one load generator node to run the microservice application for training an RL model. Following the pricing model of Azure VM D48ds_v5 [13], the cost of real-world training is $8.1 per hour with minimal three nodes to run a microservice application. TopFull's transfer learning scheme costs about $97.2 for real-world training. In contrast, without the transfer learning scheme we propose, it requires 30 days of sampling in the real world for the RL model to learn 48,000 episodes which costs about $5,832.

Performance gain of transfer learning. We evaluate the performance gain of transfer learning by comparing the average goodput of different RL models in the Train Ticket application under traffic surge. The pre-trained base model, the checkpointed model at 48,000 episodes trained with our simulator, is chosen for the experiments. We transfer the pre-trained model to the Train Ticket application and Online Boutique application to generate Transfer-TT and Transfer-OB models, respectively. Transfer-TT is obtained through further training of 800 episodes in the Train Ticket application, which took 12 hours of real-world interactions. Transfer-OB is also obtained through further training of 800 episodes in the Online Boutique application. We validate our pre-trained model, Transfer-TT, and Transfer-OB through an overload scenario on the Train Ticket application. The performance of each model is illustrated in Figure 17.

The transfer learned model serves 8-9% more requests compared to the base model. The gain comes from experiencing a real-world application through transfer learning. Note that the base model itself shows a reasonable performance by achieving an average goodput of 939 rps during a traffic surge, which is a 1.13x higher value compared to the autoscaler standalone which serves 829 rps on average.

Performance under failures. A microservice application can experience overload due to internal instance failures other than external traffic increases [10]. We also demonstrate how TopFull adapts toward the temporal failures of microservice instances. We design a scenario where *ts-station* microservice's pods suffer partial pods failures with the Train Ticket application. We delete 25 pods among 35 pods of *ts-station* microservice at time 50s. Then, Kubernetes automatically starts scaling 25 pods to maintain the number of 35 healthy *ts-station* pods. In Figure 18, we can observe that APIs suffer from severe goodput degradation immediately after pod failures. Without TopFull, microservices serve almost zero goodput until the failures are recovered even though 10 *ts-station* pods are alive. On the contrary, TopFull detects overload in *ts-station* and starts load control on APIs that pass *ts-station* microservice, guaranteeing goodput that can be achieved with 10 *ts-station* pods. The ability of TopFull to handle overloads does not depend on the causes.

Sensitivity to delay in scaling the underlying resources. We conduct sensitivity tests on TopFull's performance toward different VM startup times with the Online Boutique application. In Azure Cloud, VM startup time varied significantly by the time of the day, from less than 30s at 04 UTC and up to 267s at 16 UTC. Therefore, we emulated different VM startup times to accurately measure the effects of VM startup times. Instead of starting VM on demand, we run all 10 worker VMs and emulate VM startup time whenever the number of requested vCPUs exceeds multiples of 48. Following the empirical study [31] that reports AWS and GCP VMs show 41-124s of startup time, we tested TopFull with 20s, 40s, and 60s VM startup times.

Figure 19 shows the result of average goodput during the traffic surge that lasted 160 seconds. Both autoscaler standalone and TopFull with autoscaler show higher average goodput when VM startup time is reduced. Also, the sensitivity test shows that TopFull still shows up to 1.52x higher average goodput compared to

autoscaler standalone. Even with a short VM startup time of 20 seconds, TopFull still achieves a performance gain as it operates on a smaller timescale than autoscaling.

7 RELATED WORK AND DISCUSSION

Microservice overload control. Recent works have evolved overload control for Internet-scale services [26, 27, 37, 44, 52, 53]. Breakwater [26] proposes an overload control framework for low-latency Remote Procedure Calls (RPCs). It successfully challenges and solves overload control, where the communication and request drop costs are similar to the cost of processing a request, yet their work is subjected to a single-tier client-server relationship. Breakwater falls short in handling multi-tier microservices. DAGOR [53] considers the sub-requests generated from client requests. A client request is assigned with priority value at the entry point, and the sub-requests from the client requests inherit the same priority value. DAGOR performs admission control on individual microservices according to the priority threshold. As a result, despite the distributed overload control at individual microservices, DAGOR prevents random drop of sub-requests from the same client request at different microservices by ensuring a consistent admission standard. However, DAGOR still suffers from starvation.

WISP [44] aims to achieve end-to-end performance objectives with rate limiters at each microservice and request schedulers. WISP collects downstream microservices' admission rates and applies a priori weights to make rate-limit decisions at the upper microservices trying to rate limit at the upper layer as much as possible. Nevertheless, their request drop policy makes them vulnerable to the random sub-request drop identified by DAGOR. Moreover, WISP does not consider the contending relationship between client requests followed by dependencies with multiple overloaded microservices, leaving it vulnerable to a starvation problem. Note that none of the existing works is capable of execution path-aware adaptive API-wise load control on external APIs at entry like TopFull. Aequitas [52] provides a distributed sender-driven admission control scheme to guarantee network-specific SLOs. However, it is orthogonal to ours, as it only targets achieving network latency SLOs.

Autoscaling. Autoscaling is a feature that allows automatic allocations of resources to the current microservice application's user demands. In microservice orchestration, autoscaling mainly takes three forms; horizontal pod autoscaling, vertical pod autoscaling, and cluster autoscaling. Horizontal pod autoscaling makes resource allocation decisions by scaling the number of instances. Vertical pod autoscaling allocates more resources such as CPU or memory, to existing pods. Cluster autoscaling is the scaling of cluster nodes from the cloud provider according to the pending pods. Cluster autoscaling can take several minutes to complete [18]. According to a 2021 survey [31], commercial cloud providers, such as AWS and GCP typically take 41s to 124s to startup despite significant recent improvement [36]. The actual startup time varies depending on factors, such as cloud provider, time of day, region, instance type, and OS type [31, 36].

Numerous works have advanced autoscaling in microservices [25, 28, 30, 33, 35, 38–41, 46–51]. They focus on finding appropriate resource configurations to meet SLOs and performance goals. Despite their improvement, their operation timescale is often non-negligible, and thus overload control is crucial in handling the transient overload before the full effect of auto-scaling takes place. Machine learning-based approaches take numerous inferences in the optimization process [38] that require tens of seconds for decision-making, or take multiple discrete action steps [39] to reach the stable state after traffic changes. Some studies [30, 33] operate on time scales of minutes due to the adoption of global optimization algorithms such as a genetic algorithm and multi-objective Bayesian optimization.

Demand shaping mechanisms. Our approach focuses on developing a load control policy that maximizes end-to-end successful responses within the latency SLO. To achieve our goal, we rate-limit each API at the entry so the underlying microservices application does not suffer performance degradation. On the other hand, there are alternative methods of demand shaping instead of simply rate-limiting the requests. Aequitas [52] utilizes fair weighted queueing with high/medium/low weight network QoS queues and an admission control scheme that downgrades excessive traffic according to the independently measured latency performance of each QoS queue. Moreover, Breakwater [26] allows clients to send requests when credits are received from the server, so load control is achieved in combination with client-side rate limiting and server-side active queue management. TopFull focuses on finding the optimal load at the entry to maintain the best performance and is orthogonal to the demand shaping mechanisms.

8 CONCLUSION

We present TopFull, an overload control at entry for microservices that maximizes the aggregate goodput under various overload scenarios by adaptively adjusting the target rate limits of external APIs. To enable the responsive overload control at scale, TopFull breaks down the problem into multiple independent pieces and performs parallel load control on each independent cluster. TopFull introduces novel design concepts; adaptive load management between interdependent APIs that perform execution path-aware load control, clustering external APIs that achieve scalability through enabling parallel load control, and employing RL-based rate controllers for fine-grained load control that maximize responses satisfying SLO. Compared to the standalone Kubernetes autoscaler, TopFull achieves up to 3.91x performance gain under traffic surge and endures traffic spikes with up to 57% fewer resources when deployed on microservices together with the Kubernetes autoscaler. TopFull outperforms existing works under overloads, serving more average goodput than DAGOR by 1.82x and Breakwater by 2.26x. In addition, TopFull can also adapt toward overload caused by internal failures.

ACKNOWLEDGMENTS

We thank our shepherd Radhika Mittal and the anonymous reviewers for providing helpful feedback and suggestions to improve our work. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00340099) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00418784).

REFERENCES

[1] 2006. Avoiding a Success Disaster. https://aws.amazon.com/ko/blogs/aws/avoiding_a_succ/.

[2] 2015. Microservices at Amazon. https://www.slideshare.net/apigee/i-love-apis-2015-microservices-at-amazon-54487258.

[3] 2016. Canada's immigration website crashed due to traffic surge. https://www.ctvnews.ca/canada/canada-s-immigration-website-crashed-due-to-traffic-surge-1.3152744.

[4] 2017. Airbnb, From Monolith to Microservices: How to Scale Your Architecture. https://www.youtube.com/watch?v=N1BWMW9NEQc.

[5] 2018. Internal documents show how Amazon scrambled to fix Prime Day glitches. https://www.cnbc.com/2018/07/19/amazon-internal-documents-what-caused-prime-day-crash-company-scramble.html.

[6] 2020. Microsoft Confirms March Azure Outage Due to COVID-19 Strains. https://visualstudiomagazine.com/articles/2020/04/13/azure-outage.aspx.

[7] 2020. Zoom suffers "partial outage" amid home working surge. https://www.datacenterdynamics.com/en/news/zoom-suffers-partial-outage-amid-home-working-surge/.

[8] 2020. "Google Down": How Users Experienced Google's Major Outage. https://www.semrush.com/blog/google-down-how-users-experienced-google-major-outage/.

[9] 2021. Microservices - Netflix Techblog. https://netflixtechblog.com/tagged/microservices.

[10] 2022. Handle partial failure. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/handle-partial-failure.

[11] 2022. Horizontal Pod Autoscaler of Kubernetes. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[12] 2022. Train Ticket: A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket.

[13] 2023. Dv5 and Dsv5-series. https://learn.microsoft.com/en-us/azure/virtual-machines/dv5-dsv5-series.

[14] 2024. 2022 Istio Authors. Version Istio 1.16.2. https://istio.io/latest/.

[15] 2024. AWS Health Dashboard. https://health.aws.amazon.com/health/status.

[16] 2024. Azure. https://azure.microsoft.com/.

[17] 2024. cAdvisor (Container Advisor). https://github.com/google/cadvisor.

[18] 2024. Cluster Autoscaler. https://docs.aws.amazon.com/eks/latest/userguide/autoscaling.html.

[19] 2024. gRPC: A high performance, open source universal RPC framework . https://grpc.io/.

[20] 2024. Locust: An open source load testing tool. https://locust.io/.

[21] 2024. Online Boutique by Google. https://github.com/GoogleCloudPlatform/microservices-demo.

[22] 2024. Production-Grade Container Orchestration. https://kubernetes.io/.

[23] 2024. RLlib: Industry-Grade Reinforcement Learning. https://docs.ray.io/en/latest/rllib/index.html.

[24] 2024. Source code for ray.rllib.algorithms.ppo.ppo. https://docs.ray.io/en/latest/_modules/ray/rllib/algorithms/ppo/ppo.html#PPOConfig.

[25] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro:{Performance-Aware} resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 623–643.

[26] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for {μs-scale}{RPCs} with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 299–314.

[27] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2023. Protego: Overload Control for Applications with Unpredictable Lock Contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 725–738.

[28] Byungkwon Choi, Jinwoo Park, Chunghan Lee, and Dongsu Han. 2021. pHPA: A Proactive Autoscaling Framework For Microservice Chain. In *5th Asia-Pacific Workshop on Networking (APNet 2021)*. 65–71.

[29] Carl Doersch and Andrew Zisserman. 2019. Sim2real transfer learning for 3d human pose estimation: motion to the rescue. *Advances in Neural Information Processing Systems* 32 (2019).

[30] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1994–2004.

[31] Jianwei Hao, Ting Jiang, Wei Wang, and In Kee Kim. 2021. An empirical analysis of VM startup times in public IaaS clouds. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 398–403.

[32] Jiang Hua, Liangcai Zeng, Gongfa Li, and Zhaojie Ju. 2021. Learning for a robot: Deep reinforcement learning, imitation learning, transfer learning. *Sensors* 21, 4 (2021), 1278.

[33] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. 2021. RAMBO: Resource Allocation for Microservices Using Bayesian Optimization. *IEEE Computer Architecture Letters* 20, 1 (2021), 46–49. https://doi.org/10.1109/LCA.2021.3066142

[34] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. 412–426.

[35] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2022. 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*. 62–77.

[36] Ming Mao and Marty Humphrey. 2012. A performance study on the vm startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 423–430.

[37] Justin J Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, et al. 2023. Defcon: Preventing Overload with Graceful Feature Degradation. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 607–622.

[38] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. 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*. 154–167.

[39] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[40] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2023. {AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 387–402.

[41] Krzysztof Rzadca, Paweł Findeisen, Jacek Świderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Paweł Krzysztof Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google Scale. In *Proceedings of the Fifteenth European Conference on Computer Systems*. https://dl.acm.org/doi/10.1145/3342195.3387524

[42] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[43] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 205–218.

[44] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*. 611–623.

[45] Matthew E Taylor and Peter Stone. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10, 7 (2009).

[46] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Eva Tardos. 2023. Karma: Resource allocation for dynamic demands. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 645–662.

[47] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y Yan. 2024. Autothrottle: A Practical {Bi-Level} Approach to Resource Management for {SLO-Targeted} Microservices. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 149–165.

[48] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, KK Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X Liu. 2022. DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing*. 16–30.

[49] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt. 2019. MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 122–132. https://doi.org/10.1109/ICDCS.2019.00021

[50] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 68–75.

[51] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 2021. 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*. 167–181.

[52] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. 2022. Aequitas: admission control for

performance-critical RPCs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 1–18.

[53] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 149–161.

[54] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.

Appendices are supporting material that has not been peer-reviewed.

A BENCHMARK APPLICATION

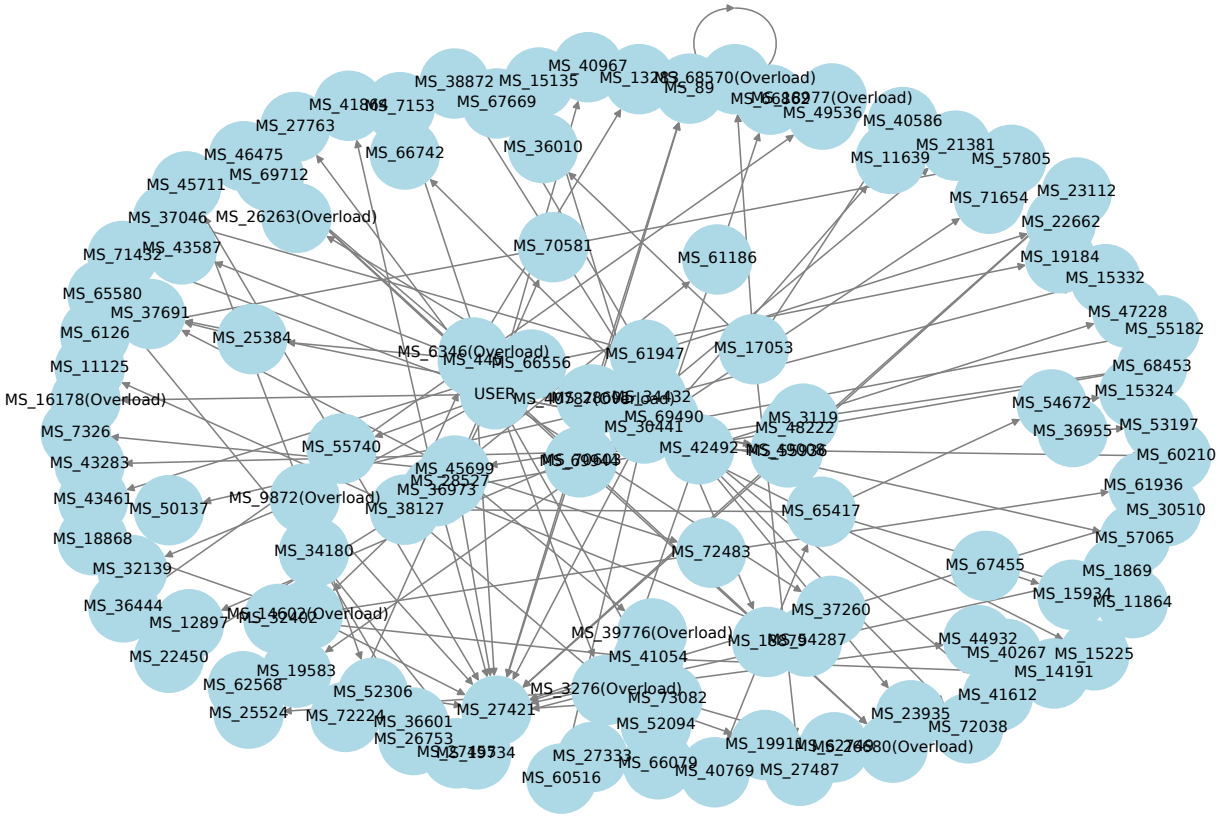


Figure 20: A microservice topology of real-trace demo application.