Derm: SLA-aware Resource Management for Highly Dynamic Microservices

Liao Chen*‖, Shutian Luo†‖, Chenyu Lin*, Zizhao Mo*, Huanle Xu*¶, Kejiang Ye‡, and Chengzhong Xu*¶
*University of Macau, Email: yc27428, mc14889, yc17461, huanlexu, czxu@um.edu.mo
†Yale University, Email: shutian.luo@yale.edu
‡Shenzhen Institute of Advanced Technology, Email: kj.ye@siat.ac.cn

***Abstract*—Ensuring efficient resource allocation while providing service level agreement (SLA) guarantees for end-to-end (E2E) latency is crucial for microservice applications. Although existing studies have made significant contributions towards achieving this objective, they primarily concentrate on static graphs. However, microservice graphs are inherently dynamic during runtime in production environments, necessitating more effective and scalable resource management solutions.**

In this paper, we present Derm, a new resource management system designed for microservice applications with highly dynamic graphs. Our principal finding is that prioritizing different microservice graphs can lead to a substantial reduction in resource allocation. To take advantage of this opportunity, we develop three main components. The first is a performance model that describes uncertainties of microservice latency through a conditional exponential distribution. The second is a probabilistic quantification of the dynamics of microservice graphs. The third is an optimization method for adjusting the resource allocation of microservices to minimize resource usage. We evaluate Derm in our cluster using real microservice benchmarks and production traces. The results highlight that Derm reduces the resource usage by 68.4% and lowers SLA violation probability by 6.7×, compared to existing approaches.

I. Introduction

The microservice architecture, which has gained popularity in cloud data centers [4], [10], [11], is gradually replacing traditional monolithic frameworks for deploying new applications. Under such architecture, different components of an application are decoupled into a set of light-weight and loosely-coupled microservices for high scalability and flexibility [1], [13], [19], [35]–[37], [44]. Thanks to the fine granularity, resource orchestration platforms such as Kubernetes [18] can independently scale resources for each microservice when resource bottlenecks occur, rather than for the entire application, to improve resource efficiency.

For microservice applications, it is important to guarantee the tail end-to-end (E2E) latency meet service level agreement (SLA) requirement while minimizing resource provisioning [42]. However, achieving this goal is quite challenging due to two fundamental issues. First, the microservice graph formed by an application request may involve hundreds of microservices with complex dependencies [20]. Globally coordinating resource allocation among such a large scale of microservices becomes extremely difficult. This complexity is compounded when the runtime of individual microservices fluctuates dramatically. Second, microservice graphs are highly dynamic at runtime. Fig. 1 illustrates that a same application can generate *diverse graphs*, comprising entirely different microservices, depending on the specific requests it handles. Consequently, the E2E latency experienced by microservice applications varies significantly, further complicating global resource management.

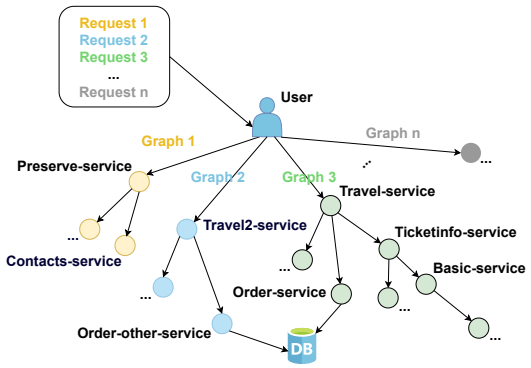


Fig. 1: Dynamic microservice graphs formed by the same TrainTicket application [45]. Microservices of diverse colors belong to distinct graphs.

Existing works have made substantial efforts to optimize resource allocation for microservice applications via deep learning or explicit modeling [5], [7], [17], [22], [29], [30], [42], [43]. In particular, deep learning methods rely on neural networks to directly predict the E2E latency in relation to the resource configuration within a specified microservice graph [29], [30], [42]. However, these methods need to evaluate many potential configurations to find the optimal resource allocation, which is not scalable for handling large microservice graphs in production environments [22]. By contrast, those modeling solutions typically quantify E2E latencies using deterministic approaches, which cannot accommodate the high degree of uncertainty in microservice latencies [7], [17], [22]. Consequently, they tend to make under-estimations and cause SLA violations.

Moreover, existing works do not investigate highly dynamic microservice graphs. However, trace analysis from Alibaba clusters shows that more than 50% of applications exhibit more

‖Co-first authors, both authors contribute equally to this work.
¶Huanle Xu and Chengzhong Xu are corresponding authors.

than two types of microservice graphs [20]. Although dynamic graphs pose challenges for guaranteeing E2E latency, they also bring good opportunities to improve resource efficiency by letting the latency of requests with different graphs meet SLA with non-uniform probabilities. As demonstrated in § II, by carefully prioritizing different graphs within the same application, the overall resource allocation can be saved by up to 20%. Therefore, it is critical to explore the difference between dynamic microservice graphs to further maximize resource efficiency.

In this paper, we introduce Derm, a new resource management system primarily for dynamic microservice applications. Motivated by the potential of graph dynamics to enhance resource management efficiency, Derm integrates three crucial elements. First, to model uncertainty, Derm characterizes microservice latency through an exponential distribution, which is a conditional probability distribution w.r.t. microservice load. This probabilistic modeling enables Derm to explicitly quantify the E2E latency of dynamic graphs. Second, Derm learns microservice graphs based on historical traces to each graph. This learning process is characterized by two key aspects: it considers both the application load and the distribution of user request arguments. Third, Derm formulates an optimization problem for resource allocation based on microservice latency uncertainties and probabilistic execution of microservice graphs. In the formulation, application requests for the complex microservice graph consisting of many microservices with high latency uncertainty need to meet the SLA requirement with a low probability to save resources. The principle behind this is that when increasing resource provisioning, the marginal improvement in E2E latency of complex graphs is less significant than that of simple graphs.

In order to make the system scalable, we have identified crucial insights that address the optimization problem, which reduces the number of variables to be solved by half. Based on this, we have incorporated a computationally efficient search method into Derm, effectively managing extensive microservice graphs. In addition, when faced with changes in load, the system requires only a linear scaling of containers, eliminating the need for frequent resolving optimization problems.

We build a prototype of Derm on top of Kubernetes [18]. Our evaluation of Derm includes real deployments in our private cluster using two benchmarks, DeathStarBench [13] and Train Ticket [45], as well as large-scale trace-driven simulations based on Alibaba traces [20]. The results demonstrate that Derm achieves high prediction accuracy for both individual microservice latency and tail E2E latency, surpassing 85%. Additionally, the graph prediction accuracy can reach as high as 93%. Furthermore, experimental results highlight that Derm can reduce the number of allocated containers by up to 68.4% and lower SLA violation probability by 6.7×, compared to the state-of-the-art approaches. In summary, we have made the following contributions in this paper:

- **Comprehensive analysis of dynamic graphs**. We have identified the main factors that lead to the dynamics of microservice graphs, which include application load and the distribution of requests arguments. In the meanwhile, we show that such dynamics can greatly affect resource allocation for microservice applications.
- **Efficient resource management for microservice applications**. Derm is the first system to optimize resource allocation for microservice applications with highly dynamic dependencies. In achieving this, Derm explicitly models latency uncertainty and graph dynamics.
- **Scalable System**. We provide a prototype implementation of Derm on top of Kubernetes [18], a widely-used container orchestration system. Derm is scalable to handle complex graphs in production environments.

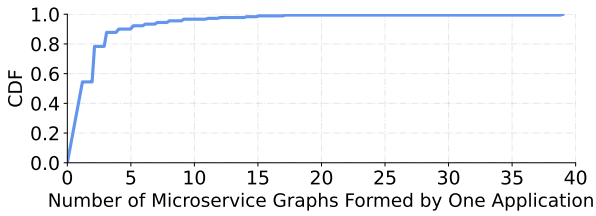


Fig. 2: Distribution of the number of microservice graphs within each application in Alibaba traces.

II. Background and Motivation

A. Dynamic Microservice Graphs

A production cluster often deploys various applications, and each application provides a variety of different applications to serve user requests [21]. An application typically contains multiple microservices running in hundreds of containers (with the same configuration) [20]. Notably, in order to handle user requests effectively, each container spawns multiple threads for processing purposes. A user request is typically sent to an entering microservice, e.g., Nginx, which then triggers a set of invocations (calls) between multiple microservices in a sequential or parallel manner. These calls along with their microservices form a *microservice graph*. The E2E latency of a request is the duration from the user sending a request to it receiving the reply.

Under the microservice architecture, each application is expected to provide multiple functionalities. For instance, in Alibaba clusters, the online shopping application needs to give discounts to users from time to time. As a result of these diverse functionalities, the execution environment of microservices varies significantly when handling different user requests, leading to dynamic microservice graphs. The illustration in Fig. 1 demonstrates how various requests originating from the same application can generate different microservice graphs during runtime. To provide a comprehensive overview, we also estimate the number of distinct microservice graphs formed by all applications in Alibaba clusters [20]. As depicted in Fig. 2, over 50% of applications contain at least two types of graphs. Moreover, approximately 10% of applications can form five distinct graphs, and these applications are typically more popular than others.

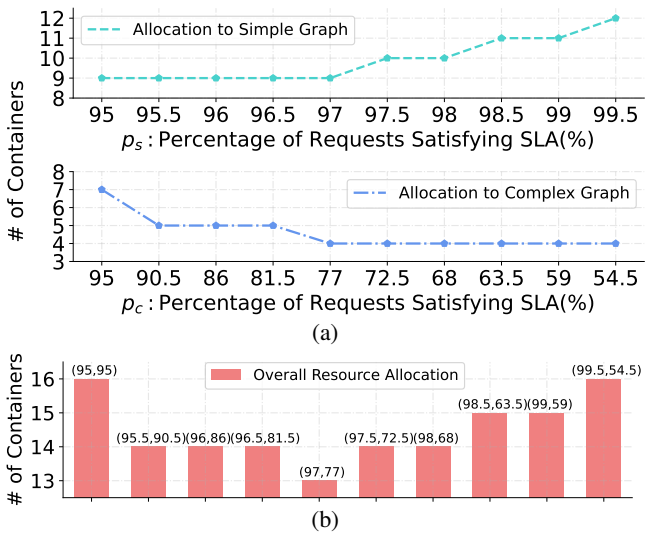


Fig. 3: Non-uniform resource allocation. (a) Allocation among different graphs within an application. (b) The overall resource usage under different (p_s, p_c) pairs.

B. Opportunities: Non-uniform Resource Allocation among Dynamic Graphs

In microservice applications, it is crucial to guarantee that the tail E2E latency, such as P95, for all user requests complies with the service level agreement (SLA), emphasizing not only the average latency [22], [42]. Moreover, the E2E latency of requests sharing the same microservice graph may exhibit substantial variability due to the inherent uncertainty associated with processing user requests. Consequently, the P95 latency is typically several times greater than the P50 latency. Given the highly dynamic nature of microservices, such gaps between different graphs within the same application tend to be even more substantial. Specifically, we executed DeathStarBench and TrainTicket within our cluster and gathered tail latency data from various graphs in these benchmarks. The results reveal that the gap in tail latency between simple and complex graphs in TrainTicket and DeathStarBench can reach up to an average of approximately 3.9 times and 2.4 times, respectively. Furthermore, upon analyzing Ali traces, this contrast becomes notably pronounced, averaging 20 times across varying loads and potentially exceeding 35 times under high load.

Because of these gaps, assigning non-uniform quantiles to diverse graphs offers a good opportunity to optimize resource allocation. For uncomplicated microservice graphs comprising less sensitive microservices, the allocation of a few additional resources can effectively reduce tail latency. Conversely, improving latency performance for complex microservice graphs necessitates a considerably larger resource allocation. Consequently, it is advantageous to redistribute performance pressure among microservice graphs, specifically by increasing the proportion of user requests meeting the SLA for simple graphs while decreasing the respective proportion for complex graphs. By doing so, we can guarantee that the tail latency for all requests from various graphs remains within SLA. Simultaneously, we can reduce the total resource usage.

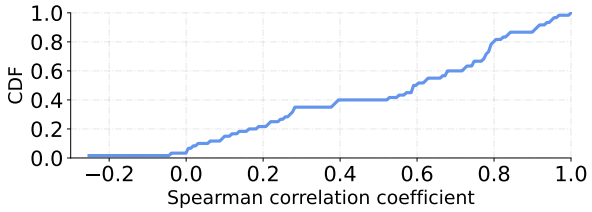


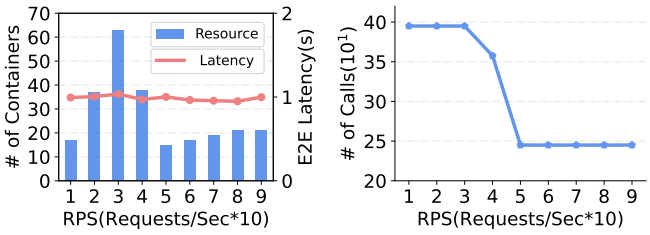
Fig. 4: The cumulative distribution highlights the strong correlation between graph dynamics and application load, as revealed by the analysis of trace data from Alibaba clusters. Here, application load is quantified by the number of requests per second (RPS).

In order to illustrate this idea, we perform experiments utilizing the ticket query application included in the Train Ticket application [45] under a fixed load. This application forms two distinct microservice graphs. In this experiment, the ratio between two different microservice graphs formed by user requests is roughly 9:1 where 9 is for the simple graph. By allocating varying numbers of containers, we can configure different percentages of user requests that comply with SLA for both graphs, while maintaining a predetermined proportion (95% by default) of all requests that fulfill the same SLA. Specifically, let $p = 95\%$, the proportions of requests that must satisfy the SLA requirements for the simple graph (p_s) and the complex graph (p_c) should follow the relationship $0.9 * p_s + 0.1 * p_c = p$. This implies that for each chosen p_s value, there exists a corresponding value of p_c, as depicted in Fig. 3(a). Fig. 3(b) illustrates the corresponding overall resource allocation for different pairs of (p_s, p_c). The optimal resource allocation is achieved when $p_s = 97\%$ and $p_c = 77\%$. This results in a 20% reduction in overall resource allocation in comparison to the uniform setting. Furthermore, raising p_s from 95% to 97% requires a mere 5% increase in resources. By adjusting p_c from 95% down to 77%, resource allocation can be reduced by approximately 45%. Note that this example only considers two types of microservice graphs. As the complexity of graph dynamics grows, optimally assigning percentages to various graphs may lead to even greater improvements.

Takeaway. Graph dynamics present a unique opportunity to enhance resource efficiency by globally adjusting the proportion of requests that meet SLA requirements across all microservice graphs. Nevertheless, determining the optimal percentage is quite difficult, as it relies on various factors such as load, E2E latency of each graph, and the overall distribution of microservice graphs.

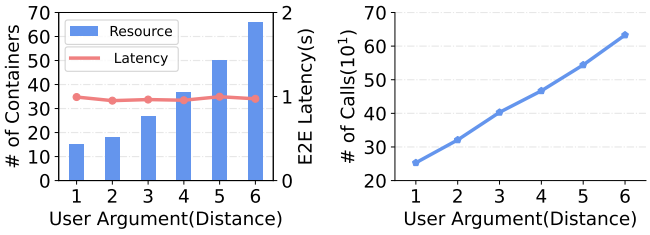
C. Challenges: Complex Characteristics of Graph Dynamics

The distribution of microservice graphs frequently changes at runtime. However, these graphs remain inaccessible until user requests are entirely processed. This situation presents a significant challenge in achieving efficient resource management that accounts for graph dynamics. In this part, we investigate how graph dynamics can be impacted by various factors in order to effectively estimate the distribution of microservice graphs during runtime.



(a) (b)

Fig. 5: Resource requirements vary in a non-monotonic manner as loads increase. (a) The number of allocated containers and the corresponding E2E latency. (b) The total number of calls between all microservices within each graph.



(a) (b)

Fig. 6: The impact of varying request arguments. (a) Resource allocation and the corresponding E2E latency. (b) The total number of calls within each graph.

Generally, the origins of graph dynamics can be classified into two categories: internal and external factors related to application. On the one hand, internal factors primarily involve changes in microservice states, such as updates to stateful services (e.g., databases) or iterative microservice development, leading to graph alterations even when processing identical requests. On the other hand, external factors arise from sources outside the application, such as request arguments or network conditions, which can provoke changes in the microservice execution logic.

Application loads can significantly influence microservice graph dynamics. It turns out that the internal factor is closely linked to the application load. Trace analysis from Alibaba clusters reveals that 95% of applications show a positive correlation between graph dynamics and load, and nearly 80% of applications display a strong correlation, as illustrated in Fig. 4. The reason behind this is that, when handling varying numbers of user requests, the storage states of stateful services can exhibit significant differences.

Applications typically require increased resource allocation to fulfill SLA requirements when accommodating heavier loads. However, this principle may not always be applicable to microservice systems, as the influence of loads on graph dynamics can alter the situation. To demonstrate this, we conducted an additional experiment using the Train Ticket application. As depicted in Fig. 5(a), when experiencing low loads ($\leq$ 30 requests/sec), the number of allocated containers should rapidly increase in response to the load in order to satisfy SLA, which requires the P95 E2E latency must be within 1 second. However, as load further increases ($>$ 30 requests/sec), the required resource allocation drops dramatically, by 47%. In this case, microservice graphs typically exhibit greater simplicity under extremely high loads. This can be attributed to the reduced number of remaining tickets and the consequent simplification of execution logic, which requires fewer conditional checks. As illustrated in Fig. 5(b), microservice graphs consist of nearly 400 edges under low loads, but only about 250 edges under high loads. In this context, each edge represents a call between a pair of microservices. Generally, a lower count of edges corresponds to a simpler graph, consequently diminishing the application's resource allocation requirements.

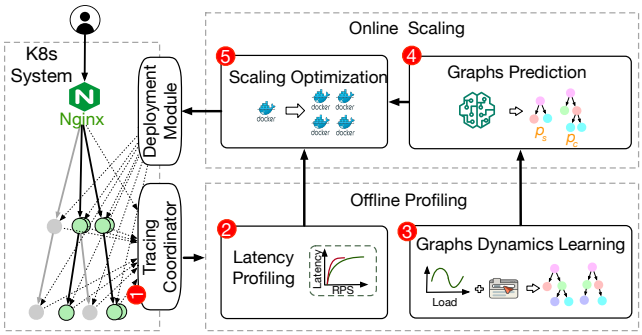


Fig. 7: The system architecture of Derm.

The arguments of user requests also have a heavy impact on graph dynamics. The external factor primarily pertains to user arguments. When users submit requests to applications, they often specify corresponding arguments. These arguments significantly impacts the microservice execution logic, leading to varying microservice graphs and E2E latency.

We conducted an experiment using the Train Ticket application to investigate optimal resource allocation under different user arguments while maintaining a consistent load. As illustrated in Fig. 6(a), there is a gradual increase in the number of allocated containers as the distance (a crucial argument in user requests) grows. Specifically, only 12 containers are needed when all user requests have a distance of 1, but this number increases six times when the distance expands to 6. Correspondingly, the number of edges present in microservice graphs demonstrates an almost linear growth in relation to distance, as depicted in Fig. 6(b).

Takeaway. It is essential to carefully model the distribution of microservice graphs in relation to application loads and user arguments in order to attain optimal resource efficiency within microservice applications.

III. Overview of Derm Architecture

In this section, we present the overall architecture of the Derm system. Derm is a microservice resource manager that dynamically scales the number of containers for microservice applications to satisfy SLA requirements.

Derm deploys a *Tracing Coordinator* (❶ in Fig. 7) on top of Jaeger, a popular tracing system [3]. *Tracing Coordinator* collects microservice graphs at runtime for all applications and extracts the individual microservice latency for all deployed microservices based on historic traces.

Derm includes an *Offline Profiling* component, which deploys two key modules, Latency Profiling (❷ in Fig. 7) and Graph Dynamics Learning (❸ in Fig. 7). Specifically, Latency Profiling module fetches samples of microservices latency under different application loads from *Tracing Coordinator*. It fits these latency samples with an exponential distribution, whose parameters are a linear function w.r.t. loads. Graph Dynamics Learning module collects microservice graphs from *Tracing Coordinator* and learns the relationship between the distribution of microservice graphs, loads, and request arguments based on machine learning (ML).

The core component of Derm is *Online Scaling*, which utilizes profiling results to determine the number of containers that should be scaled for each microservice. It further consists of two modules, Graph Prediction (❹ in Fig. 7) and Scaling Optimization (❺ in Fig. 7). Specifically, based on the observed loads and the corresponding request arguments, the Graph Prediction module predicts the distribution of all microservice graphs that can be formed by all requests. Stem from this distribution and profiled microservice latency models, the Scaling Optimization first quantifies the E2E latency of each microservice graph. It then formulates an online optimization problem to minimize resource allocation. Moreover, this module also adopts convex optimization methods to find the optimal scaling results. Finally, container scaling actions are executed on the underlying Kubernetes cluster through the deployment module.

IV. Microservice Graph Latency Modeling

In this section, we derive probabilistic models that enable precise quantification of E2E latency distribution for a given microservice graph. By utilizing the graph prediction outcomes with the E2E latency model, Derm can effectively facilitate resource allocation that accommodates the highly dynamic nature of microservice applications. Specifically, our initial focus is on profiling the latency of individual microservices, with a particular emphasis on accurately capturing uncertainty. Following this, we elucidate the methodology by which Derm employs the latencies of individual microservices to construct probabilistic models of E2E performance.

A. Profiling individual microservice latency

The latency of a microservice is established by subtracting the response times of all downstream microservices from its own response time [22]. This response time encapsulates the duration between the arrival of a request and the moment the corresponding response is dispatched. To conduct this calculation, request timestamp data can be acquired from tracing systems such as Jaeger [3]. Notably, the response time encompasses both queuing time and execution time, effectively reflecting the resource pressure experienced by an individual microservice. Given that a microservice graph typically comprises numerous sequential and parallel interactions among fine-grained individual microservices [20], the latency of each individual microservice has a global impact on the overall E2E performance. Therefore, it becomes crucial to accurately profile the latency of each microservice.

Microservice latency exhibits a significant level of uncertainty when processing requests, stemming from the potential for microservice calls to execute diverse program branches and each microservice offering multiple interfaces for external invocation. Consequently, the execution times of microservices can experience notable fluctuations, contingent upon the chosen interface and execution branch. This, subsequently, gives rise to considerable variations in queuing times. Furthermore, the average latency of individual microservice i typically increases with the load of each microservice container due to the increased queuing time. Taking uncertainty into account, we represent the microservice latency L_i as a random distribution, conditional on load W_i, that is:

$$P_i(l, w) = \Pr(L_i = l | W_i = w). \tag{1}$$

Here, P_i denotes the probability density function (PDF) of the latency of microservice i. This PDF and the corresponding cumulative distribution function (CDF) can be well-represented by exponential functions. Referring to the experiment conducted on benchmarks and Alibaba clusters, as presented in § VIII-B1, the fitted exponential distribution closely matches the actual distribution for various microservices. This outcome can be further explained by queuing theory. In the case of an M/M/1 queue, the response time of a service follows an exponential distribution when both the request arrival process and service process are governed by a Poisson process [9], which is one of the most widely-used counting processes.

Derm's Latency Profiling Module employs the exponential distribution to estimate the latency of individual microservices under specific loads. This module periodically retrieves running samples of microservice latency under various loads from historical traces and utilizes these samples to determine the parameters of the conditional exponential distributions. In particular, Derm models the CDF function of latency L_i for microservice i using the following expression:

$$\Pr(L_i \le l | w_i) = \begin{cases} 1 - e^{-\lambda_i (l - l_i^0)}, & l \ge l_i^0, \\ 0, & l < l_i^0. \end{cases} \tag{2}$$

Here, w_i denotes the load that each container of microservice i must handle. λ_i is recognized as the rate parameter of the exponential distribution, which is both microservice-dependent and a function of the load w_i. The rate parameter illustrates the extent of latency uncertainty; specifically, a smaller λ_i implies higher uncertainty. l_i^0 represents the intercept point, capturing the execution time of microservice i without queuing. Furthermore, the Latency Profiling module adopts non-linear least squares [27] to calculate λ_i for each w_i.

A simple approach to generating the final latency profile involves creating a unique model for each λ_i with a given w_i. However, this approach incurs high computational overhead due to the vast range of w_i values and does not accommodate instances with new w_i values. As such, it is essential to derive a model that can estimate λ_i under varying w_i values. According to queuing theory, when both the arrival and service

processes follow Poisson process, the parameter λ_i can be measured as the difference between the service rate and arrival rate [9]. Under each configuration of containers, the service rate can be considered constant. Therefore, Derm models λ_i as a linear function with respect to the load w_i, that is,

$$\lambda_i = a_i \cdot w_i + b_i. \tag{3}$$

Derm collects multiple pairs of $\{\lambda_i, w_i\}$ for each microservice i and adopts linear regression (LR) to learn the parameters a_i and b_i. A significant advantage of linear regression is its requirement of only a few samples for profiling, which results in minimal overhead. Additionally, as demonstrated in the experiment described in § VIII-B1, the accuracy of linear regression is shown to be on par with other complex models. Importantly, Derm can leverage this regression to systematically determine the optimal resource allocation, as elaborated in § VI-B.

B. Quantifying E2E latency

Derm estimates the E2E performance of a system by combining the latency distributions of all microservices within a graph. This involves operations on both sequential and parallel calls, which can be discerned from historical traces. For instance, consider a microservice graph that contains two microservices, denoted as i and j, executing sequentially. The E2E latency is in the form of the sum of individual latencies, so PDF can be formulated as:

$$\Pr(Z = z) = \sum_{k=0}^{z} \Pr(X = k, Y = z - k), \tag{4}$$

where two random variables, X and Y, represent the latency of microservices i and j, respectively. Microservice latency primarily captures the duration of time that a request spends within each individual microservice. This duration is calculated by subtracting the response times of all downstream microservices from its own response time, and if a downstream microservice is blocking, a microservice can continue processing other requests normally by utilizing newly created threads. Consequently, this metric offers an independent measurement of the performance of each microservice, effectively isolating any potential impact from downstream services. Consequently, X and Y can be treated as independent random variables and the PDF can be expressed as a convolution operation: $\Pr(Z = z) = \sum P_i(X = k) \cdot P_j(Y = z - k)$, where P_i denotes the PDF of microservice i. By utilizing the latency models from profiling individual microservices, the distribution of E2E latency can be formulated using the following closed-form expression:

$$\Pr(Z = z) = \frac{\lambda_i \lambda_j}{\lambda_i - \lambda_j} \left(e^{-\lambda_j (z - l_i^0 - l_j^0)} - e^{-\lambda_i (z - l_i^0 - l_j^0)} \right). \tag{5}$$

On the other hand, if two microservices are called in parallel within a graph, the cumulative distribution of the E2E latency takes the form of:

$$\Pr(Z \leq z) = \Pr(X \leq z, Y \leq z). \tag{6}$$

Similar to convolution operation, in this case, we can simplify the above CDF as $\Pr(Z \leq z) = P_i(X \leq z) \cdot P_j(Y \leq z)$. These results can be readily extended to complex graphs with a large number of microservices. Specifically, when a graph contains n microservices with sequential execution, the PDF of E2E latency can be represented by cascading multiple convolution operations, as expressed below:

$$\Pr(Z = z) = \prod_{k}^{n} \lambda_k \cdot \sum_{i=1}^{n} \frac{e^{-\lambda_i (z - \sum_{i=1}^{k} l_i^0)}}{\prod_{i \neq j} (\lambda_j - \lambda_i)}. \tag{7}$$

This expression can also accommodate parallel executions. In particular, Derm selects the distribution of microservices with the smallest rate parameter as the ultimate distribution of response times for parallel execution.

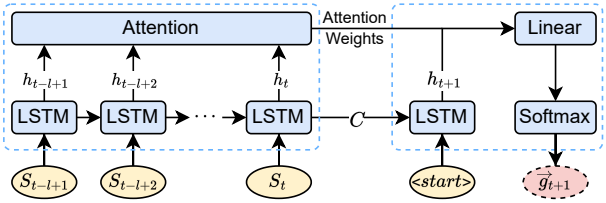


Fig. 8: The architecture of graph learning model under Derm.

V. Microservice Graph Dynamics Prediction

As explained in § II-C, the dynamics of microservice graphs are heavily influenced by application loads and the specific characteristics of request arguments. This section describes how Derm effectively leverages these attributes to predict the distribution of microservice graphs at runtime.

A. Feature Representation

Leveraging historical data, the Graph Dynamics Learning module employs machine learning techniques to construct a model for estimating graph distribution. This module is designed to learn graph dynamics through a model $\mathcal{M}$ by utilizing historical data on request argument distributions, loads, and microservice graphs. The fundamental concept involves assigning a class label to each type of microservice graph generated by online applications, thereby transforming graph learning into a conventional classification task.

Specifically, the Learning module classifies request arguments into distinct classes based on their values and subsequently computes a percentage for each class. These percentages are aggregated into a vector, $\vec{a}_t$, which represents the distribution of arguments at a specific time t. Similarly, microservice graphs are categorized based on their topologies (elaborated in § VII), and the distribution of these graphs is captured in another vector, $\vec{g}_t$. By incorporating load data w_t, a sample at time t is represented as $s_t = \langle \vec{a}_t, \vec{g}_t, w_t \rangle$. Given a time series sequence $S = \{s_{t-l+1}, s_{t-l+2}, ..., s_t\}$ over a time window l (set to 5 by default, with further details in § VIII-B), the learning model $\mathcal{M}$ aims to learn the relationship between S and the subsequent distribution of microservice graphs $\vec{g}_{t+1}$ at time $t+1$: $\vec{g}_{t+1} = \mathcal{M}(S)$.

B. Graph Dynamics Learning with LSTM

Dynamics within microservice graphs naturally exhibit temporal patterns during runtime. Consequently, we incorporate $\mathcal{M}$ into the Long Short-Term Memory (LSTM) model [16],

[32], [41] to leverage the superior performance of LSTM over traditional models like exponential smoothing and ARIMA, particularly in managing long-term dependencies within the data. Furthermore, information within time series data is unevenly distributed, with certain time steps containing more critical features. For example, during peak loads, microservice graphs display heightened dynamics. As a result, Derm integrates an attention mechanism with LSTM to focus on crucial features, thereby enhancing prediction accuracy.

The model $\mathcal{M}$, illustrated in Fig. 8, consists of an encoder, decoder, and an attention module. In detail, the encoder processes the input sequence S and produces the latent variable C, summarizing contextual information. The decoder incorporates an LSTM module, a linear layer, and a softmax layer. It utilizes attention weights and the extracted latent variable to generate the output distribution of microservice graphs for the next timestep, denoted as $\vec{g}_{t+1}$. Derm employs stacked LSTM cells with two layers, where the encoder and decoder possess 128 and 32 features in the hidden states, respectively.

C. Graph Dynamics Prediction

The Graph Prediction module predicts the distribution of microservice graphs by leveraging the graph leaning model $\mathcal{M}$, which is based on the current load and characteristics of request arguments. In order to minimize computational overhead associated with scaling optimization, particularly for applications that can produce numerous microservice graphs, Derm focuses on graph types that occur most frequently, as only 95% of requests (by default) must meet SLA requirements. As illustrated in Fig. 2, production clusters may contain over 40 different graph types, but the majority are seldom encountered. Therefore, for less frequent graphs with an overall predicted probability below 2%, Derm allocates one container by default to each microservice within these graphs to reduce SLA violations.

VI. Optimal Resource Scaling

In this section, we will provide a detailed explanation of how resource scaling is achieved with Derm. Currently, Derm uses a dynamic scaling approach that involves adjusting the number of containers (with the same configuration) allocated to each microservice as needed.

A. Basic Optimization Model

The Scaling Optimization module minimizes resource allocation while ensuring that the tail E2E latency meets the SLA requirement. However, due to the constantly changing nature of microservice graphs, Derm must allocate an appropriate amount of resources to each type of microservice graph, which yields the following formulation:

$$\begin{aligned} &\min_{\vec{w}} \sum_{g\in G}\sum_{i\in g}\frac{p_g\cdot W}{w_i}, \\ &\text{subject to,} \sum_{g\in G} p_g\cdot \Pr(Z_g \le \text{SLA}) \ge p. \end{aligned} \tag{8}$$

In (8), W denotes the application load, and G denotes the set that contains all graph types within each application. Additionally, p_g represents the percentage of requests that make up microservice graph g, which is obtained from the Graphs Prediction module. The optimization variable w_i captures the load that is processed by each container of microservice i and $\vec{w} = \{w_i\}$. Consequently, the number of containers allocated to microservice i can be calculated using $\frac{p_g W}{w_i}$. The random variable Z_g denotes the E2E latency yielded by graph g. The required percentage of requests with E2E latency below the SLA is represented by $p = 95\%$.

Rather than requiring each microservice graph to independently satisfy the SLA requirement, Derm forces the weighted E2E latency between all graphs to meet the SLA with probability p. This enables Derm to coordinate resource allocation globally across all microservices within the application. So, the constraint in (8) can be reformulated as:

$$\sum_{g\in G} p_g \cdot \int_0^s P_g(z)dz \ge p, \tag{9}$$

where $P_g(z)$ denotes the PDF of the E2E latency, while s represents the SLA. Based on the characterization results presented in § IV-B, the constraint can be rewritten as:

$$\sum_{g\in G} p_g \cdot \sum_{i=1}^{n}\prod_{k}^{n}\lambda_k \cdot \frac{e^{-\lambda_i(s-\sum_{i=1}^{k} l_i^0)}}{\lambda_i\prod_{i\neq j}(\lambda_j-\lambda_i)} \le 1-p. \tag{10}$$

In this context, λ_i is defined as $\lambda_i = a_i w_i + b_i$, which is updated when the per-container load, w_i, changes. By identifying the optimal solution for each w_i within this optimization problem, Derm can compute the necessary number of containers to be scaled for each individual microservice.

B. Scalable System Design

Designing a scalable system is essential for rapidly adapting to load fluctuations while maintaining compliance with SLA requirements. This poses a great challenge as conventional optimization methods, such as gradient-based approaches, can result in substantial overhead in identifying optimal solutions, particularly for large-scale graphs. To address this challenge, we first explore insights when finding optimal solutions for simpler graphs and then utilize these insights to tackle more complex graph scenarios.

For a simple graph g consisting of two microservices i and j, the optimization problem becomes:

$$\begin{aligned} &\min_{\vec{w}} \frac{W}{w_i}+\frac{W}{w_j}, \\ &\text{subject to,} \frac{\lambda_j e^{-\lambda_i(s-l_0^i-l_0^j)}}{(\lambda_j-\lambda_i)} + \frac{\lambda_i e^{-\lambda_j(s-l_0^i-l_0^j)}}{(\lambda_i-\lambda_j)} \le 1-p. \end{aligned} \tag{11}$$

It is worth noting that maximizing the values of w_i and w_j can be advantageous since it implies lower resource allocation for the two microservices. Consequently, minimizing the values of λ_i and λ_j is beneficial since λ_i decreases with an increase in w_i, as described in Eq. (3). By rearranging terms, the above constraint can be reformulated as:

$$\frac{e^{-\lambda_i(s-l_0^i-l_0^j)}-1+p}{\lambda_i} \le \frac{e^{-\lambda_j(s-l_0^i-l_0^j)}-1+p}{\lambda_j}. \tag{12}$$

This formula suggests that both microservices should share the same rate parameter, meaning $\lambda_i = \lambda_j$. If this were not the case, the larger parameter could be further decreased to still satisfy the inequality. Adhering to this principle, we conclude that within a microservice graph consisting of three unique microservices, the optimal solution is achieved when at least two of these microservices share an identical rate parameter. Consequently, in the case of more complex graphs, the Scaling Optimization module of Derm can effectively allocate resources by computing parameters for only half of the microservices present in the graph, substantially diminishing the complexity of the problem.

Towards the end, Derm concentrates on examining the K.K.T. (Karush-Kuhn-Tucker) conditions associated with the optimization problem [6], [15]. These conditions specify the first derivative tests for a feasible solution in nonlinear programming to be optimal. Derm utilizes an iterative approach to solve the K.K.T. equations, updating relevant parameters including w_i and λ_i in a sequential manner until convergence is achieved. Additionally, an efficient initialization of the rate parameter, λ_i, can speedup the iterative process. Derm accomplishes this by judiciously choosing pairs of microservices that share the same rate parameter. The key principle involves selecting pairs according to their sensitivity to load, represented by a_i in Expression (3). The rationale for this selection is that, when maintaining a fixed rate parameter, a lower sensitivity results in a greater load capacity for each container. This optimization method is significantly more efficient than gradient descent, which demands careful hyperparameter tuning and a multitude of iterations for convergence. Experimental results indicate that the search method utilized by Derm necessitates a mere five iterations to reach convergence for graphs containing 50 microservices.

Furthermore, when only the load W changes, the optimal solution to the optimization problem for w_i remains the same. In this sense, if there is no update in the distribution of microservice graphs, Derm can simply scale resources linearly, without the need to re-solve the optimization problem. This can further reduce the system overhead.

VII. Derm Implementation Details

We have developed a prototype of Derm atop Kubernetes [18], deployed with Jaeger [3], a tracing system used for collecting latency samples and analyzing microservice graphs.

In a typical setup, Jaeger primarily captures metadata of HTTP requests. To record request arguments within Jaeger, entering microservices are configured to tag these arguments and send them to Jaeger. The *Tracing Coordinator* retrieves these arguments recorded in Jaeger based on their associated tags. Recording request arguments is a common practice in many companies to facilitate detailed analysis and optimize the microservice architecture [2], [33]. As both the tracing system and the microservice system operate within a private cluster, the risk of privacy breaches is significantly reduced.

To extract microservice graphs, the *Tracing Coordinator* designates the entering microservice of each application as the graph's root and adds an edge for every pair of dependent microservices. This traversing operation is repeatedly applied to form a graph until no downstream microservices remain. Additionally, for graph classification, we acquire the adjacency matrix of each graph and determine whether two graphs are identical by comparing their adjacency matrices.

The Profiling module employs lightweight ML models. In order to improve profiling accuracy, Derm incorporates a strategy to retrain the profiling model over time due to relatively low training overhead. This retraining process encompasses both recent and historical data to learn the periodic patterns in the distribution of arguments.

The Scaling Optimization module is designed as a Kubernetes plugin and is developed using the Kubernetes Python client library. This module accelerates the whole optimization procedure by improving parallelism. Specifically, it creates multiple processes that simultaneously compute the rate parameter for different microservices within each iteration.

VIII. System Evaluation

A. Experiment Setup

Microservice Benchmarks: We evaluate Derm by employing two open-sourced microservice benchmarks, **DeathStarBench** [13] and **Train Ticket** [45]. DeathStarBench encompasses three diverse applications, namely Social Network, Hotel Reservation, and Media. These applications contain 36, 15, and 38 unique microservices respectively. Meanwhile, Train Ticket consists of a suite of three distinct applications: ticket booking, ticket querying, and food ordering. Notably, the applications incorporated within Train Ticket generate dynamic microservice graphs at runtime, with an average of over 20 microservices invoked per application. In order to perform a comprehensive evaluation of Derm, we also leveraged traces from the widely-used online shopping application, **Taobao**, provided by Alibaba [20]. The Taobao application comprises more than 2500 microservices. These traces were replayed to simulate ontainer scaling in production clusters.

Cluster Setup: We deployed Derm in a private cluster consisting of 10 two-socket physical hosts. Each host is equipped with 52 CPU cores and 128 GB RAM. Each microservice container is configured with 0.2 core and 1GB memory.

Load Generation: We discovered that the maximum throughput of applications in our cluster is 200 (1600) requests/sec under TrainTicket (DeathStarBench). We conducted an evaluation of Derm using static and dynamic loads. Regarding the static load, we generated a diverse range of requests, spanning from as low as 10 to the threshold of 200 (1600) requests per second for each application. Dynamic loads are derived from Alibaba clusters [20]. We adopted the P95 E2E latency as the metric for tail latency, aiming to keep it below the SLA targets. And the range of SLA targets is set from 200 ms (low) to 550 ms(high) in the experiments.

Baseline Schemes: We compared the performance of Derm against the state-of-the art solutions, including Erms [22], Sinan [42], Kraken [38], and Firm [30].

- **Erms:** It profiles microservice latency through a piece-wise linear function and computes the latency target for each microservice in the static graph through an explicit model.
- **Sinan:** It estimates the end-to-end latency of a static graph under different resource configurations via deep learning algorithms. Based on this estimation model, Sinan identifies an efficient resource configuration that prevents SLA violations.
- **Kraken:** To mitigate the cold start of serverless instances and improve resource efficiency, Kraken profiles the invocations through the Markov process and leverages the transition probability matrix to predict the invocation times for each instance to reserve resources in advance.
- **Firm:** It first identifies a critical microservice on each critical path of a static graph that has a heavy impact on the end-to-end latency and then applies reinforcement learning to tune resource allocation for this microservice.

B. Prediction Accuracy

To assess the accuracy of Derm's profiling modules, we deployed DeathStarBench and TrainTickets on our cluster and gathered one-day traces for each microservice. Concurrently, we assembled datasets of equivalent magnitude for microservices within the Ali trace.

Training Data Collection: In training the latency profiling module, the dataset includes 2000 samples per microservice under each load, with 20% of the data reserved for testing. The latency fitting time per microservice under a single load does not exceed 0.1 seconds. Additionally, we only require 10 sets of data under different loads to fit the linear relationship between exponent λ in latency profiling module and load. Consequently, the total training duration for individual microservice latency profiling does not surpass 1 second. For graph learning module, exemplified by the more numerous and complex microservice graphs in Ali Trace compared to the benchmarks, collected data spanning a week, consisting of 62,000 samples. The training set includes 12 types of user arguments, distributed evenly, and incorporates 36 different topological structures. The load data is evenly distributed within the range of 10 to 500 requests/second. Approximately 15% of these structures are complex graphs, while simple graphs account for 35%. To evaluate the model's robustness, the test set includes complex graphs constituting 20% and simple graphs comprising 30%. Graph learning can be fully trained in less than 10 minutes for case of Ali trace. By contract, benchmarks such as TrainTicket and DeathstarBench, requiring less than one-tenth of the dataset of Ali trace to be well trained, with the overhead being within one minute.

1) Microservice latency estimation accuracy: To characterize the long-tailed distribution of individual microservice latency, we adopted four commonly-used distribution models, including exponential distribution, log-logistic, Burr, and Pareto, to fit the distribution of microservice latency. As shown in Fig. 9(a), the accuracy of the exponential distribution can reach as high as 90%, outperforming other distributions.

Besides linear regression, we implemented three sophisticated models, including SVR, XGBoost, and Random Forest,

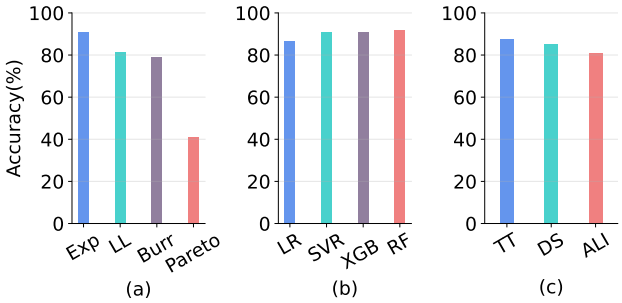


Fig. 9: Accuracy of latency profiling. (a) Profiling distribution using exponential (Exp), log-logistic (LL), burr and pareto. (b) Fitting model using LR, SVR, XGBoost (XGB), and Random Forest (RF). (c) E2E latency estimation on Train Ticket (TT), DeathStarBench (DS), and Alibaba traces (ALI).

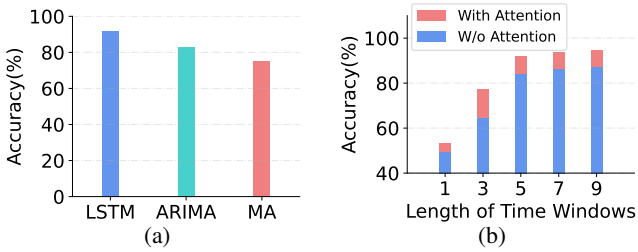


Fig. 10: The prediction accuracy of graph dynamics. (a) Time-series prediction of microservice graphs using LSTM, ARIMA and Mean Average (MA). (b) Accuracy under different length of time windows with and without attention mechanism.

to predict the value of the parameters of the exponential distribution with respect to load for each microservice. As depicted in Fig. 9(b), the accuracy of linear regression reaches 86%, which is only a marginal 5% less than that of sophisticated models. This accuracy is high enough for resource management and, importantly, linear regression model can be leveraged to simplify the optimization problem in § VI-A.

In addition, we carried out E2E latency estimation for applications on all three benchmarks. As shown in Figure 9(c), Derm achieves an accuracy of up to 85%, highlighting its effectiveness in resource scaling. While more complex models might potentially achieve even higher levels of accuracy, Derm's model enables efficient handling of dynamic graphs.

2) Graphs dynamics prediction accuracy: We have evaluated various ML algorithms to assess their accuracy in predicting the distribution of microservice graphs. As depicted in Fig. 10(a), the LSTM algorithm demonstrated the highest prediction accuracy, reaching 92%, which is 10.8% and 22.6% higher than ARIMA and MA, respectively. Furthermore, Fig. 10(b) indicates that as the length of the time window increases, the predictive accuracy of the LSTM algorithm also improves, peaking at 94.2%. Nevertheless, the rate of improvement gradually diminishes, accompanied by a significant increase in training cost. Consequently, we opted to set the time window at 5 under Derm's implementation. Moreover, the incorporation of an attention mechanism enhanced the model's performance, resulting in an average accuracy increase of 12.8%.

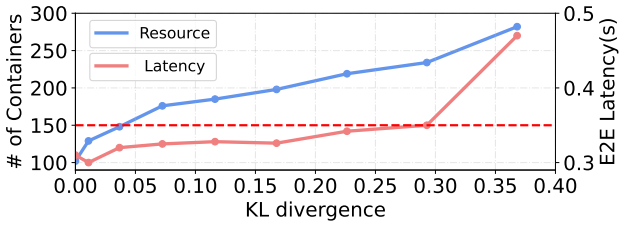


Fig. 11: Resource usage under varying prediction accuracy.

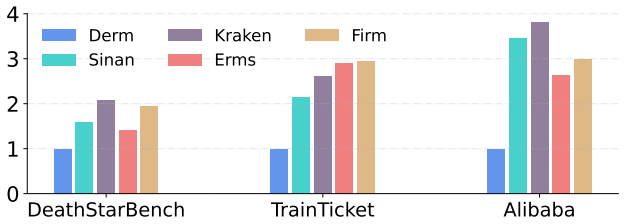


Fig. 12: Normalized resource usage under all benchmarks.

The aforementioned results emphasize the accurate microservice graph dynamics learning capability of Derm. To provide a clear illustration of the impact of prediction accuracy on resource allocation, we conducted a new experiment using the Ticket Booking application. In this experiment, we manually controlled the load and user request arguments. The distribution of microservice graphs was randomly generated and then input into Derm's optimization model to ascertain resource allocation for each microservice. We utilized Kullback-Leibler (KL) divergence as a measure of accuracy for the randomly generated distributions, and then evaluated the performance of each allocation within our cluster accordingly. As shown in Fig. 11, predictions with lower accuracy, i.e., a KL divergence of 0.2, may lead to an over-allocation of resources by as much as 103.9%. In contrast, accurate predictions with a KL divergence of less than 0.1 result in an over-allocation of less than 65%. In extreme cases, resource allocation based on inaccurate predictions was 2.8 times the actual requirement. Surprisingly, in such situations, an excessive allocation of resources could even trigger SLA violations. This phenomenon is due to the fact that resources allocated to one graph may have reached saturation, making any further increase ineffective in improving performance. And resources allocated to other graph will decrease until reaching a critical point, leading to a precipitous drop in performance.

C. End-to-end Evaluation

1) Static load: In this part, we evaluate the resource usage and E2E latency of applications under different static loads and SLA settings. In each setting, we ran all applications for 30 minutes.

Resource saving. We quantified resource usage in terms of the number of containers allocated to all applications in both DeathStarBench and Trainticket as well as in a trace-driven simulation using Alibaba traces. Fig. 12 illustrates the average resource usage across different applications, comparing different approaches under various loads and SLA

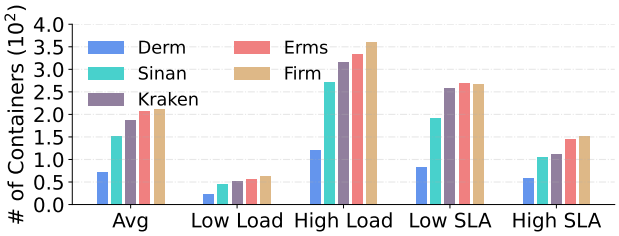


Fig. 13: Resource usage under different settings, where Avg indicates the average performance.

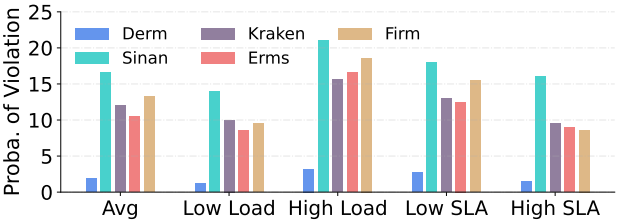


Fig. 14: SLA violation probability across all settings.

settings. The results demonstrate that Derm outperforms other approaches by a substantial margin. On DeathStarBench, Derm reduces resource usage by 36.9%, 29.3%, 51.8%, and 48.3% compared to Sinan, Erms, Kraken, and Firm, respectively. When evaluating Trainticket, the resource savings achieved by Derm are even more substantial, reaching 52.9%, 65.2%, 61.5%, and 65.9%. This can be attributed to the fact that other approaches fail to effectively incorporate the dynamics of microservice graphs. Interestingly, when evaluating performance using Alibaba traces, Derm achieves an impressive 68.4% reduction in resource usage compared to baseline approaches, surpassing the observed improvements in the benchmarks. This discrepancy can be attributed to the inherent complexity of the microservice graph in the production cluster.

We conducted a comprehensive comparison between Derm and baselines, considering various settings using TrainTicket as the benchmark due to its highly dynamic nature. As depicted in Fig. 13, Derm exhibits even more pronounced advantages in scenarios that present high load and stringent SLAs. Specifically, Derm demonstrates impressive savings of up to 75% under high-load conditions, while achieving a 60% reduction in resource usage during low-load situations. This trend is similarly observed when varying SLA requirements. Under less stringent SLA scenarios, the reduction in resource usage attained by Derm is markedly greater than in situations with higher SLA settings. Given that a lower SLA implies a relaxed latency target for each microservice, there is significant opportunity for optimization of resource usage. Consequently, under relaxed SLA conditions, Derm has been observed to conserve an additional 20% of resources.

E2E delay. Additionally, Derm exhibits an average SLA violation probability below 1.9% under TrainTicket, which is significantly lower compared to the 16.6%, 11%, 10.5%, and 15.8% observed for Sinan, Kraken, Erms, and Firm, respectively, as illustrated in Fig. 14. It is worth noting that higher loads and stricter SLAs result in an increased probability of SLA violations across all schemes.

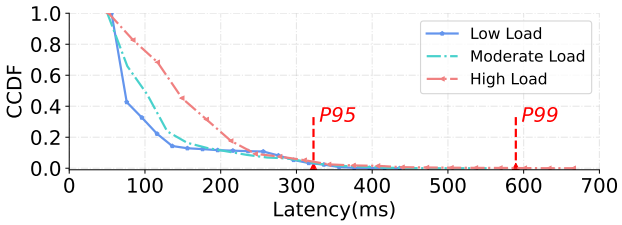


Fig. 15: E2E latency distribution.

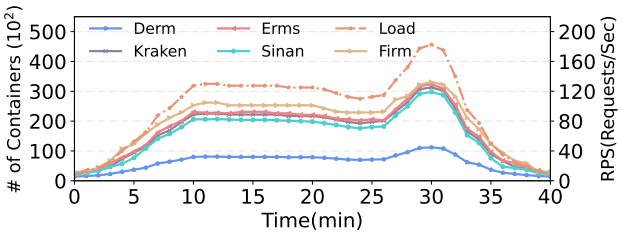


Fig. 16: Resource allocation over time.

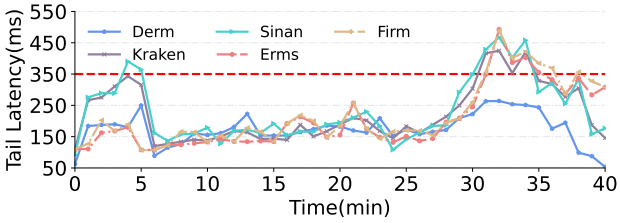


Fig. 17: P95 E2E latency fluctuates over time.

In terms of the actual tail E2E latency, Derm consistently achieves an average reduction of 15% compared to other approaches. Moreover, we have also examined the P99 latency performance in addition to the P95 latency, considering a SLA of 350ms. The results, illustrated in Fig. 15, demonstrate that the P99 tail latency remains within twice the value of P95 across various loads. This indicates that Derm effectively maintains consistent performance in terms of tail latency, ensuring that even the P99 latency remains within an acceptable range for dynamic applications.

2) Dynamic load: We also generated a dynamic load based on Alibaba traces and set a SLA target of 350ms. This experiment aims to dynamically scale containers for microservices in order to satisfy the SLA using TrainTicket. As depicted in Figure 16, all examined schemes are capable of promptly responding to load fluctuations. However, the Derm scheme stands out by offering significant savings, reducing the use of containers by up to 60% in comparison to other schemes on average.

Fig. 17 illustrates the tail latency of requests submitted over a 40-minute span. It is noteworthy that Derm consistently meets SLA, even during instances of rapid load increase. In contrast, other schemes are prone to violating the SLA by 28% during peak load periods. Sinan, in particular, demonstrates a higher likelihood of SLA violation, attributable to its limited capacity to detect bottleneck microservices.

D. Microscopic Evaluation

In this part, we present the benefit brought by the two key components of Derm including graph learning, and resource optimization algorithm. We also report the overhead of Derm to emphasize its suitability for large-scale production clusters.

1) Benefit of graphs learning: In this study, our objective is to assess the enhancement brought by the incorporation of the Graphs Learning module. This evaluation involves quantifying the resource usage of all schemes, both with and without the module, under various static loads and SLA settings. To facilitate this, we adjusted the baseline schemes by integrating Derm's Graphs Learning, which predicts microservice graphs and subsequently allocates resources in line. For Derm, we input the static graph distributions, extracted from historical traces, into the optimization engine.

On average, the introduction of Graphs Learning module results in a savings of 40% in container use for Derm, as illustrated in Fig. 18. Moreover, the findings show that graph learning enhances resource efficiency across all schemes, with a notably larger improvement observed in Derm (40% versus 20%). This can be attributed to the fact that Derm's graph learning employs an accurate model to capture the variations between different microservice graphs, thereby resulting in a significant decrease in overall resource consumption.

2) Benefit of scaling optimization: We have also examined the benefits accrued from Derm's optimization module, whose primary function is to minimize the overall resource usage. Central to this optimization strategy is the selective disregard of infrequent call graphs, as Derm's design only necessitates tail latency compliance with SLAs. In order to accurately quantify this benefit, we deactivated the optimization module in Derm, requiring every graph to meet the SLA. We also repeated this procedure under all baselines. We analyzed the overall resource usage under each scheme to measure the effects. As illustrated in Fig. 19, when the optimization module was active, Derm achieved a saving of nearly 32% in terms of the number of allocated containers. This reduction is more significant than the savings observed under the baselines, primarily due to the global optimization performed by Derm.

E. Scalability of Derm

Profiling Scalability. Derm's profiling, conducted initially in a smaller setup, can be seamlessly scaled and deployed within a large-scale cluster. Specifically, latency profiling, which takes into account the number of containers, and graph modeling, which captures the dynamics of the microservice graph in relation to the application, are independent of cluster size. This independence ensures that both latency profiling and graph modeling can be applied to large-scale clusters without direct correlation to the cluster's scale. Futhermore, microservices often undergo evolution twice a week, Derm only needs to perform re-profiling as necessary when changes occur in the microservice architecture or new microservices are incorporated.

Scaling Overhead. We evaluate Derm's scaling overhead using Alibaba traces. Our results suggest that the average overhead for scaling optimization on an AMD EPYC 7452 CPU is approximately 200ms. Moreover, the computational

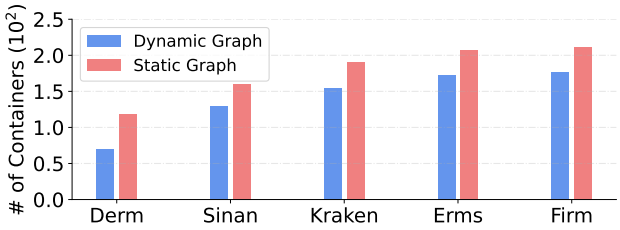


Fig. 18: Allocation w/wo graph learning under all schemes.

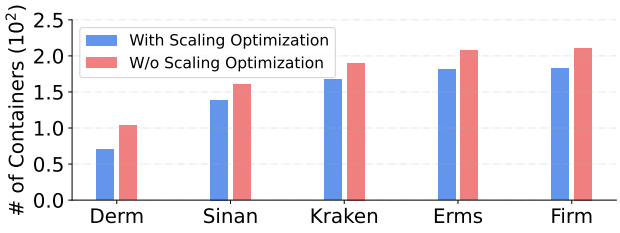


Fig. 19: Resource savings brought by scaling optimization.

overhead for the largest graph of more than 400 microservices, measures around 350ms. Given that a container often needs several seconds to set up the runtime environment before execution, Derm's overall scaling overhead is relatively minimal.

IX. Related Work

Microservice Performance Modelling. GrandSLAm [17] computes the average execution time of each microservice as its performance profile. Erms [22] adopts a piece-wise linear function to profile the microservices latency in terms of load and resource interference. However, only the tail latency of each microservice can not quantify the distribution of E2E latency accurately. Moreover, ORION [23] characterizes serverless latency as a random distribution and profiles the latency performance under various resource configurations using ML. It needs to evaluate many possible configurations and therefore the scalability is quite limited.

Microservice Resource Scaling. There exist many works on resource management for microservices [5], [7], [8], [12], [14], [17], [22], [24]–[26], [28], [30], [31], [39], [40], [42], [43]. GrandSLAm [17] and Rhythm [43] quantify the contribution of each microservice to the E2E latency. However, they only consider simple graphs with sequential executions. DeepRest [8] incorporates a GNN-based architecture to conduct resource scaling. One limitation of these works is that they only consider a single type of microservice graphs. While Kraken [5] profiles dynamic invocations between serverless functions. The Markov model fails to support parallel calls, and its fixed transition probability matrix falls short in capturing the varying distributions of graphs. Nodens [34] is capable of extracting dynamic graphs during runtime and adjusting resource allocation to incrementally restore the desired QoS. However, this reactive approach to graph estimation and resource allocation can easily lead to SLA violations.

X. Conclusion and Remarks

Derm is the first attempt to optimize resource allocation for microservices with highly dynamic graphs through explicit modeling. By leveraging simple models including exponential fitting and linear regression, we are able to derive a closed-form expression for the E2E latency of microservice applications. These models can potentially be used for other applications.

While Derm is primarily focused on revealing the dynamic characteristics of microservices and employs cluster-level resource management to support diverse graphs with varying priorities, its insights are also applicable to architecture-level implementations. This facilitates the prioritization of memory controllers and last-level caches across different microservice processes. This approach addresses a gap in cluster-level resource management, which typically overlooks the hierarchical memory access intricacies of NUMA systems by merely allocating memory sizes to specific containers. Integrating this granular control over hardware resources with cluster-level management strategies can significantly boost application performance.

One limitation of Derm is that currently it does not allow changing the configuration of microservice containers when scaling resources. Exploring this in the context of more complex microservice graphs will be our future work.

XI. Acknowledgment

We sincerely thank the anonymous ISCA'24 reviewers for their valuable suggestions that improved the paper. This work is supported by the Science and Technology Development Fund of Macau (0024/2022/A1, 0071/2023/ITP2, 0081/2022/A2, and 0123/2022/AFJ), as well as the Multi-Year Research Grant of University of Macau (MYRG2022-00119-FST, MYRG-GRG2023-00019-FST-UMDF) and the 2024 Conference Grant of FST (CG-FST-2024).

References

[1] "Cncf:," https://www.cncf.io/.
[2] "Alibaba microservices cluster traces." https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021, 2021.
[3] "Jaeger," https://jaegertracing.io/, 2022.
[4] A. C. C. Apps, https://azure.microsoft.com/en-us/services/container-apps/, 2022.
[5] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proceedings of SoCC*, 2021.
[6] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.
[7] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of ASPLOS*, 2019.
[8] K.-H. Chow, U. Deshpande, S. Seshadri, and L. Liu, "Deeprest: Deep resource estimation for interactive microservices," in *Proceedings of EuroSys*, 2022.
[9] A. D. Crescenzo, V. Giorno, B. K. Kumar, and A. G. Nobile, "M/M/1 queue in two alternating environments and its heavy traffic approximation," *Journal of Mathematical Analysis and Applications*, 2021.
[10] A. C. M. Engine, https://www.alibabacloud.com/product/microservices-engine, 2022.
[11] G. K. Engine, https://cloud.google.com/kubernetes-engine, 2022.
[12] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical & scalable ml-driven performance debugging in microservices," in *Proceedings of ASPLOS*, 2021.
[13] Y. Gan, Y. Zhang *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of ASPLOS*, 2019.

[14] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *Proceedings of ICDCS*, 2019.
[15] L. A. Hageman and D. M. Young, *Applied iterative methods*. Courier Corporation, 2012.
[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[17] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, and J. Mars, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of Eurosys*, 2019.
[18] Kubernetes, https://kubernetes.io., 2022.
[19] M. Liang, Y. Gan, Y. Li, C. Torres, A. Dhanotia, M. Ketkar, and C. Delimitrou, "Ditto: End-to-end application cloning for networked cloud services," in *Proceedings of ASPLOS*, 2023.
[20] 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 ACM SoCC*, 2021.
[21] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C.-Z. Xu, "An in-depth study of microservice call graph and runtime performance." IEEE, 2022.
[22] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: Efficient resource management for shared microservices with sla guarantees," in *Proceedings of ASPLOS*, 2023.
[23] 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 *Proceedings of OSDI*, 2022.
[24] A. Mirhosseini, S. Elnikety, and T. F. Wenisch, "Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices," in *Proceedings of ACM SoCC*, 2021.
[25] A. Mirhosseini and T. F. Wenisch, "μsteal: A theory-backed framework for preemptive work and resource stealing in mixed-criticality microservices," in *Proceedings of ICS*, 2021.
[26] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch, "Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices," in *Proceedings of HPCA*, 2020.
[27] M. Newville, T. Stensitzki, D. B. Allen, M. Rawlik, A. Ingargiola, and A. Nelson, "Lmfit: Non-linear least-square minimization and curve-fitting for python," *Astrophysics Source Code Library*, 2016.
[28] J. Ortiz, B. Lee, M. Balazinska, J. Gehrke, and J. L. Hellerstein, "Slaorchestrator: Reducing the cost of performance SLAs for cloud data analytics," in *Proceedings of ATC*, 2018.
[29] J. Park, B. Choi, C. Lee, and D. Han, "Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices," in *Proceedings of ACM CoNext*, 2021.
[30] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proceedings of OSDI*, 2020.
[31] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," 2021.
[32] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, long short-term memory, fully connected deep neural networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 4580–4584.
[33] H. Saokar, S. Demetriou, N. Magerko, M. Kontorovich, J. Kirstein, M. Leibold, D. Skarlatos, H. Khandelwal, and C. Tang, "{ServiceRouter}: Hyperscale and minimal cost service mesh at meta," in *Proceedings of OSDI*, 2023.
[34] J. Shi, H. Zhang, Z. Tong, Q. Chen, K. Fu, and M. Guo, "Nodens: Enabling resource efficient and fast {QoS} recovery of dynamic microservice applications in datacenters," in *Proceedings of USENIX ATC*, 2023.
[35] 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.
[36] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for *oldi* microservices," in *Proceedings of OSDI*, 2018.
[37] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2016, pp. 1–10.
[38] K. Veeraraghavan, J. Meza *et al.*, "Kraken: leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services," in *OSDI*, 2016.
[39] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, "Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows," in *Proceedings of ICDCS*, 2019.
[40] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proceedings of ICWS*, 2019.
[41] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: Lstm cells and network architectures," *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.
[42] 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 ASPLOS*, 2021.
[43] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao, "Rhythm: component-distinguishable workload deployment in datacenters," in *Proceedings of EuroSys*, 2020.
[44] 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 ACM SoCC*, 2018.
[45] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of ICSE*, 2018.

Artifact Appendix

A. Abstract

We present Derm, a SLA-aware Resource Management for Highly Dynamic Microservices. To underscore the efficacy of Derm, we have made our artifact available on Zenodo, comprising the complete source code for our proposed implementation. This encompasses all components, such as the scheduler, and functionalities discussed in the paper, including the latency profiler, dynamic predictor, and optimization module. Additionally, users have access to training and profiling scripts, as well as a sample dataset from the TrainTicket benchmark, which can be effortlessly customized to suit their individual models and datasets. Example configuration and explanation can be found in our codes.

B. Hosting

The source code of Derm is publicly available on Zenodo: https://zenodo.org/records/10947677.

C. Hardware Configurations

We have built a prototype of Derm on top of Kubernetes, which is deployed in a private cluster consisting of 10 two-socket physical hosts. Each host is equipped with 52 CPU cores and 128 GB RAM. Each microservice container is configured with 0.2 cores and 1GB memory.

D. Software Dependencies

We list the software depedencies required by building up the Derm system in a cluster as follows:

- Python 3.11.4
- Matplotlib 3.6.0
- Pandas 1.5.0
- Numpy 1.23.4
- Seaborn 0.12.1
- Scikit-learn 1.2.0
- XgBoost 1.7.2
- Torch 1.13.1
- Kubernetes 1.23.2