1. Motivation

In recent years, cloud applications have progressively shifted from monolithic services to graphs with hundreds of single-purpose and loosely-coupled microservices [1, 4, 5, 12, 23, 24]. This shift is becoming increasingly pervasive, with large cloud providers, such as Amazon, Twitter, Netflix, and eBay having already adopted this application model [1, 4, 5]. Despite advantages such as flexible development and rapid iteration, microservices also introduce new challenges, especially in resource management, since the complex dependencies between microservices exacerbate queueing effects, and introduce cascading QoS violations that are difficult to identify and correct in a timely manner [12, 28]. Given the increasing number of cloud services now designed as microservices, addressing their resource management challenges is a pressing need.

2. Limitations of the State of the Art

Current cluster managers are mainly designed for monolithic applications, or applications consisting of a few pipelined tiers, and are not expressive enough to capture the complexity of microservices [13, 14, 15, 16, 17, 18, 21, 22, 25]. Traditionally-employed empirical approaches based on resource utilization thresholds, like autoscaling [3], or approaches based on queuing analysis [27] result in significant QoS violations and resource inefficiency when applied to microservices.

On the other hand, machine learning-driven (ML) approaches have been shown to be effective at solving resource management problems for large-scale systems in previous work [7, 9, 20]. However, these methods are designed specifically for monolithic services or VMs operating independently from each other, excluding interference effects from colocation [8, 9, 26], and hence cannot be directly applied to graph of microservices. More recently, prior work examined the potential of ML-driven techniques for performance debugging in microservices, such as with Seer [11], however, performance debugging is an invasive process that should ideally only be relied on to correct resource allocations during events of suboptimal performance, instead of being tasked with adjusting the resources of all active microservices in a cluster at all times. Appropriately managing resources to begin with lightens the burden on the performance debugging system, allowing it instead to focus on quickly resolving unexpected events introducing poor performance.

3. Key Insights

Sinan provides the following four insights regarding resource management in microservices.

1. Dependencies among tiers: Resource management in microservices is additionally complicated by the fact that dependent microservices are not perfect pipelines, and can introduce backpressure effects that are hard to detect and prevent [12, 28]. These dependencies can be further exacerbated by the specific RPC and data store API implementation. Therefore, the manager must have a global view of the microservice graph, and assess the impact of dependencies on end-to-end QoS.

2. System complexity and large action space: Microservices change frequently, therefore resource management decisions need to happen online. This means that the resource manager must traverse a space that includes all possible resource allocations per microservice in a practical manner. Unfortunately prior empirical approaches that rely on either resource utilization or queue length monitoring cannot be directly employed in microservices with tens of tiers and complex dependencies for two reasons. First, microservice dependencies mean that the resource usage of different tiers is codependent, so examining fluctuations in individual tiers can attribute poor performance to the wrong tier. Second, although queue lengths are accurate indicators of system state for microservices, obtaining exact queue lengths is hard, as queues exist across the system stack from the NIC and OS to the network stack and application level. Accurately tracking queue lengths requires application changes and heavy instrumentation, which can negatively impact performance or is not possible in public clouds. This is also the case when applications include third-party software whose source code cannot be instrumented. Alternatively, expecting the user to express each tier’s resource sensitivity is problematic, as users already have a hard time reserving resources for simple single-tier services, leading to well-documented underutilization [8, 9, 19], and the impact of dependencies between microservices is especially difficult to assess, even for expert developers.

3. Delayed queueing effect: Multi-tier microservices conform to queueing network principles. This means that, when a queueing system’s processing throughput falls below the offered load, queues will start accumulating. Despite this, the service’s QoS target is not immediately violated, as queue accumulation requires time. The converse is also true; by the time QoS is violated, the built-up queues take a long time to drain, even if resources are upscaled immediately upon detecting the violation. Multi-tier microservices are complex queueing systems with queues both across and within microservices. This delayed queueing effect highlights the need for automating the process of assessing a service’s performance evolution after a resource allocation, and for proactively preventing reducing resources too aggressively, to avoid latency spikes with long recovery periods. We observe that to mitigate
a QoS violation, the manager must increase resources proactively, otherwise the violation becomes unavoidable, even if more resources are allocated a posteriori.

4. Importance of boundaries of the resource space: Given the large resource allocation space in microservices, it is essential for any resource manager to quickly identify the boundaries of that space, and explore the system’s normal operation, and cannot contain any points close to the resource boundary of the service. The service’s state and history. Sinan uses this model to maximize resource efficiency while meeting QoS.

4. Contributions & Main Artifacts

To tackle the aforementioned challenges, we take a data-driven approach that abstracts away the complexity of microservices from the user, and leverages ML to assess the impact of resource allocations on end-to-end performance. We present Sinan, a scalable and QoS-aware resource manager for interactive cloud microservices. Our major contributions are:

- **Efficient boundary-aware space exploration** We design an efficient space exploration algorithm that quickly traverses the resource allocation space, and guarantees the exploration of boundary regions that may violate QoS.

- **Hybrid ML model** We design a hybrid ML model that predicts the near-future end-to-end latency and the probability of a QoS violation for a resource configuration, given the system’s state and history. Sinan uses this model to maximize resource efficiency while meeting QoS.

- **Sinan design** We build Sinan as a centralized resource manager with distributed node agents, and deploy it on a local cluster and a large cluster of GCE.

- **Real system evaluation** We deploy and validate our ML models on the two clusters mentioned above using Docker Swarm, demonstrate minimal estimation errors, and quantify Sinan’s performance and efficiency gains over prior work.

Specifically, Sinan first uses an efficient space exploration algorithm to traverse key points in the resource allocation space, especially focusing on corner cases that violate QoS. This yields a high-quality training dataset used for two models: a Convolutional Neural Network (CNN) for detailed short-term performance prediction, and a Boosted Trees model that evaluates the long-term performance evolution. The combination of the two models allows Sinan to both examine the near-future outcome of a resource allocation, and to account for the system’s inertia in building up queues with higher accuracy than a single model examining both time windows. Sinan operates online, adjusting per-tier resources dynamically according to the service’s status and end-to-end QoS. Sinan is implemented as a centralized resource manager with global visibility into the cluster, and with per-node resource agents that track per-tier performance and resource utilization.

Finally, we demonstrate the explainability benefits of Sinan’s models, delving into the insights they can provide for the design of large-scale systems. Specifically, we use an example of Redis’s log synchronization, which Sinan helped identify as the source of unpredictable performance out of tens of dependent microservices to show that the system can offer practical and insightful solutions for clusters whose scale make previous empirical approaches impractical.

5. Key Results

We evaluate Sinan using two end-to-end applications from DeathStarBench [12]: a Social Network and a Hotel Reservation site. Examined applications are deployed with Docker Swarm and Locust [2] as the workload generator. We conduct experiments both on a local cluster and on GCE.

We compare Sinan against both traditionally-employed empirical approaches, such as autoscaling [3], and approaches based on queueing analysis, such as PowerChief [27]. We demonstrate that Sinan outperforms previous work both in terms of performance and resource efficiency, successfully meeting QoS for both applications under diverse load patterns. On the simpler Hotel Reservation application, Sinan saves 25.9% of resources on average, and up to 46.0% compared to other QoS-meeting methods. On the more complex Social Network service, where abstracting application complexity is more essential, Sinan saves 59.0% of resources on average, and up to 68.1%, essentially accommodating twice the load, without more resources. We also validate Sinan’s scalability on Google Compute Engine (GCE), and demonstrate that the models collected from the local cluster can be reused on GCE with only minor adjustments instead of global retraining.

6. Why ASPLOS

Sinan tackles resource management for microservices, an emerging cloud programming model. Resource management has been a popular topic in prior ASPLOS iterations, and involves hardware (resource partitioning), OS (runtime scheduling), and programming (application design and monitoring) level challenges. Sinan applies ML to microservice management, which also fits ASPLOS’s topic on ML for systems.

7. Citation for Most Influential Paper Award

For introducing ML-driven management to complex interactive microservices, and for showing that in addition to performance and efficiency gains, ML for cloud systems can be explainable, insightful, and improve the management of systems for which prior empirical techniques do not scale.
References


