2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Ursa: Lightweight Resource Management for Cloud-Native Microservices

Yanqi Zhang†, Zhuangzhuang Zhou†, Sameh Elnikety‡, Christina Delimitrou††
Cornell University†, Microsoft Research‡, MIT††
yz2297@cornell.edu, zz586@cornell.edu, samehe@microsoft.com, delimitrou@csail.mit.edu

***Abstract*—Resource management for cloud-native microservices has attracted a lot of recent attention. Previous work has shown that machine learning (ML)-driven approaches outperform traditional techniques, such as autoscaling, in terms of both SLA maintenance and resource efficiency. However, ML-driven approaches also face challenges including lengthy data collection processes and limited scalability. We present *Ursa*, a lightweight resource management system for cloud-native microservices that addresses these challenges. Ursa uses an analytical model that decomposes the end-to-end SLA into per-service SLA, and maps per-service SLA to individual resource allocations per microservice tier. To speed up the exploration process and avoid prolonged SLA violations, Ursa explores each microservice individually, and swiftly stops exploration if latency exceeds its SLA.**

We evaluate Ursa on a set of representative and end-to-end microservice topologies, including a social network, media service and video processing pipeline, each consisting of multiple classes and priorities of requests with different SLAs, and compare it against two representative ML-driven systems, Sinan and Firm. Compared to these ML-driven approaches, Ursa provides significant advantages: It shortens the data collection process by more than $128\times$, and its control plane is $43\times$ faster than ML-driven approaches. At the same time, Ursa does not sacrifice resource efficiency or SLAs. During online deployment, Ursa reduces the SLA violation rate by 9.0% up to 49.9%, and reduces CPU allocation by up to 86.2% compared to ML-driven approaches.

I. Introduction

Production cloud services, such as Twitter and Netflix, are increasingly built as graphs of microservices [53], [6], [28], and deployed with cloud-native frameworks like Kubernetes [12], [4], [8], [1]. Despite the benefits of modularity and elasticity, resource management for microservices that must meet SLA constraints, e.g., end-to-end latency, is challenging, due to the diverse resource requirement of individual microservices and their inter-service dependencies [27], [28].

Resource management for microservices has been the topic of recent studies, and machine learning (ML) models, especially deep neural networks (DNN), have become a popular choice to address the complexity of microservice topologies. Previous studies have either used ML to predict important performance metrics, such as latency and load [60], [27], [22], [56], [41], or to directly adjust resource allocation [46], and demonstrate that ML-driven approaches outperform traditional techniques, such as autoscaling [18], in performance and resource efficiency.

However, ML-driven approaches still face key challenges limiting their adoption. First, ML-driven approaches typically require a lengthy exploration process to collect tens of thousands of data points to train the models, and therefore can not track changes in user behavior or handle the frequent updates to the microservice logic. Second, these ML models are on the critical path for every resource management decision, limiting the speed and scalability of resource management. Third, previous studies are evaluated using conventional benchmarks that use remote procedure calls (RPC) as the only method of inter-service communication and include only lightweight text processing in the business logic [28], [50], [62]. Modern cloud-native applications increasingly use both RPCs and message queues (MQ) [40], such as Kafka [3] and Redis streams [17], handle different request classes, such as image processing and ML workloads [49], [42], [47], and support different request priorities. Different request classes or priorities exhibit different latencies and therefore have different SLAs, making resource management more challenging.

To address these challenges, we propose *Ursa*, a lightweight resource management framework for cloud-native microservices. As a first step, we conduct a case study to understand how latency anomalies due to poor resource provisioning propagate through different communication methods. We show that backpressure is only significant for RPCs and is most pronounced in the parent service of the culprit tier (bottlenecked microservice). Given this, we design a method to determine the resource utilization threshold for each microservice that prevents backpressure in the application topology. By enforcing that the system operates in a backpressure-free zone, microservices in a topology can be treated as independent, reducing the number of factors the resource manager much consider from $O(N^2)$, where each pairwise microservice dependency must be accounted for, to $O(N)$, where the latency of individual microservices only depends on their own resource allocations. This greatly simplifies resource management, as most prior work resorts to complex ML models due to the need to capture the impact that microservice dependencies have on end-to-end performance.

Moreover, in a backpressure-free system, we develop a performance model based on mixed integer programming (MIP) which decomposes end-to-end latency SLA constraints into per-service latency constraints, and maps them to resource allocation thresholds for individual microservices. The model also supports specifying different SLAs for different request classes and priorities. To speed up the resource exploration, Ursa explores each microservice individually, and swiftly

stops exploration when latency exceeds SLA or the resource utilization reaches its backpressure-free threshold.

To better reflect modern microservices, we re-implemented several DeathstarBench applications [28] using Dapr [5], a popular microservice framework developed and used by major cloud providers. The re-implemented benchmarks use both RPCs and message queues (MQs), and implement different request classes and priorities, executing more diverse business logic than before. We compare Ursa to two representative ML-driven systems, Sinan [60] and Firm [46] as well as traditional autoscaling. Ursa reduces the required exploration time by more than $128\times$, making it more practical to track frequent changes to microservice logic. During online deployment, Ursa's control plane is $43\times$ faster than prior work, and Ursa reduces the SLA violation rate by 9.0% to 49.9%, and the CPU allocation by up to 86.2% compared to ML-driven approaches.

II. RELATED WORK

Microservices. Microservices have emerged as the dominant paradigm for interactive cloud services over the past few years. Unlike traditional monolithic architectures that contain the entire functionality of an application in a single binary, microservice architectures are graphical structures composed of tens or hundreds of single-purpose, loosely-coupled microservices, scaled independently, and in some cases implemented in different programming languages. The popularity of microservices is justified by several reasons, including flexible development, rapid iteration and fine-grained elasticity.

The emergence of microservices has also prompted efforts to benchmark and characterize them. Representative benchmarks include DeathstarBench [28] and ticket reservation [62], which implement several end-to-end user-facing applications. These benchmarks use RPCs and `http` RESTful API as the only inter-service communication methods, and mostly perform lightweight text processing in the business logic. More recently, Luo et al. [40] characterized microservices running on AliCloud and showed that MQs are common in practice, accounting for 23% of all communication methods, and the performance of microservices is most sensitive to CPU interference. Related work [61], [47], [42], [49] also shows that cloud-native applications implement a variety of business logic, including ML workloads, webserving, image and video processing, etc.

Resource management. A large body of work has focused on using ML to adjust resource allocation for microservices. These systems use ML to predict important performance metrics, such as latency and load or diagnose performance issues [60], [27], [22], [56], [41] or to directly adjust resource allocations [46], [60]. They demonstrate that ML-driven approaches outperform traditional approaches, such as autoscaling [18] and queuing based mechanisms [59], in terms of performance and resource efficiency. However, ML-driven, especially deep learning driven approaches also suffer from demand of large training dataset, difficulties in adapting to changing application logic and user workload, and limited control plane scalability. In addition to using ML, Zhou et al. [61] reduce request failure rate in WeChat microservices with overload control, Yang et al. [59] propose to identify bottleneck services in multi-phase applications by monitoring the queueing status, Suresh et al. [52] adopt deadline-based scheduling to improve tail latency in multi-tier workloads, and Sriraman et al. [51] present an auto-tuning framework for microservice concurrency, and show the impact of threading decisions on application performance and responsiveness.

Improving resource efficiency in cloud platforms in general is an important research area, and recent work [20], [26], [29], [31], [34], [35], [45], [54], [24], [36], [44], [25] has proposed several directions for how cluster scheduling frameworks can improve resource usage. Resource central [23] uses a set of ML models to predict VM performance metrics, such as CPU utilization and VM lifetime, Autopilot [48] uses an ensemble of models to tune container configurations, Ambati et al. [19] propose providing SLOs for resource harvesting VMs, and Narayanan et al. [43] propose to efficiently solve large-scale resource allocation problems by partitioning them to smaller problems. However, these proposals are mainly applicable to single VMs or containers, rather than microservices with directed acyclic graph (DAG) topologies.

III. BACKPRESSURE EFFECT

Backpressure is one of the major challenges in microservice resource management, and it refers to the phenomenon of the resource allocation of one service affecting the latency of upstream services, in addition to its own. In the presence of backpressure, modeling microservice latency requires modeling $O(N^2)$ dependencies, in the worst case, for a topology with N microservices, as each microservice's latency can be affected by the resources of its downstream services. Many microservice resource management frameworks use a centralized performance model that requires global information about all microservices to account for their inter-service dependencies, at the cost of scalability [60], [27], [22]. To achieve scalable resource management for microservices, we first conduct a case study to understand how backpressure propagates through different communication methods, including RPCs and MQs.

We study three types of chains connected by nested RPCs, event-driven RPCs, and message queues (MQs), respectively. Nested RPCs, as shown in Figure 1(a), are a synchronous system where, upon receiving the client request ($R0$), the upstream service forwards the request to the downstream service ($S0$) via RPC, blocks until the response is received ($R1$), and then returns the result to the client. Event-driven RPCs [58], as shown in Fig. 1(b), are more asynchronous in that upon receipt of a client request, the upstream service dispatches the request to another thread and returns immediately ($S0$), while the dispatched thread contacts the downstream service via RPC and waits for the response ($R1$). It is noteworthy that event-driven RPCs are still not fully asynchronous, as they involve a two-step process. Upon receiving a user request, the event handling thread works asynchronously, by yielding immediately after creating a daemon thread that further processes the request. However, the daemon thread talks to the downstream

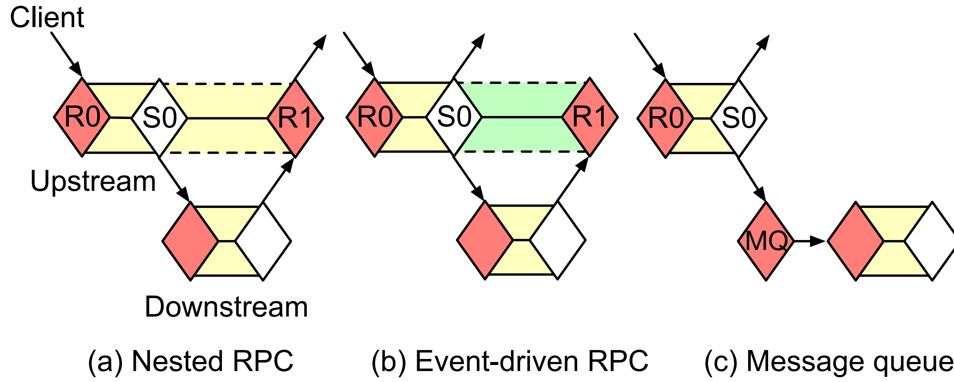

Fig. 1: Inter-service communication methods.

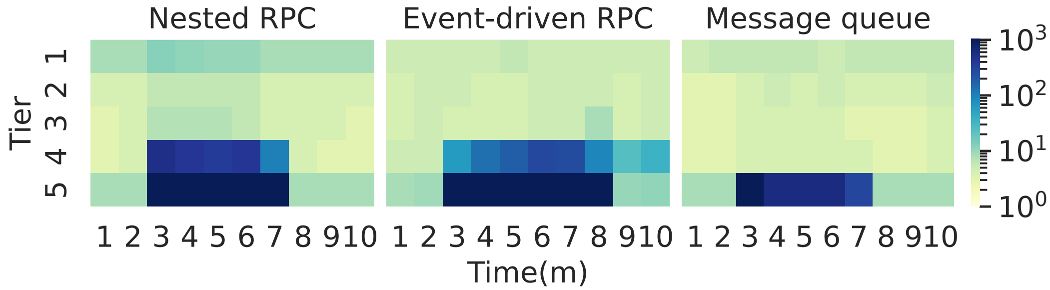

Fig. 2: Backpressure effects in a service chain.

service synchronously and informs the event thread when it receives the response. This interaction involves maintaining synchronous connections between upstream and downstream services. Message queues, on the other hand, are completely asynchronous. Unlike RPCs, MQs, such as Kafka [3] and Redis streams [17], mostly use a publish-subscribe paradigm, where publishers publish messages to topics hosted by the MQ and subscribers consume messages by subscribing to the topics. As shown in Figure 1(c), the upstream service does not directly contact the downstream service, but instead sends the client request to the MQ, and the downstream service gets new requests by polling the MQ.

Characterizing backpressure. We implement the RPC service chains with gRPC [9] and the MQ with Redis streams [17]. Each chain is configured to include 5 tiers, with each tier implementing a CPU-intensive loop as the request handler. We record the per-tier *response time* ($S0-R0$), which in the cases of MQ is the service latency, and in the case of RPC is the service latency *excluding the duration waiting for the downstream response*. We measure the tier's response time because it is closely related to the resource allocation of the tier itself. We stress test each service chain for 10 minutes, injecting performance anomalies into the leaf tier (tier 5) by throttling its CPU limit between minutes 3 and 6. The resulting backpressure behavior is shown in Figure 2, where each column on the x-axis represents a one minute interval, each row on the y-axis corresponds to a tier (tier 1 is client-facing), and the color of each cell highlights the per-tier 99th response time during that minute. For both nested and event-driven RPCs, significant backpressure is observed, especially for tier 4, the parent of the throttled leaf tier, and the backpressure rapidly diminishes up the call chain and becomes negligible above tier 3. In contrast, MQ shows no backpressure behavior, even on tier 4.

Fig. 3: Backpressure profiling engine architecture.

Determining conditions for negligible backpressure. Backpressure complicates resource management because the latency of a service also depends on the resources of its downstream services, in addition to its own. To simplify resource management, a natural approach is to determine safe CPU utilization thresholds to avoid backpressure in the system, as the performance of microservices is most sensitive to CPU utilization [40], [28]. The approach generalizes to other resources as well, for microservices with different resource profiles. To this end, we use a profiling engine with the 3-tier architecture shown in Figure 3, where the proxy acts as the parent service and simply forwards the request to the tested service via RPC. The engine gradually increases the CPU limit of the tested service, and monitors the latency of the proxy and the CPU utilization of the tested service, until the latency of the proxy converges. The convergence of proxy latency is determined by comparing the latency recorded under the last two CPU limits with Welch's t-test [57], a classical hypothesis testing method for identifying whether the means

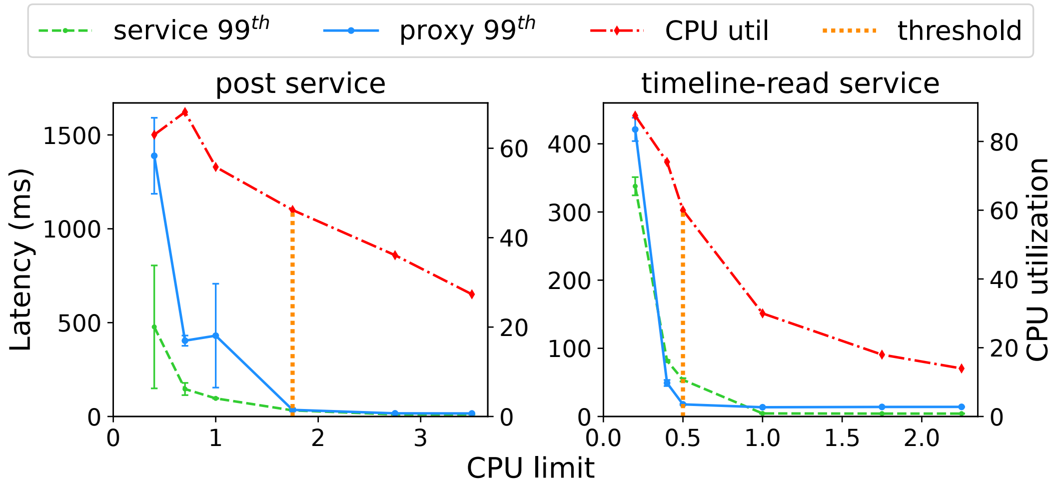

Fig. 4: Identifying backpressure-free CPU thresholds in a service mesh. We incrementally decrease the amount of resources allocated to the tested microservice, until we observe an increase in the latency of the proxy. Given the proxy's lack of computation activity, this increase signals the presence of backpressure. We use this threshold as the utilization the tested service should not exceed to avoid introducing backpressure to its parent tiers.

of the two sets of samples are equal. The CPU utilization just before the convergence of the proxy latency is then recorded as the threshold for not triggering backpressure. In order to account for complex invocation patterns, such as multiple upstream services sending requests to a downstream service concurrently, the workload generator synthesizes aggregate loads from different upstream services, so that the measured backpressure-free threshold is valid under fan-in/out patterns.

Figure 4 shows the profiling process for two microservices in a social network application similar to [28]: the post service, responsible for querying the contents of user post, and the timeline-read service, responsible for querying the post IDs in user timelines. The x-axis corresponds to the CPU limit of the tested service. The left and right y-axis correspond to latency and CPU utilization, respectively. The blue and green lines show the average 99th percentile latency of the proxy and the tested service under different CPU limits, and the error bars represent the standard deviation. The red line indicates the CPU utilization of the tested service. The orange line highlights the point where the proxy latency converges, and the corresponding CPU utilization is recorded as the threshold, 46.2% for post service and 60.0% for timeline-read service. When significant backpressure is observed, the 99th percentile latencies of the proxy and the tested service have already increased by more than $5\times$ and $10\times$.

Main insights. The study produces the following insights.

1) Backpressure complicates resource management because the latency of a service depends on resources of its downstream services, in addition to its own. Backpressure is common in RPCs but negligible in MQs.
2) Backpressure diminishes along the invocation chain. Of all the upstream services of the culprit, the parent service shows the most significant increase in latency.
3) The backpressure-free resource utilization threshold of a service can be profiled by monitoring the latency of an upstream proxy. By operating within the thresholds, backpressure can be avoided in the microservice system.
4) By eliminating backpressure, the number of factors required to model microservice latency becomes $O(N)$ with N services, because a microservice's latency is only a function of its own resources. Otherwise the number is $O(N^2)$ in the worst case, as a microservice's latency may depend on resources of its downstream services.

IV. Performance Model

When the system has negligible backpressure, the latency distribution of a microservice becomes mainly a function of its own resources. We now build a performance model for mapping microservice SLAs to resources. First, we decompose the end-to-end latency constraint for each request type to a set of per-microservice latency constraints. Second, we map each per-microservice latency constraints to resources for that service. In this section we describe the performance model we use, based on mixed-integer programming.

Decomposing end-to-end latency. Without loss of generality, we consider the end-to-end latency of a chain that handles a single type of requests, which is the basic structure of microservice DAGs and to which other topologies can be transformed. For examples, a tree consists of multiple chains from the root to the leaf services, and similarly, fan-in and fan-out topologies can be decomposed to multiple chains from the source to the sink services. If a service is accessed multiple times by an upstream service, we consider the cumulative latency of all accesses as the latency of that service.

Theorem 1: Consider a chain of microservices S_1 to S_n, and their response time distributions t_1 to t_n, where $t_i(x_i)$ is the x_i^{th} percentile latency of microservice S_i, $x_i \in [0, 100]$. Similarly, we define t_e to be the end-to-end latency distribution, and $t_e(x_e)$ to be the x_e^{th} percentile end-to-end latency, $x_e \in [0, 100]$. Then,

$$t_e(x_e) \leq \sum_{i=1}^{n} t_i(x_i), \text{ if } 100 - x_e \geq \sum_{i=1}^{n} 100 - x_i \quad (1)$$

The theorem holds true regardless of the joint distribution of microservice latencies (i.e., if services are independent or correlated), and it denotes that *the sum of per-microservice latencies provides an upper bound for the end-to-end latency at an arbitrary percentile, as long the sum of residuals of per-microservice percentiles is no greater than the residual of the end-to-end percentile.* The proof can be found in the supplementary material.

Theorem 1 proposes a method to *guarantee the end-to-end latency SLA by examining the latency of individual microservices*. For example, in a chain consisting of two microservices S_1 and S_2, with the SLA defined as the end-to-end 99th percentile latency, Theorem 1 suggests that the actual end-to-end 99th percentile is less than the sum of x_1^{th} percentile latency of S_1 and x_2^{th} percentile latency of S_2, as long as $100 - x_1 + 100 - x_2 \leq 1$, and in other words, (x_1, x_2) can be $(99.1, 99.9)$, $(99.5, 99.5)$, $(99.7, 99.3)$, etc. Since all such combinations of x_i are upper bounds of the actual end-to-end latency, the end-to-end SLA must be satisfied as long as the corresponding sum of the per microservice latencies for one combination is less than the end-to-end SLA target. More generally, given the end-to-end latency SLA of x_e^{th} percentile latency needing to be less than T in a chain of length n, the end-to-end SLA is satisfied if

$$\exists [x_1 ... x_n] \ s.t. \ \sum_{i=1}^{n} t_i(x_i) \leq T \ \& \ 100 - x_e \geq \sum_{i=1}^{n} 100 - x_i \quad (2)$$

Mapping per-microservice latency to resources. To optimize resource allocation, the per-microservice latency distributions t_i need to be associated with resources, to derive a model that maps SLAs to resources. In addition, the model should be able to handle multiple classes or priorities of requests instead of a single class as in Theorem 1.

In cloud-native frameworks, such as Kubernetes [12], [4], [8], dynamic resource tuning is typically achieved by the changing the number of replicas, or container instances, each with a predefined resource configuration (CPU and memory). Therefore, we use *load per replica (LPR)* as the metric to relate resources to latency, where load is measured in requests per second (RPS). Considering a service S_i that handles c classes or priorities of requests (v_1 to v_c), the load per replica y_i can be represented as a vector $[a_i^1 ... a_i^c]$, where a_i^c is the load for request class v_c. If the load per replica vector y_i is used as the resource allocation threshold and the total load to S_i is $[A_i^1 ... A_i^c]$, the resources consumed by S_i can be calculated using Equation 3, where u_i is the resource consumption per replica.

$$r_i(y_i) = \max_{1 \leq j \leq c} \lceil \frac{A_i^j}{a_i^j} \rceil \cdot u_i \quad (3)$$

On the other hand, since the latency distribution of request v_j in S_i is a function of LPR y_i, the x_i^{th} percentile latency of v_j can be denoted as $t_i^j(y_i, x_i)$. Then the $t_i(x_i)$ items in Equation 2 can be replaced by $t_i^j(y_i, x_i)$, transforming the latency constraint to a resource allocation constraint. While $t_i^j(y_i, x_i)$ can be fitted with profiling data, the resource-latency function can be an arbitrary non-increasing function that is not necessarily convex, which makes it hard to use convex optimization models. Instead, we can discretize the variables and use the function in Mixed Integer Programming (MIP), which can be efficiently solved by modern optimization solvers using heuristics, such as branch-and-bound algorithm [39]. Specifically, we discretize the percentile variable x_i and LPR y_i, and represent the latency distributions under different LPRs as a matrix D_i^j, where each element of D_i^j is the latency that corresponds to a certain percentile for a given LPR. For example, assuming that S_i is profiled under m different LPRs $Y_i = [y_i^1 ... y_i^m]$ and the latency distribution is discretized into h different predefined percentiles $P = [p_1 ... p_h]$, D_i^j will be a $m \times h$ matrix where $D_i^j[\alpha, \beta]$ is the latency at percentile p_β under LPR y_i^α. As a result of the discretization, the LPR variable y_i can be represented by a one-hot vector δ_i of length m, indicating which LPR is chosen as the resource allocation threshold. Similarly, the percentile variable can be presented by one-hot vector γ_i^j of length h, indicating which percentile contributes to thesum of per-microservice latencies for request class or priority v_j. With the two one-hot decision variables, the latency of request class or priority v_j in microservice S_i can be expressed as $\delta_i^T D_i^j \gamma_i^j$, and the resource consumption can be expressed as $\delta_i^T R_i$, in which R_i is a 1-D vector corresponding to resource consumption under the profiled LPRs, computed with Equation 3.

Resource optimization model. Given that we can provide an upper bound on the end-to-end latency using the sum of per-microservice latencies and map per-microservice latencies to resource allocation thresholds, we can now design an optimization model that calculates the most efficient resource allocation threshold for each microservice, such that they all meet their respective per-microservice SLAs. Specifically, the inputs to the model include the load of the application, SLAs for different request classes and priorities, and per-microservice latency distributions under different LPR thresholds. The output of the model is the most efficient per-microservice LPR threshold that satisfies SLAs. With the described notations summarized in Table I, we derive the following solvable mixed-integer programming (MIP) model that yields optimal resource configurations given a set of end-to-end constraints: for each request class or priority v_j, the x_j^{th} percentile latency should be less than T_j.

	Description
δ_i	Resource (LPR) one-hot vector
γ_i^j	Latency percentile one-hot vector
R_i	Resource consumption under different LPRs
P	Discretized percentile values
D_i^j	Latency distribution matrix
T_j	End-to-end SLA target
x_j	End-to-end SLA target percentile
$\mathbb{1}$	1-D vector whose elements are all 1

TABLE I: Notations in the MIP formulation.

$$
\begin{array}{lll}
\text{minimize} & \sum_{i=1}^{n} \delta_i^T R_i, & \\
\text{subject to} & \sum_i \delta_i^T D_i^j \gamma_i^j \leq T_j, \forall j & (1) \\
 & \sum_i 100 - P^T \gamma_i^j \leq 100 - x_j, \forall j & (2) \\
 & \mathbb{1}^T \delta_i = 1, \forall i & (3) \\
 & \mathbb{1}^T \gamma_i^j = 1, \forall i, j & (4) \\
\text{variables} & \delta_i \ (0 \leq \delta_i \leq 1 \ \& \ \delta_i \in Z) & \\
 & \gamma_i^j \ (0 \leq \gamma_i^j \leq 1 \ \& \ \gamma_i^j \in Z) &
\end{array}
\quad MIP1
$$

The objective of MIP 1 is to minimize the total resource consumption, while meeting the end-to-end SLA. Constraint 1 specifies that for each request class or priority, the sum of per-microservice latencies must be smaller than the SLA target, and constraint 2 specifies that the sum of per-microservice latencies in the constraint 1 is an upper bound of the actual end-to-end latency. The rest of the constraints enforce the decision variables to be one-hot vectors. For each microservice, the LPR one-hot vector δ_i produced by MIP 1 corresponds to the most efficient resource allocation threshold among all profiled LPRs, which allows resource allocation of each microservice to be decided independently, by simply checking the load of the microservice.

Mitigating latency overestimation. The quality of the solution of MIP 1 is related to the tightness of the upper bound given by Theorem 1, as a loose upper bound well above the actual latency can lead to overprovisioning of resources. An intuitive way to tighten the upper bound is to record the ratio of the upper bound to the actual value and use that ratio to refine the SLA constraint in MIP 1. For example, if the overestimation ratio of request class or priority v_j is α_j and its expectation is $E(\alpha_j)$, constraint 1 in MIP 1 can be refined to $\sum_i \delta_i^T D_i^j \gamma_i^j \leq E(\alpha_j) T_j$. With a fixed resource allocation denoted by δ_i^*, for microservice $S_i, \forall i$, the upper bound on the latency of request class or priority v_j can be solved using MIP 2. The objective value is the tightest upper bound, because any percentile combination satisfying constraint 1 in MIP 2 establishes a upper bound on latency, and the objective is the smallest among all these upper bounds. Thus, the overestimation ratio α_j is the ratio of the objective over the actual latency, and $E(\alpha_j)$ is the average of α_j with different resource allocations.

$$
\begin{array}{lll}
\text{minimize} & \sum_i \delta_i^{*T} D_i^j \gamma_i^j, & \\
\text{subject to} & \sum_i 100 - P^T \gamma_i^j \leq 100 - x_j, \forall j & (1) \\
 & \mathbb{1}^T \gamma_i^j = 1, \forall i, j & (2) \\
\text{variables} & \gamma_i^j \ (0 \leq \gamma_i^j \leq 1 \ \& \ \gamma_i^j \in Z) &
\end{array}
\quad MIP2
$$

In practice, instead of using the expected overestimation ratio to tighten the bound, one can choose other metrics, such as the overestimation ratio at a high percentile, to tradeoff the potential resource efficiency and the risk of SLA violation. We leave this exploration to future work.

Discussion. In Ursa, we use this performance model to find the most efficient resource allocation given per-microservice latency SLA constraints, however, the model can be extended to other cases with minor modifications. For example, the model can handle end-to-end latency minimization under resource constraints, by replacing the objective of MIP 1 with the sum of per-microservice latencies, and using the total available resources as a constraint.

In addition, the model can handle SLAs defined in terms of request failure rates. The failure rate of an end-to-end request is no greater than the sum of request failure rates of the microservices it goes through, and each microservice's request failure rate is related to its resources, since insufficient resources will cause requests to time out and fail. The sum of per-microservice failure rates can then be used as a constraint to strengthen MIP 1. The model can also support dynamic request paths by adding recorded paths to the model during deployment. In the case of a microservice being accessed multiple times in a dynamic path, the model can be simplified by considering the total time spent in each microservice. We plan to investigate these potential use cases in future work.

Algorithm 1: LPR threshold profiling algorithm.

Input: Initial replica R, SLA violation threshold F_{sla}, backpressure-free threshold CPU_{bp}, profiling time T;
Output: Mapping from LPR to latency distributions;
Variable: Replica r, replica tuning step $step$, Load L, SLA violation frequency f_{sla}, CPU utilization cpu, latency distribution d_{lat};
Initialize $r \leftarrow R$, $map \leftarrow \{\}$;
while $r > 0$ **do**
  $wait(T)$;
  if $cpu \geq CPU_{bp}$ —— $f_{sla} \geq F_{sla}$ **then**
    terminate;
  else
    $map[\frac{L}{r}] = d_{lat}$, $r = r - step$;
  end
end
return map

Allocation space exploration. The task of allocation space exploration is to collect input data for the MIP model, including the potential resource allocation thresholds and the

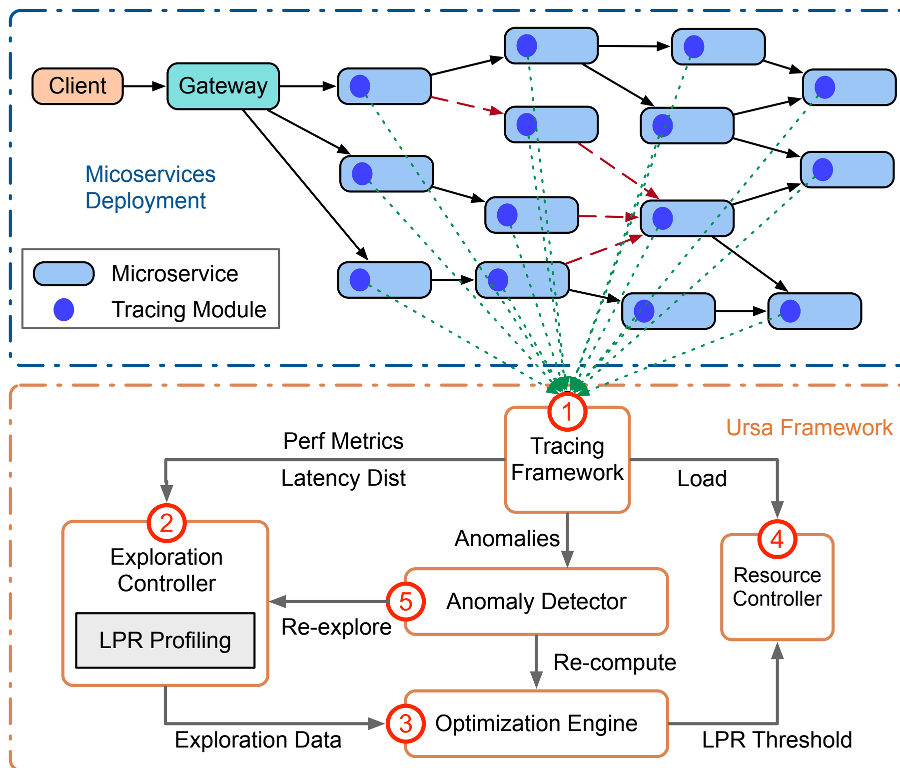

Fig. 5: System architecture of Ursa.

corresponding latency distributions for each microservice. Exploration should include the most efficient resource allocation threshold for each microservice and converge fast, by terminating swiftly if the latency exceeds the SLA. To achieve this goal, we explore each microservice individually with the LPR threshold profiling algorithm shown in Algorithm 1, by replaying the workload trace on the profiled microservice. During profiling, we gradually reduce the replicas of the profiled microservice to increase the load on each replica and record the corresponding latency distributions. Profiling is terminated when SLA violations are observed. Additionally, profiling is also terminated when the CPU utilization of the microservice exceeds the service's backpressure-free threshold to preserve the independence assumption used by the performance model.

V. Design and Implementation

We now present the design and implementation of ***Ursa***, a resource management framework based on the proposed performance model and allocation space exploration mechanism. Ursa is built on top of Kubernetes [12], a popular container orchestration framework adopted by major cloud providers [2], [4], [8], [1], and leverages Kubernetes's APIs to dynamically allocate resources by tuning the number of replicas per microservice. Ursa requires the user to provide the topology and the end-to-end SLAs of the microservice application, including request paths, percentiles, and target latencies.

Ursa aims to make allocation decisions fast and scalable. Ursa simplifies resource management decisions to threshold-based scaling by implementing the performance model from Section IV. In addition, Ursa collects input data for the performance model via the exploration process used to identify the relation between resources and per-microservice SLAs. The components of Ursa are shown in Figure 5, and the functionality of each component is described below.

1. The **tracing framework** is implemented with Prometheus [15], a time-series database for metrics monitoring. It collects the CPU and memory usage data, as well as the request counts and latency distributions of each service which are used to calculated statistics required by the MIP model.

2. The **exploration controller** implements the allocation space exploration mechanism. It first determines the backpressure free CPU utilization thresholds for each RPC-connected microservice individually (Section III). Then, it explores feasible resource allocation thresholds of each microservice individually using Algorithm 1. The complete exploration of all microservices is required for a new application and must be completed before Ursa begins to manage the application. In the case of continuous development of individual microservices, only the updated microservices require re-exploration.

3. The **optimization engine** then determines the resource allocation threshold for each microservice using the performance model in Section IV, using exploration data and user load information collected by the tracing framework. The optimization engine is implemented with Gurobi [10]. During the initial deployment phase, the optimization engine is invoked with current user load information to generate the optimal scaling thresholds of each microservice. Subsequently, the optimization engine may need to be invoked again if the

mix of user requests changes significantly or when the business logic of microservices is updated.

4. Using the load per replica (LPR) threshold calculated by the optimization engine, the **resource controller** dynamically adjusts the number of replicas as the load changes, ensuring that for any class or priority of request, the average load on each replica does not exceed the threshold. Specifically, the resource controller determines whether the average load in one replica exceeds the threshold using Welch's t-test [57] to accommodate the noise of load fluctuations. Specifically, the resource controller compares the actual load of the microservice, with the recorded load used as the scaling threshold, and considers the threshold to be exceeded if the t-test rejects the hypothesis that the mean of the actual load is less than the mean of the recorded load.

5. During deployment, the **anomaly detector** periodically checks for anomalies in load and latency, and triggers recalculation of resource allocation thresholds or re-exploration, if necessary. Load anomalies refer to drastic changes in the ratio of different classes or priorities of requests that may lead to resource over-provisioning, in which case resource allocation thresholds are recalculated to improve resource efficiency. The anomaly detector identifies changes in request ratios by monitoring the *request ratio deviation* of each microservice, which measures the difference between the load of the microservice and the load per replica threshold for scheduling. The metric is denoted by $\max_i \frac{l_i}{t_i} \frac{\sum_i t_i}{\sum_i l_i}$, where l_i and t_i are the total load and per-replica load threshold for the ith request class or priority. When the request ratio deviation exceeds a user-defined threshold, the anomaly detector asks the optimization engine to recalculate the thresholds and update the resource controllers. If the re-calculated thresholds still fail to mitigate the request ratio deviation, indicating that the load pattern is not covered by previous exploration, the anomaly detector asks the exploration controller to re-explore the affected microservice.

On the other hand, latency anomalies refer to SLA violations, which indicate that the latency distribution recorded during exploration needs to be updated. Similar to load anomalies, users can specify an end-to-end SLA violation threshold that triggers the re-exploration process if the SLA violation exceeds the threshold during deployment.

VI. Microservice Benchmarks

Conventional microservice benchmarks [28], [50], [62] have several limitations. First, conventional benchmarks use RPCs as the only method for inter-service communication, whereas MQs are becoming increasingly common in practice [40]. Second, the business logic of conventional benchmarks involves only lightweight text processing, whereas a modern microservice handles different user request classes performing tasks, such as image processing and ML workloads [49], [42], [47], and even different request priorities, making resource management more challenging. To address these limitations, we implement three benchmark applications using Dapr [5], a popular microservice framework developed and used by major cloud providers, as described below. For all the applications, we implement the business logic in Golang and Python, and use gRPC [9] for RPCs, Redis streams [17] for message queues and Redis [16] for data stores. We incorporate MQ into our approach because it is an increasingly popular communication framework for microservices.

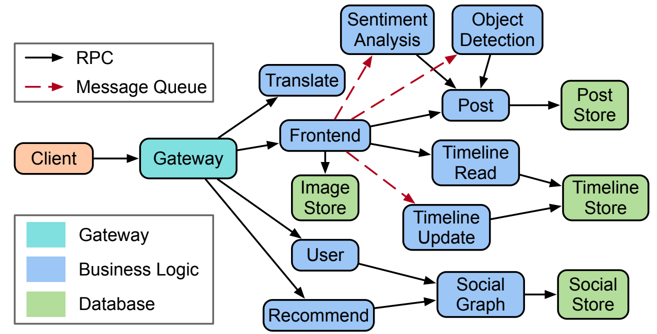

Fig. 6: Social network microservice topology.

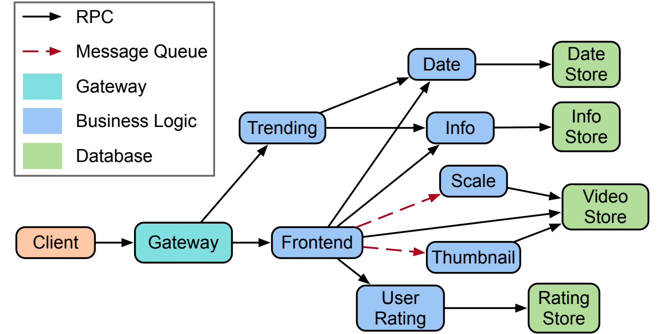

Fig. 7: Media service topology.

In our benchmarks, the interactive functionalities demanding immediate response, such as reading timelines from the social network, are implemented with RPC. Conversely, functionalities that do not require immediate response, such as labeling objects in user-uploaded images, are implemented with message queues.

Social network. The social network application is a re-implementation of the DeathStarBench [28] application. In addition to the original features including uploading text posts and reading timelines, the re-implemented version includes several new features, including uploading images, sentiment analysis of texts, and object detection of images. Sentiment analysis and object detection are implemented with machine learning models from Hugging Face [11], and are connected to other services via MQs.

Media service. The media service is also a re-implementation of the corresponding DeathStarBench application. In addition to the original features including reviewing and rating videos, the re-implemented version additionally allows users to upload and download actual videos, and includes video-processing tasks, such as transcoding to different resolutions and generating thumbnails via FFmpeg [7]. The video transcoding and

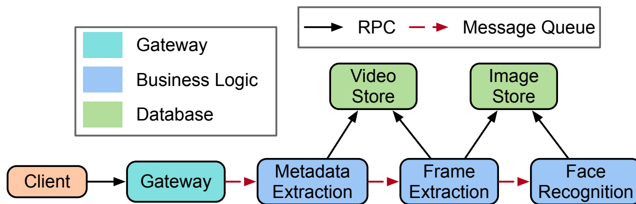

Fig. 8: Video processing pipeline.

thumbnail services are connected to other services via MQs.

Video processing pipeline. Video processing pipeline consists of three stages: The first stage extracts video metadata, the second stage takes snapshots from the video at fixed intervals, and the third stage performs face recognition on the video snapshots. The first two stages use FFmpeg, the third stage uses OpenCV [14], and stages are connected with MQs. The application handles two request priorities. High-priority requests are always processed immediately when worker threads are available, while low-priority requests are processed only when there is no high-priority request waiting in the queue.

Previous work typically handles a single SLA and only manages synchronous requests. For example, Sinan [60] handles a single SLA of 500ms for the 99th percentile latency of upload-post, read-timeline, and update-timeline in social network. However, different request classes have diverse latencies. For example, in social network, it takes tens of milliseconds to upload a post, hundreds of milliseconds to update timelines, and a few seconds to perform object detection. To reflect these latency ranges, we assign an SLA per request class and priority. We stress test the applications with high user loads and use the latency before saturation as the SLA. The SLAs of the social network, media service, and video processing pipeline are listed in Table II, III, IV, respectively. The SLAs are mostly defined as the 99th percentile, except for the low-priority requests in the video processing pipeline, which is defined as the 50th percentile latency.

Request type	99th latency (ms)
upload-post/comment	75
read-timeline	250
update-timeline	500
upload-image	200
download-image	75
sentiment-analysis	500
object-detect	10000

TABLE II: SLA requirements of the social network.

VII. Evaluation

We aim to answer the following questions:

1) What is the overhead of Ursa's exploration process? (Section VII-C)

Request type	99th latency (ms)
upload-video	2000
download-video	1500
get-info	250
rate-video	400
transcode-video	40000
generate-thumbnail	2000

TABLE III: SLA requirements of the media service.

Request type	Percentile	Latency (ms)
high-priority	99th	20000
low-priority	50th	4000

TABLE IV: SLA requirements of the video processing pipeline.

2) How accurate is Ursa's performance model in capturing end-to-end latency (Section VII-D)?
3) How effective is Ursa in reducing resource usage and maintaining SLAs? (Section VII-E)
4) What is the latency required for Ursa to make resource allocation decisions? (Section VII-F)
5) Is Ursa able to adapt to business logic changes of microservices? (Section VII-G)

A. Experimental Setup

We use the benchmarks in Section VI and use Locust [13] to generate input load following a Poisson arrival process. The applications are deployed on a local Kubernetes cluster consisting of 8 machines, each with 40-88 CPUs and 126-188 GB of memory each, with a NIC bandwidth of 10 Gbps. To reduce interference between containers colocated on the same server, we set the CPU management policy of Kubernetes to the static policy [37], which allows each container to access exclusive CPUs, as long as it is configured with an integer number of CPUs. The CPU configuration of each microservice's container is determined by monitoring the CPU usage of the container at low RPS and rounding it to the nearest integer, and similarly, the memory configuration is set to the maximum profiled memory usage to avoid OOM errors. During online deployment, we adjust the resource allocation for each microservice by adjusting the number of replicas.

B. Competing Approaches

We compare Ursa to the following systems.

Sinan. Sinan [60] is a model-based ML-driven microservice management framework. It uses a CNN and boosted trees model, to predict the end-to-end latency of a microservice topology given a certain resource allocation, as well as the probability that a resource allocation will lead to an SLA violation later into the future, taking into account the system's inertia in building up queues. Sinan is implemented as a centralized scheduler that periodically queries the model with different resource allocations, and chooses the one using the

least amount of resources, while meeting the end-to-end SLA. The training data for the models are collected with a process designed to explore unseen resource allocations and to keep the ratio of violating to meeting SLAs at $1:1$, so that the trained models are not biased towards either predicting SLA violation or SLA satisfaction. We train Sinan with 10,000 samples, per the paper's recommendation.

Firm. Firm [46] is a model-free, ML-driven framework for microservice resource management. Unlike Sinan that trains models to predict latency, Firm assigns a reinforcement learning agent to each service that directly adjusts the resource allocation for the service, given its resource usage and end-to-end SLA status. The reward for each agent is designed to be the weighted sum of the reduced resource usage and the SLA violation status after applying the resource allocation decision. The agents are trained by injecting performance anomalies during online deployment. Similarly to Sinan, we also use 10,000 training samples for Firm to allow accuracy to converge.

For both Sinan and Firm, we modified the systems to handle asynchronous events and queues, since the original systems were designed exclusively for RPCs.

Autoscaling. Autoscaling [18] is a widely adopted resource management method. The autoscaling controller relies on manually configured resource utilization thresholds based on expert knowledge to dynamically adjust resource allocation. In our evaluation, we experiment with two configurations for autoscaling. The first configuration uses the default setting of AWS step scaling [18], which increases resources when CPU utilization exceeds 60%, and reduces resources when the CPU utilization is below 30%. This configuration is optimistic in terms of resource usage, but comes at the expense of SLA maintenance. The second configuration is manually tuned to preserve the SLAs of tested applications, but uses more resources. In the rest of the paper, we name the first configuration Auto-a, and the second configuration Auto-b.

C. *Exploration Overhead*

We now compare the exploration overheads of Ursa, Sinan and Firm. During exploration, the combination of user requests for each application is the same across the three approaches. Specifically, for the social network application, the ratios of post, comment, download-image and read-timeline are approximately 1:75:15:25, adopted from [60], [38], [30]. For the media service application, the ratios of upload-video, get-info, download-video, and rate-video are approximately 1:100:25:25. For the video processing pipeline, we experiment with four different ratios of high and low priority requests, including 5:95, 25:75, 50:50, and 75:25. Across all approaches, the sampling frequency is set to once per minute.

We run Ursa's exploration process for each microservice individually, as described in Algorithm 1. Since CPU and memory are the two major resources on cloud platforms, where CPU directly affects latency and memory is configured to avoid OOM error, during Ursa's exploration we configure the initial replica of each microservice by providing it with adequate CPUs to keep the microservice's latency low. Each microservice is profiled using Algorithm 1, and in each iteration we reduce the number of replicas by 1 and collect 10 samples, until the frequency of SLA violations exceeds 10%, or the CPU utilization exceeds the backpressure-free threshold. For Sinan and Firm, we run their data collection algorithm and online training process separately and collect 10k samples for each application, matching the order of magnitude in Sinan for DeathStarBench [60].

Table V summarizes the number of samples collected during exploration and the required exploration time. For Ursa, the exploration time is decided by the longest time required to profile a single microservice, as each microservice can be profiled individually, and the number of samples is the sum of samples collected for all microservices. Compared to the ML-driven approaches, Ursa reduces the required sample size by a factor of 16.7 up to 25.6, and the exploration time by a factor of 128.2 up to 208.4. The high online exploration overheads of ML-driven approaches result from the nature of deep neural networks, which require a large amount of data to generalize, due to their large parameter space. In contrast, Ursa's analytical model inherently contains fewer parameters than DNNs. The model calculates end-to-end latency as the sum of per-microservice latencies, and foresees end-to-end SLA violations when the latency of individual microservices increases rapidly. Notably, despite the small required sample size, Ursa still maintains SLA and achieves high efficiency during deployment, as shown in Section VII-E.

App	Systems	Samples	Time(h)
Social	Ursa	440	1.2
	Sinan/Firm	10000	166.7
Media	Ursa	390	0.8
	Sinan/Firm	10000	166.7
Video	Ursa	600	0.8
	Sinan/Firm	10000	166.7

TABLE V: Exploration overheads with Ursa compared to two ML-driven frameworks, Sinan and Firm.

D. *Model Accuracy*

The performance model (Section IV) estimates the end-to-end latency by multiplying the latency's upper bound with the expected overestimation rate. To evaluate the estimation accuracy, we record the per-microservice and end-to-end latency distributions every 5 min for a total of 150 min during online exploration with dynamically changing resource allocations, and calculate the estimated latency for each type of request.

Figure 9 shows the measured and estimated latency of four representative request types in Social Network, including post, update-timeline, object-detection, and sentiment-analysis. The blue line indicates the measured 99th percentile latency and the red line indicates the estimated 99th percentile latency. For each class of requests, the estimated latency closely follows

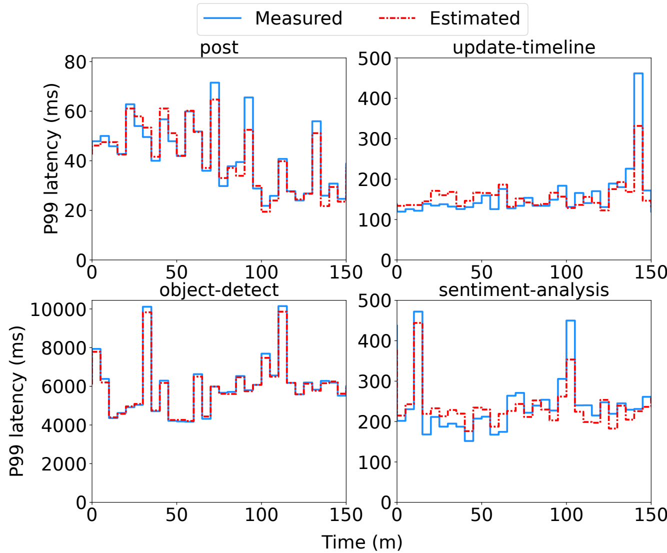

Fig. 9: Social network–Estimated vs. measured latency.

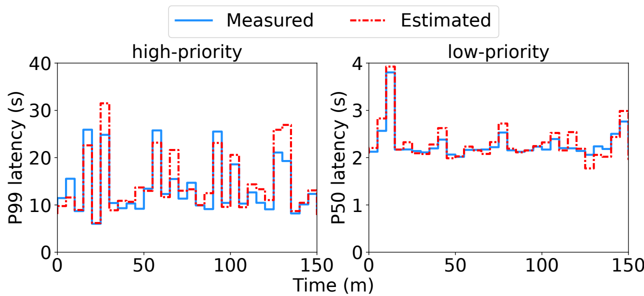

Fig. 10: Estimated vs. measured latency for the video processing pipeline.

the measured latency, with the average ratio of estimated to measured latency ranging from 0.97 to 1.05. Additionally, Figure 10 shows the measured and estimated latency of the video processing pipeline which includes two request priorities, with SLAs defined at the 50th and 99th percentiles, for low and high priority requests, respectively. For both priorities, the estimated latency is close to the measured latency, with the average ratio of estimated to measured latency being 0.96 and 1.00 for low and high priority requests, respectively.

E. Performance Comparison

We now compare resource usage and SLA violations during deployment between Ursa and prior work, with Ursa and ML-driven systems using exploration data from Section VII-C. In addition to the three previous applications, we also show the results for the vanilla social network which implements the same functionality as the original benchmark, by disabling the newly added ML services, to highlight the challenges that stem from resource need heterogeneity across microservices.

For each application, we experiment with three user loads; *constant load*, *dynamic load*, and *skewed load*. *Constant load* refers to Poisson arrival processes with constant RPS. *Dynamic load* has time-varying RPS, including diurnal patterns where the RPS first gradually increases and then gradually decreases, and burst patterns, where the RPS increases sharply by 50% to 125%. The ratio of different types of requests for constant and dynamic loads is the same as in online exploration. In *skewed load*, the ratio of request types differs from that in the online exploration. For social network and media service, we experiment with two other request combinations, the first doubling the frequency of update requests, and the second halving the frequency of update requests. For the video processing pipeline, the ratios of high-priority to low-priority requests include 40:60 and 60:40, which do not exist in online exploration. For Ursa specifically, the skewed load stresses the case where the request mix changes, and the LPR thresholds needs recalculated using available exploration data that do not include the current request mix. For each type of load, Ursa calculates the optimal load-per-replica thresholds once, at the beginning of the experiment.

Figure 11 shows the SLA violation rate, and Figure 12 shows the average CPU allocation, or the total amount of CPU resources allocated to the microservice. Compared to ML-driven systems, Ursa significantly reduces SLA violation rates, achieving 0.1% to 8.5% SLA violation rates under constant and dynamic loads, and 0.5% to 2.0% SLA violation rates under skewed load, whereas ML-driven systems incur 9.1% to 29.2% SLA violation rates under constant and dynamic loads, and 14.2% to 51.9% SLA violation rates under skewed load. ML-driven systems cause higher SLA violation rates for the new social network than for vanilla social network, because the latency of ML microservices is less stable and more challenging for resource management, compared to lightweight text processing. In terms of resources, Ursa reduces CPU allocation by 2.3% to 86.2% for constant and dynamic loads. For skewed loads, Ursa uses an average of 8.2% more CPUs, but the ML-driven systems result in SLA violation rates significantly higher than Ursa. As for autoscaling, Auto-a, which uses the default setting of AWS step scaling, uses the least resources but results in SLA violation rates of over 40%. Auto-b, the manually tuned configuration, maintains SLAs most of the time, with SLA violation rates only 0.9% to 6.1% higher than Ursa. Despite the low SLA violation rates, Auto-b uses significantly more resources than Ursa, allocating 43.9% to 148.0% more CPUs under constant and dynamic loads, and 13.6% to 57.0% more CPUs allocation under skewed loads.

Ursa may use more resources under skewed loads because it prioritizes maintaining SLAs and makes conservative decisions with the available exploration data. As a conceptual example, assume a microservice handles two classes of requests, and its total load is $(4, 6)$, where each element of the vector is the load of one class of requests. If the microservice's exploration data only includes one feasible LPR threshold $(3, 2)$, Ursa will provision 3 replicas for the microservice to ensure that the load of any request class is below the threshold, while in reality the actual per-replica load will be $(1.3, 2)$, which is below the $(3, 2)$ threshold. Figure 13 shows the load and CPU allocation for four representative microservices in the social

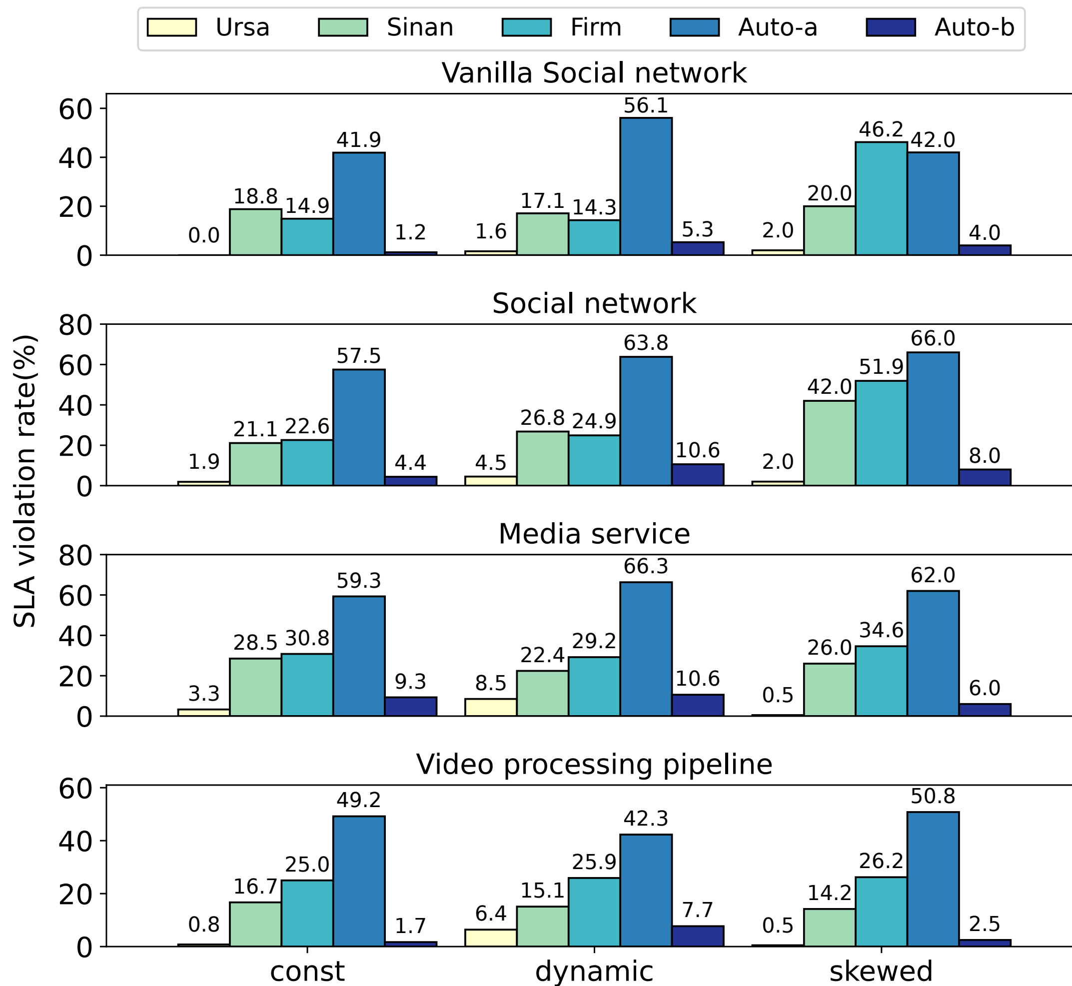

Fig. 11: SLA violation rate across load patterns for the original (Vanilla) and reimplemented Social Network application, the Media service, and the Video Processing pipeline, with Ursa, Sinan, Firm, and the empirical Automscaling system.

network under a diurnal load when managed with Ursa, where the left Y-axis represents the RPS of load and the right Y-axis represents the CPU allocation. For each microservice, Ursa is able to scale out and scale in promptly as the load increases and decreases.

Ursa outperforms ML-driven systems with much lower exploration overheads, because Ursa's analytical model accurately decomposes the end-to-end latency to per-microservice latencies, which can be mapped directly to per-microservice resource allocation. However, the ML-driven techniques need to learn the relation between resource allocation and SLA from scratch in a much larger parameter space, requiring more data and leading to lower accuracy. Specifically, Sinan's SLA violation predictor can only achieve 80% to 85% accuracy due to the presence of multiple request classes with different SLAs, resulting in more SLA violations and higher resource usage. On the other hand, in addition to the issue of large parameter space, Firm does not always prioritize preserving SLAs because its agent's reward function is a weighted sum of the SLA violation rate and the resource utilization, which makes Firm prioritize resource savings over SLA if the savings are significant, resulting in more violations.

Discussion. We previously compared the resource consumption of the evaluated systems. Another metric of interest in cloud native environments is throughput per dollar, i.e., the user request throughput achievable with the same cost budget. Since all systems are evaluated under the same workload patterns, the improvement in Ursa's throughput per dollar is the inverse of its resource savings. For example, compared to ML-driven systems, Ursa reduces CPU allocation by 2.3% to 86.2% for constant and dynamic loads, which represents an improvement in throughput per dollar of $1.02\times$ to $7.24\times$. Ursa's improvement is even more pronounced when considering goodput per dollar, i.e., the user request throughput that meets SLA under the same cost budget, since Ursa significantly reduces SLA violations compared to other systems.

F. Control Plane Latency

The latency of resource allocation decisions is directly related to responsiveness of resource management systems.

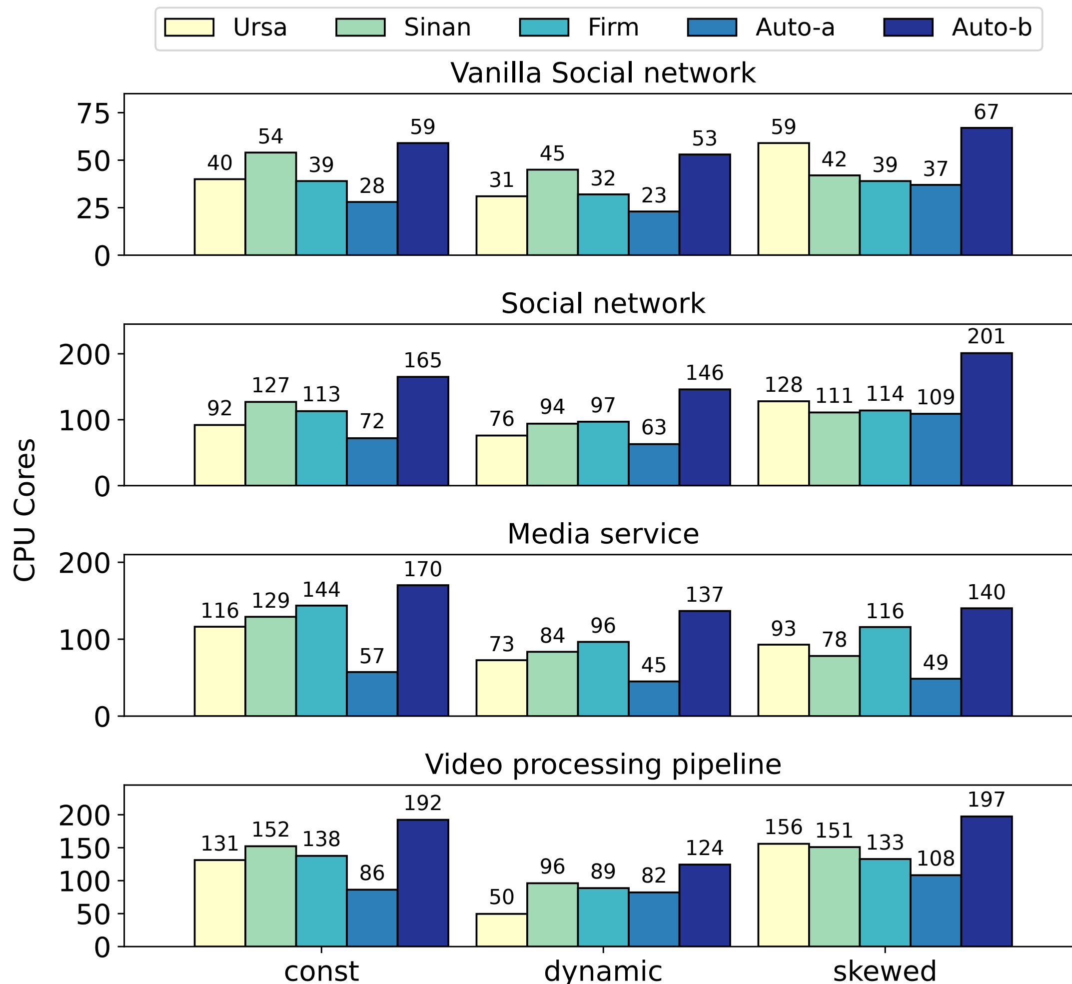

Fig. 12: Average CPU allocation across load patterns for the original (Vanilla) and reimplemented Social Network application, the Media service, and the Video Processing pipeline, with Ursa, Sinan, Firm, and the empirical Automscaling system.

Resource allocation decisions are fast in Ursa because the critical path only includes the resource controller, which calculates the number of replicas, based on the load-per-replica thresholds, while inference of ML requires over thousands of floating-point operations. There are also situations where the model needs to be updated to account for changes in service business logic or load combinations. In such cases, Ursa needs to recompute the optimization models, and ML-driven approaches also need to be retrained. Table VI shows the average control plane latency (in milliseconds) across the different approaches, in the case of deployment and model update. In the comparison, the control planes are always allocated 4 CPUs. Autoscaling is undoubtedly the fastest, as it involves only a single threshold check. In terms of deployment, Ursa is on average $691.6\times$ faster than Sinan with its centralized ML model, and $43.4\times$ faster than Firm, which uses per-service RL agents. In terms of model updates, Sinan's retraining time is linear with the size of the dataset, and takes minutes even on a dedicated GPU. Firm can adapt to load changes gradually by updating the RL agent online, but is still slower than Ursa by $4.4\times$ even for a single iteration. The RL agent may require thousands of iterations to update its weights and fully learn new resource patterns, whereas Ursa only needs to solve the optimization problem once to fully adapt to the changes.

	Ursa	**Sinan**	**Firm**	**Autoscaling**
Deploy	0.5	345.8	21.7	0.1
Update	271.7	N/A	1.2×10^3	0.1

TABLE VI: Average control plane latency in milliseconds across Ursa, Sinan, Firm, and Autoscaling for the initial application deployment, as well as when retraining of the model is required, due to a change in application logic.

G. Adapting to Service Changes

A basic premise of microservices is that they enable frequent logic updates without the high overhead of redeploying the entire service mesh. We now conduct a case study to

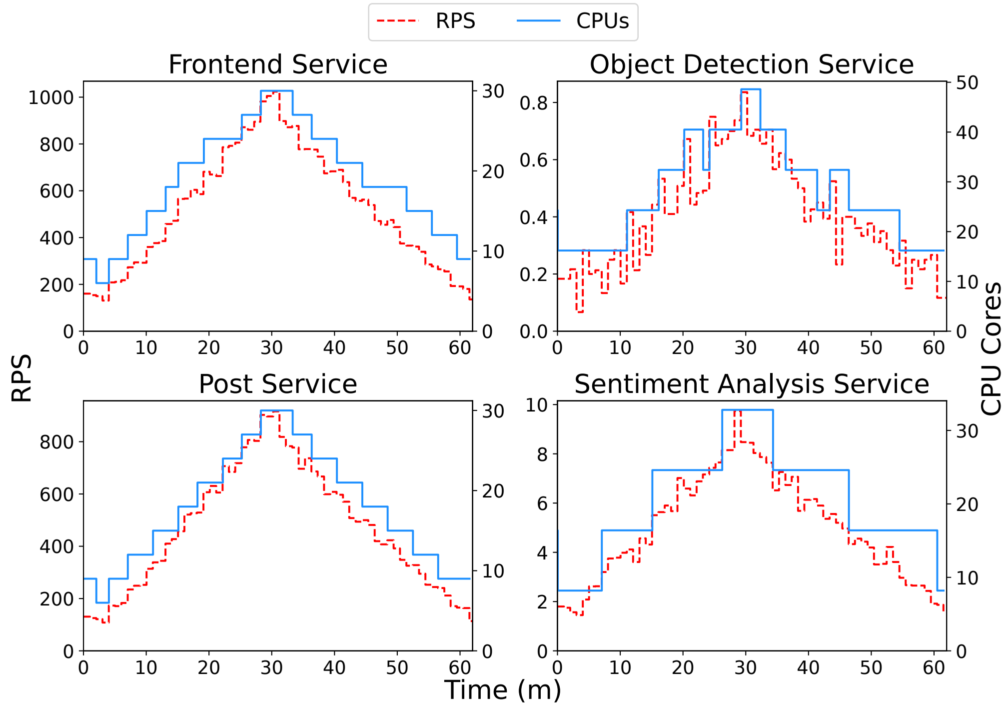

Fig. 13: Ursa's CPU allocation under a diurnal load. We show the RPS and CPU allocations for individual, representative microservices.

demonstrate Ursa's ability to adapt to such business logic updates. Specifically, we modify the object-detection service in the social network application, and change the model from DETR [21] which combines Transformer [55] and Resnet [32] for object detection, to the more lightweight Mobilenet [33]. The exploration controller performs a partial online exploration to profile only the modified object-detect service. It collects a total of 75 samples in 1.25 hours, during which 4 SLA violations are triggered, resulting in an SLA violation rate of 5.3%. Then the optimization engine recalculates the LPR threshold of each service. We deploy the modified social network application under various RPS, and Figure 14 shows the distribution of the 99th percentile latency of the end-to-end object-detect requests for the original and the updated Social network service mesh. Object-detect requests go through the frontend service, image store, post service, and the object-detect service. The red line represents the SLA and the blue line represents the cumulative distribution function, with SLA violation rates of 0.62% and 0.50% for the original and updated microservice, respectively, showing Ursa's ability to quickly adjust to changes in the application logic.

H. Summary

Compared to ML-driven approaches, Ursa reduces the required exploration time by more than $128\times$, making it practical to track frequent changes in service business logic and user loads. Ursa also achieves better performance, maintaining low SLA violation rates of 0.1% to 8.5% during deployment, 9.0% to 49.9% lower than ML-driven approaches, and reducing resource allocation by up to 86.2%. Ursa also outperforms traditional autoscaling, both the default setting of AWS step scaling, and a conservative, manually tuned configuration. In addition, Ursa's control plane is faster than ML, enabling faster and more scalable management decisions. Finally, we demonstrate that Ursa is able to adapt to service changes and maintain SLAs while incurring low exploration overheads.

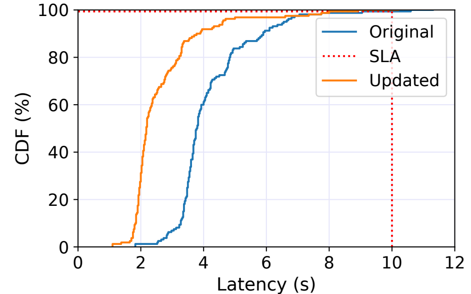

Fig. 14: 99th latency distribution of object-detect.

VIII. Conclusion

We present Ursa, a lightweight resource management framework for microservices. Ursa uses an analytical model to decompose the end-to-end SLA into per-microservice SLAs, and maps them to resource allocations. During exploration, Ursa profiles each microservice individually and swiftly stops exploration in the case of SLA violations to shorten the exploration process. Using benchmarks implemented with

popular microservice frameworks, we demonstrate that Ursa outperforms ML-driven approaches and traditional autoscaling in both SLA maintenance and resource efficiency, with significantly lower exploration overheads.

Acknowledgements

We sincerely thank Ricardo Bianchini, Rodrigo Fonseca, Íñigo Goiri, Mahesh Ketkar, Ramesh Illikkal, Nikita Lazarev, Mingyu Liang, Varun Gohil, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was partly done during an internship at Microsoft Research and was in part supported by an NSF CAREER Award CCF-1846046, an Intel Research Award, an Intel Faculty Rising Star Award, a Sloan Research Fellowship, a Microsoft Research Fellowship, a Facebook Research Faculty Award, and a Google Research Award.

References

[1] "Alibaba Cloud Container Service for Kubernetes," https://www.alibabacloud.com/product/kubernetes.
[2] "Amazon Elastic Kubernetes Service," https://aws.amazon.com/eks/.
[3] "Apache Kafka," https://kafka.apache.org/.
[4] "Azure Kubernetes Service," https://azure.microsoft.com/en-us/products/kubernetes-service/#overview.
[5] "Dapr: APIs for building portable and reliable microservices," https://dapr.io/.
[6] "Decomposing twitter: Adventures in service-oriented architecture," https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.
[7] "FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video," https://ffmpeg.org/.
[8] "Google Kubernetes Engine," https://cloud.google.com/kubernetes-engine.
[9] "gRPC: A high performance, open source universal RPC framework," https://grpc.io/.
[10] "Gurobi Optimization," https://www.gurobi.com/.
[11] "Hugging Face," https://huggingface.co/.
[12] "Kubernetes: Production-Grade Container Orchestration," https://kubernetes.io/.
[13] "Locust: A modern load testing framework," https://locust.io/.
[14] "OpenCV Face Recognition," https://opencv.org/.
[15] "Prometheus," https://prometheus.io/.
[16] "Redis," https://redis.io/.
[17] "Redis streams tutorial," https://redis.io/docs/data-types/streams-tutorial/.
[18] "Step and simple scaling policies for amazon ec2 auto scaling," https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html.
[19] P. Ambati, Í. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, and R. Bianchini, "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms," in *OSDI*, 2020.
[20] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *OSDI*, 2014.
[21] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *European conference on computer vision*. Springer, 2020, pp. 213–229.
[22] K.-H. Chow, U. Deshpande, S. Seshadri, and L. Liu, "Deeprest: deep resource estimation for interactive microservices," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 181–198.
[23] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167.
[24] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *USENIX ATC*, 2015.
[25] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.
[26] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed Job Latency in Data Parallel Clusters," in *EuroSys*, 2012.
[27] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ml-driven performance debugging in microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 135–151.
[28] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, Y. He, and C. Delimitrou, "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*. ACM, 2019, pp. 3–18.
[29] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *NSDI*, 2011.
[30] K. Gligorić, A. Anderson, and R. West, "How constraints affect content: The case of twitter's switch from 140 to 280 characters," in *Twelfth International AAAI Conference on Web and Social Media*, 2018.
[31] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *OSDI*, 2016.
[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[33] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
[34] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *SOSP*, 2009.
[35] 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 *SOSP*, 2016.
[36] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *USENIX ATC*, 2015.
[37] Kubernetes, "Kubernetes cpu management policy," https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/, 2021.
[38] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. AcM, 2010, pp. 591–600.
[39] A. H. Land and A. G. Doig, "An automatic method for solving discrete programming problems," in *50 Years of Integer Programming 1958-2008*. Springer, 2010, pp. 105–132.
[40] 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.
[41] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The power of prediction: microservice auto scaling via workload learning," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 355–369.
[42] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "{ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 303–320.
[43] D. Narayanan, F. Kazhamiaka, F. Abuzaid, P. Kraft, A. Agrawal, S. Kandula, S. Boyd, and M. Zaharia, "Solving large-scale granular resource allocation problems efficiently with pop," in *Proceedings of*

the ACM SIGOPS 28th Symposium on Operating Systems Principles, 2021, pp. 521–537.
[44] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, “Sparrow: Distributed, Low Latency Scheduling,” in *SOSP*, 2013.
[45] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger, “3sigma: distribution-based cluster scheduling for runtime uncertainty,” in *EuroSys*, 2018.
[46] 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,” *arXiv preprint arXiv:2008.08509*, 2020.
[47] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, “Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications,” *arXiv preprint arXiv:2104.13869*, 2021.
[48] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, “Autopilot: workload autoscaling at google,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.
[49] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 205–218.
[50] A. Sriraman and T. F. Wenisch, “μ suite: a benchmark suite for microservices,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 1–12.
[51] A. Sriraman and T. F. Wenisch, “μtune: Auto-tuned threading for OLDI microservices,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 177–194.
[52] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu, “Distributed resource management across process boundaries,” in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 611–623.
[53] Tony Mauro, “Adopting microservices at Netflix: Lessons for architectural design,” https://www.nginx.com/blog/microservicesat-netflix-architectural-best-practices/.
[54] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, “TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters,” in *EuroSys*, 2016.
[55] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
[56] Z. Wang, S. Zhu, J. Li, W. Jiang, K. Ramakrishnan, Y. Zheng, M. Yan, X. Zhang, and A. X. Liu, “Deepscaling: microservices autoscaling for stable cpu utilization in large scale cloud systems,” in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 16–30.
[57] B. L. Welch, “The generalization of ‘student’s’problem when several different population varlances are involved,” *Biometrika*, vol. 34, no. 1-2, pp. 28–35, 1947.
[58] M. Welsh, D. Culler, and E. Brewer, “Seda: An architecture for well-conditioned, scalable internet services,” *ACM SIGOPS operating systems review*, vol. 35, no. 5, pp. 230–243, 2001.
[59] 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*, ser. ISCA ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 133–146.
[60] 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*, 2021, pp. 167–181.
[61] 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*. ACM, 2018, pp. 149–161.
[62] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,” *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.