Kuber: Cost-Efficient Microservice Deployment Planner

Harshavardhan Kadiyala
Univ. of British Columbia, Canada
devkhv129@ece.ubc.ca

Alberto Misail
Univ. of British Columbia, Canada
titomisa@student.ubc.ca

Julia Rubin
Univ. of British Columbia, Canada
mjulia@ece.ubc.ca

***Abstract*—The microservice-based architecture – a SOA-inspired principle of dividing backend systems into independently deployed components that communicate with each other using language-agnostic APIs – has gained increased popularity in industry. Realistic microservice-based applications contain hundreds of services deployed on a cloud. As cloud providers typically offer a variety of virtual machine (VM) types, each with its own hardware specification and cost, picking a proper cloud configuration for deploying all microservices in a way that satisfies performance targets while minimizing the deployment costs becomes challenging.**

Existing work focuses on identifying the best VM types for recurrent (mostly high-performance computing) jobs. Yet, identifying the best VM type for the myriad of all possible service combinations and further identifying the optimal subset of combinations that minimizes deployment cost is an intractable problem for applications with a large number of services. To address this problem, we propose an approach, called KUBER, which utilizes a set of strategies to efficiently sample the necessary subset of service combinations and VM types to explore. Comparing KUBER with baseline approaches shows that KUBER is able to find the best deployment with the lowest search cost.

I. INTRODUCTION

Microservice-based architecture is a SOA-inspired principle of building complex backend systems as a composition of small, loosely coupled components that communicate with each other using language-agnostic APIs [1]. This architectural principle is now becoming increasingly popular in industry due to its advantages, such as greater software development agility, elasticity, and a pay-per-consumption deployment model. Realistic microservice-based applications contain tens or even hundreds of services. Running them continuously in public clouds [2], [3] induces significant expenses [4], [3]. As cloud providers typically offer a variety of virtual machine (VM) types, each with its own hardware specification and cost, identifying a proper deployment configuration for microservice-based applications – one that satisfies performance targets while minimizing the deployment costs, becomes an important yet challenging release engineering task [5], [6], [7].

Existing work focused on identifying the cheapest yet performant VM types for recurrent (mostly, high-performance computing) jobs. This is typically done either by *prediction-based* approaches that estimate the execution time of a job on each target VM type based on pre-existing data [8], [9], [10], or by *sample-based* approaches which perform runtime sampling of job execution on a carefully selected subset of VMs and extrapolating this data on the remaining VMs [7], [11], [12], [13]. However, microservice-based applications bring additional complexity: it is not practical to explore all possible service combinations and then find the right subset of combinations that leads to the optimal deployment.

Consider, for example, the Sock Shop microservice-based demo application [14] in Figure 1, which contains seven services and is deployed on Amazon EC2 [15] – a cloud infrastructure offering more than 300 different VM types with a variety of hardware options [16]. Even though Amazon provides recommendations for how each VM type should be used and designated only 50 VM types for microservice-based applications, the space of all possible deployment configurations is still very large, e.g., the services *Cart*, *Catalogue*, *Shipping* could be combined and placed together on one VM, placed individually on three different VMs of the same type, or placed on three different VMs of three different types.

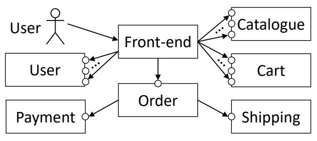


Fig. 1: A demo application: Sock Shop [14].

Finding the right VM type for each service combination depends on whether the services have competing CPU / memory / network requirements and on the capabilities of the particular VMs. A straightforward solution to this problem is to test all service combinations on all VM types. However, exploring the space of all combinations is exponential in the number of services: even for an application with seven services, there are 127 combinations, which quickly grows to millions for an application with only 20 microservices.

To further complicate matters, even if the performance of all service combinations on all possible VMs is known, finding the optimal subset of combinations that minimizes deployment cost is a non-trivial task by itself. In our example, services *Cart*, *Catalogue*, *Shipping* can be combined and placed together on a certain VM or can be further combined with services *Orders* and *Payment* and placed on a more expensive VM. In fact, this problem of finding the optimal deployment translates to the *weighted independent domination* (WID) problem, which is known to be NP-hard [17] and is thus intractable for applications with a large number of services.

In this paper, we propose a sample-based approach for addressing this problem, called KUBER, which relies on (a) a set of strategies for carefully selecting service combinations and VM types to sample and (b) a deployment mechanism to efficiently test the performance of the chosen combinations on a VM. We choose to follow a sample- rather than a prediction-based approach as our and others' experience, e.g., [7], shows that prediction-based approaches fail to accurately capture the correlation between workloads and VM capacity. We also confirm this claim in our evaluation.

KUBER performs an efficient search in the space of possible combinations and VM types by relying on three main insights:

1. The partial ordering of service combination allow KUBER to exclude a large number of VM types that will not meet the performance targets. For example, if a service combination {*Shipping*, *Orders*, *User*} does not meet the performance target on a certain VM type, any superset of this combination, e.g., {*Shipping*, *Orders*, *User*, *Carts*}, will not meet the target on that VM type either. KUBER thus implements logic for keeping and propagating prior execution results to future combinations.

2. Executing a service combination of a particular VM type is only worthwhile if the obtained solution has the potential to decrease the overall deployment cost. For example, it is not worthwhile to check whether the service combination {*Shipping*, *Orders*, *User*} meets a performance target on a VM that costs \$4 if it is already established that each individual service can work on a VM costing \$1 each. KUBER thus employs several strategies to efficiently narrow down the space of considered configurations.

3. The nature of our problem enables efficient improvements in existing heuristic approaches for solving the WID problem [18], scaling it to microservice-based applications of realistic size and complexity.

We evaluated KUBER on four open-source benchmark microservice-based applications, comparing it with baseline sample-based approaches which do not use combination/VM selection strategies and a prediction-based approach built on top of existing work. Our evaluation shows that KUBER outperforms the baseline approaches, finding the best deployment configuration faster and with the lowest search cost. Moreover, the differences between the approaches become more pronounced as the size of the applications grow.

Contributions. The paper makes the following contributions:

1. It formulates the problem of picking a proper cloud configuration for deploying microservice-based applications.

2. It proposes a sample-based approach for addressing this problem, implemented in a tool called KUBER. KUBER consists of a set of strategies for minimizing the number of runtime experiments, an efficient solution for collecting performance data at runtime, and a problem-domain-inspired approach for improving the scalability of an existing heuristic WID solution, so it can be applied to applications of realistic size and complexity.

3. It evaluates the effectiveness of KUBER on four case-studies, comparing it with a number of baseline approaches.

4. It makes our implementation and evaluation setup publicly available to facilitate replication and further research [19].

II. BACKGROUND

In this section, we provide a short overview of microservice-based application development and deployment.

A. Microservice-based Applications

Microservice-based architectures are closely related to service-oriented architectures (SOA), which is a style of software design where services represent application components that communicate over a network [20]. Microservices aim at shortening the development lifecycle while improving the quality, availability, and scalability of applications at runtime. Another major advantage of microservice-based architectures is independent deployment, which reduces the coordination effort needed to align on common application delivery cycles and also leads to independent scaling at runtime [21].

A microservice-based development style is often used for in latency-critical applications, such as user-facing websites, where decreased performance leads to decreased user satisfaction and loss of business [22]. In such cases, it is common for developers to specify performance targets as part of their applications' service-level agreement (SLA) – a commitment between a service provider and a client. Performance targets are usually evaluated on a p-th percentile (e.g., the 99th or 95th percentile) of all requests the service receives [23].

B. Cloud Infrastructure

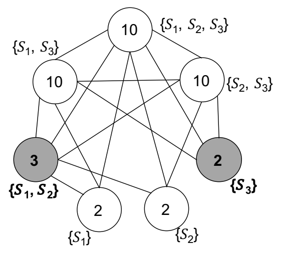
Cloud providers, such as Amazon AWS [16] and Microsoft Azure [24], offer customers compute resources running on the providers' physical infrastructure. Specifically, they provide a wide range of Virtual Machine (VM) types, which differ in the processor architecture (e.g., Intel vs. AMD, CPU vs. GPU) and size (e.g., 2 vs. 16 CPU cores). VM types are grouped into families; VMs in a family typically have the same underlying architecture and differ by their size. At the time of writing, AWS provides more than 300 VM types grouped into more than 40 families [16]; Azure provides more than 400 VM types grouped into more than 50 families [25].

To increase utilization and achieve better cost-efficiency, multiple VMs are typically hosted on the same physical machine and thus share CPU, caches, memory, storage, and networking devices. While cloud providers guarantee certain resources, such as CPU, memory capacity, and storage, by dedicating them to a particular VM, other resources are shared by VMs running on the same physical machine [26]. Increased load on a physical machine might cause *VM interference*, which results in performance degradation for applications running on these VMs [27], [28].

Moreover, developers have the option to co-locate multiple workloads/microservices on the same VM. OS-level virtualization solutions, such as Docker containers [29], help enable co-location of microservices by providing fault and dependency isolation, thereby preventing failures in one service from propagating to others. As containers do not guarantee performance isolation between workloads, when services running on the same VM rely on a particular shared resource, they face *service-level interference* [30].

Combin.	VM Type: Price ($)
$\{S_1\}$	VM_1: 2
$\{S_2\}$	VM_1: 2
$\{S_3\}$	VM_1: 2
$\{S_1, S_2\}$	VM_2: 3
$\{S_1, S_3\}$	VM_3: 10
$\{S_2, S_3\}$	VM_3: 10
$\{S_1, S_2, S_3\}$	VM_3: 10

(a) Input data



(b) WID Graph and Solution

Fig. 2: Weighed Independent Domination (WID).

Unlike the case of VM-level interference, no guarantees on the performance of interfering services are available and it is up to the development team to decide which co-locations of microservices are desirable given the performance targets. If a shared resource is heavily utilized by co-located services, none of the services gets all the resources they need to meet the required performance target.

III. Approach

We now discuss our approach for finding a desired deployment configuration for a microservice-based application.

A. Problem Statement

We assume as input an application $\mathbb{S}$ with n services S_1, ..., S_n, where each service S_i has i_j APIs, denoted by S_i:A_1, ..., S_i:A_{i_j}. We say that each API S_i:A_j has a performance target (e.g., measured in terms of time to process a request); we denote the performance target of S_i:A_j by S_i:A_j^t.

We also assume as input a compute cluster $\mathbb{VM}$ with m VM types VM_1, ..., VM_m, where each VM type has its own hardware specification and cost; we denote the cost for VM_i by $VM_i{}^c$. We say that a service combination π, formed by colocating a subset of $\mathbb{S}$ on the same VM, satisfies the performance target if the performance targets of all APIs of all services in π are satisfied on that VM.

Our goal is to find a deployment configuration Λ for $\mathbb{S}$, which maps service combinations of $\mathbb{S}$ to target VM types, such that: (a) every $S_i \in \mathbb{S}$ is part of exactly one service combination π in Λ; (b) every service combination π in Λ is mapped to a VM on which the performance target of π is satisfied; and (c) there is no other configuration Λ' such that the total cost of all VM types in Λ' is lower than in Λ.

That is, we aim at finding the cheapest deployment configuration that can co-locate multiple services on the same VM and that satisfies the performance targets of all services in $\mathbb{S}$.

B. A First-Approximation Solution

Our experience and prior work show that there is no direct correlation between the cost and the performance of a service on a VM [8]. The most obvious solution to the problem of finding cheapest deployment configuration is thus to first order all VM types in $\mathbb{VM}$ by their cost and then run each service combination in $\mathbb{S}$ on each VM type one by one, until the cheapest VM type for each service combination is found. We refer to this solution as *Sort and Find* (SF).

Once the best VM type for each service combination is determined, we need to identify a subset of combinations that satisfy the conditions above. More formally, given a mapping of service combinations to the cheapest VM type for which each service combination satisfies its performance target, we need to find the subset of combinations that includes each service once and only once and has the lowest possible deployment cost. Consider, for example, an application with only three services, S_1, S_2, and S_3, which is deployed on a cluster with three VM types, VM_1, VM_2, VM_3. Let us assume that the cost of these VM types are 2, 3, and 10, respectively. There are seven possible service combinations. For illustration purposes, Figure 2a shows, for each combination, the cost of the cheapest VM type for which the performance target is satisfied. Multiple deployment options are possible for this example: each of the services could be deployed individually on different instances of VM_1; the overall cost of this solution would be 6. A cheaper deployment would be to deploy the combination $\{S_1, S_2\}$ on VM_2 and $\{S_3\}$ on VM_1; the cost of this solution would be 5. A deployment that contains service combinations $\{S_1, S_2\}$ and $\{S_1, S_3\}$ would be invalid as S_1 would be deployed more than once. Similarly, a deployment that contains the service combination $\{S_1, S_2\}$ only would be invalid as S_3 would not be deployed.

The problem of finding the cheapest valid deployment given the mapping from a service combination to its cheapest working VM type (like in Figure 2a) can be translated into the Weighed Independent Domination (WID) problem [17]. The input to WID is a weighted undirected graph $G = (V, E)$, where nodes $v \in V$ and edges $e = (v, u) \in E$ have non-negative weights $w(v) \geq 0$ and $w(v, u) \geq 0$, respectively. WID then finds a subset of nodes $D \subseteq V$ which satisfy the following criteria: (1) *Independent*: no two nodes in D are adjacent. (2) *Dominant*: any node in V is either in D or adjacent to a node in D. (3) *Least Weight*: D minimizes the following cost function: $f(D) = \sum_{u \in D} w(u) + \sum_{v \in V \setminus D} min\{w(v, u),$ for $u \in D$ and $(v, u) \in E\}$, which is the sum of the weights of the nodes in D plus the sum of the weights of the minimum-weight edges connecting nodes in $V \setminus D$ to nodes in D.

To rephrase the deployment detection problem as WID, we define V to be the set of all possible service combinations. We place an edge between a pair of nodes in V iff their corresponding combinations have at least one service in common. We set the node weights to be the cost of the cheapest VM type on which the combinations meet their performance target. We do not use edge weights and thus set them all to 0. Figure 2b shows such graph for the mapping in Figure 2a.

A solution D produced by the WID algorithm (highlighted in grey in the example in Figure 2a) results in a cheapest valid deployment Λ: (1) By the *Independent* property, every $S_i \in \mathbb{S}$ is part of at most one service combination in Λ because if a service is part of a combination that was chosen in D, no other service combination that contains the service is in D. (2) By the *Dominant* property, every $S_i \in \mathbb{S}$ is part of at least one service combination in Λ because the service combination

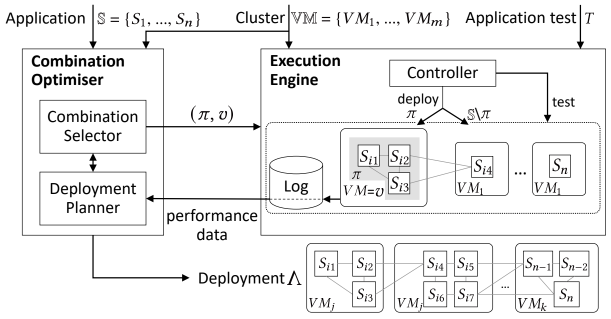


Fig. 3: KUBER overview.

$\{S_i\}$ is either in D or one of its adjacent nodes (that also contain S_i) is in D. (3) By the *Least Weight* property, there is no other Λ' such that the total cost of all VM types in Λ' is lower than in Λ because WID's cost function translates to the sum of the weights of all nodes in D, i.e., their deployment cost. Thus, minimizing this function means that no other valid deployment with lower costs exists.

As WID is an NP-hard problem [17], we rely on an iterative greedy algorithm by Davidson et al., which approximates the result and was shown to outperform existing work [18]. In a nutshell, the algorithm starts from greedily selecting an initial set of nodes in D and then iteratively improves the initial result by taking a random subset of nodes out of D (partial destruction phase) and greedily completing it to form a valid solution again (reconstruction phase).

Davidson et al. evaluated their approach on randomly generated graphs of varying sizes (between 100 and 1000 nodes). Yet, our graphs tend to be substantially larger (more than a million service combinations for an app with 20 services) and much more dense (as many nodes share common services); we thus modify and adapt this approach to our scenario, as discussed in Section III-C.

C. Kuber Solution

An overview of KUBER, which further extends the approach outlined in the previous section, is shown in Figure 3. KUBER consists of two main parts. The first part, *Combination Optimizer*, improves the SF solution with a more efficient selection of service combinations to explore. The second part, *Execution Engine* takes care of service deployment and runtime data collection. We now discuss these two components in detail.

Combination Optimizer. The SF solution performs a runtime experiment for every non-empty subset of $\mathbb{S}$, i.e., $\mathbb{P}(\mathbb{S})$-1 times. Such runtime experiments are costly, with respect to both time and budget. To reduce this cost, *Combination Optimizer* relies on a number of strategies, summarized in Algorithm 1.

It first initializes a set of variables: the map M, which keeps, for each service combination, the cheapest VM type where the performance target of the combination is satisfied (line 3); Λ, which keeps the best deployment configuration identified so far (line 4); and Λ^c, which keeps the cost of that deployment (line 5). The algorithm then computes the set of all non-empty service combinations of the input application $\mathbb{S}$ and sorts them by the number of services in a combination, i.e., first the combinations with one service, then combinations with two services, etc. (line 6). It iterates over all combinations in order and, for each combination, explores all VM types in order (lines 7-30). Before collecting performance data for each combination π on a VM v, it checks that the following conditions hold:

Condition 1 (lines 10-12): If the cheapest working VM type for at least one subset of services $\overline{\pi} \subset \pi$ is more expensive than v, that implies that the performance target of $\overline{\pi}$ was not met on v. As adding more services to $\overline{\pi}$ cannot improve the performance of services that are already in that set, π cannot meet its performance target on that VM type either, and this runtime experiment can be skipped altogether. For example, the algorithm will skip executing the service combination $\{S_1, S_2, S_3\}$ on VM_i if a subset of services, say $\{S_1, S_2\}$, does not meet the performance target on that VM type.

Condition 2 (lines 13-17): If executing π cannot lead to a deployment that is cheaper than the current solution, executing the experiment is unnecessary and can be skipped as well. To estimate whether the experiment has a chance to improve the cost of the current solution, we conservatively assume that still unexplored combinations have a chance to meet their performance target on certain VM types. More specifically, for each still unexplored combination, we utilize our knowledge about best VM types selected for its subset combinations (if any) and optimistically assume that the target combination will work on the most expensive of those VM types (lines 33-40).

Like in the previous case, we leverage the idea to order all explored combinations by size, making sure smaller combinations are executed earlier and their performance data can be propagated to larger combinations. Moreover, we conservatively pick the cheapest possible VM type (or VM_1 for combinations of one service) to ensure we do not skip any experiments that have a chance to lead to a better deployment placement in the future. For example, if S_1 and S_2 meet their performance targets on VM_2 and VM_4, respectively, we optimistically assume that a still unexplored combination $\{S_1, S_2\}$ will meet its performance target on VM_4.

We rely on the *Deployment Planner* component to decide whether an experiment is worthwhile to execute. It accepts as input a map M (from a combination to its best VM type) and an experiment of interest m; it calculates the deployment solution using our extended version of the WID algorithm (described below) or returns $\emptyset$ if at least one of the services does not have any VM type mapped to it yet. When m is given, the method ensures that m is part of the produced solution. Otherwise, it returns any solution for the given map of combinations.

To decide whether to execute an experiment (π, v), we pass to the *Deployment Planner* a map containing all previously explored and optimistically projected service combinations, as well as the experiment of interest (line 14). We only proceed to actually executing the experiment if placing π on v could indeed lead to a cheaper deployment that includes this placement. We continue to the next combination otherwise, as placing π on even a more expensive VM type cannot further

```
1  Input: Application $\mathbb{S}$ = $\{S_1, ..., S_n\}$,
   Cluster $\mathbb{VM}$ = $\{VM_1, ..., VM_m\}$ (ascending order by VM cost)
   Output: Deployment $\Lambda$
2  begin
3      $M \leftarrow \emptyset$                                ▷ A map of combination ⇝ best VM type
4      $\Lambda \leftarrow \emptyset$                                ▷ No solution yet
5      $\Lambda^c \leftarrow \infty$                               ▷ Upper bound for current solution cost
6      $\Pi \leftarrow \mathbb{P}(\mathbb{S}) \setminus \emptyset$   ▷ All non-empty combinations of services in $\mathbb{S}$, arranged by the
         number of services in a combination
7      while $\pi \in \Pi$ do
8          $\pi = popFirst(\Pi)$          ▷ Fetch and remove the first combination in $\Pi$
9          foreach $v \in \mathbb{VM}$ do
10             if $\exists \bar{\pi} \subset \pi$ such that $M(\bar{\pi}) = \bar{v} \wedge \bar{v}^c > v^c$ then
                   ▷ Condition 1: One of the subsets of $\pi$ did not meet the performance
                   target on $v$, hence $\pi$ cannot meet the performance target on $v \Rightarrow$
                   proceed to the next VM type
11                 continue
12             end
13             $M' = OptimisticGuess(\Pi, M)$   ▷ Optimistically find the best
                 possible VM type for unexplored combinations
               ▷ Deployment under this assumption
14             $\Lambda' \leftarrow DeploymentPlanner(M \cup M', (\pi, v))$
15             if $cost(\Lambda') \geq \Lambda^c$ then
16                 break          ▷ Condition 2: Solution does not lead to a better
                     deployment $\Rightarrow$ explore next combination.
17             end
18             $execute(\pi, v)$                         ▷ Collect runtime performance data
19             if performance targets of $\pi$ is satisfied on $v$ then
20                 $M[\pi] \leftarrow v$                      ▷ This is the cheapest VM type for $\pi$
21                 $\Lambda \leftarrow DeploymentPlanner(M, \emptyset)$            ▷ current best
22                 $\Lambda^c \leftarrow cost(\Lambda)$                              ▷ current best cost
23                 $\Lambda' \leftarrow DeploymentPlanner(M \cup M', \emptyset)$
24                 if $cost(\Lambda') \not< \Lambda^c$ then
25                     return $\Lambda$          ▷ Condition 3: No better solution is possible
26                 end
                   ▷ The cheapest VM type for $\pi$ is found $\Rightarrow$ explore next combination
27                 break
28             end
29         end
30     end
31     return $\Lambda$
32 end
33 Procedure OptimisticGuess($\Pi$, $M$)
34     begin
35         $M' \leftarrow \emptyset$
36         foreach $\pi \in \Pi$ do
37             $M'[\pi] \leftarrow$ the most expensive VM type of all subsets of $\pi$
                 in $M$ or $VM_1$ if non of the subsets is in $M$
38         end
39         return $M'$
40     end
```

Algorithm 1: Combination Optimizer.

improve the cost (lines 15-17).

If placing π on v has the potential to lead to a better solution, we proceed to executing the experiment and collecting real performance data (line 18). For combinations that satisfy the performance target on the given VM type, we update the combination to best VM type map (line 20) and then rely on the *Deployment Planner* again to calculate the best current solution and its cost (lines 21-22). This time, we only pass M as the parameter as we are interested in the best possible realistic solution rather than a solution that contains (π, v) or that relies on predicted data.

Condition 3 (lines 23-26): Finally, when a combination π can successfully run on a VM type v, the algorithm checks whether any further improvements are still possible. To this end, it uses the *Deployment Planner* again, this time passing it the map containing both executed and projected combinations (line 23). If no solution that can improve the cost of current deployment (with or without the executed combination π) is possible, the algorithm terminates and returns the current result (lines 24-26). Otherwise, it proceeds to exploring the next combination in order (line 27), as the cheapest VM type for this combination is already identified.

Deployment Planner. As discussed in Section III-B, we build up on the algorithm by Davidson et al. [18] for heuristically solving the WID problem. When computing a solution (in both initial and reconstruction phases), this algorithm iteratively and greedily chooses the next node to be one that has the highest ratio between the number of edges to remaining candidate nodes and the weight of the node. The rationale for this decision is to choose a dominant node (one that has a large number of edges) with a low weight. For the example in Figure 2b, the first node picked would be $\{S_1, S_2\}$ as it has five edges to the neighbor nodes and a weight of 2, giving a ratio of 2.5 – larger than that of any other node. Then, the selected node and all its neighbors are removed from the set of possible candidates, to satisfy the *Independent* property. For the example in Figure 2b, that would remove all but the node $\{S_3\}$; that node is selected next to complete the solution.

This algorithm does not scale well for inputs of our size. E.g., for apps with 20 services, the number of nodes would be more than a million and it will contain more than half a trillion edges; storing this information explicitly is not possible at this scale. Our main observation is that our graph has a very particular structure – its nodes are the service combinations and edges represent a partial order over the set of combinations. To choose the next node in every iteration, we mainly need to know the number of other candidate nodes a node is connected to. Moreover, when a particular node is selected, we only need to compute and remove from the set of future candidates all other nodes it is connected to. Using our knowledge about the graph structure, we adapt the algorithm by Davidson et al. [18] to compute this information on-demand, without explicitly storing the underlying graph, thus improving the algorithm's scalability.

Assuming that a certain number of nodes has already been selected to be part of the solution, let R be a subset of services that have not yet been included in any of these nodes. Let r be the number of these services, i.e., $r = |R|$. The number of nodes remaining for selection is then $2^r - 1$. Let v be a candidate node; it can have edges to at most all still unselected nodes composed from services in R but itself: $2^r - 1 - 1$.

To calculate the exact number of neighbors of v, we consider the services it contains. Assuming there are r' such services, there are $r - r'$ services in R that are not part of v and there are $2^{r-r'} - 1$ nodes composed from these services. As two nodes have an edge only if they share at least one service, v has no edges connecting it to any of these nodes. As such, v has $(2^r - 1 - 1) - (2^{r-r'} - 1) = 2^r - 2^{r-r'} - 1$ edges.

We use this formula to calculate the number of edges for each remaining candidate node and pick the one with the maximal ratio. In Figure 2b, when the algorithm starts,

$R = \{S_1, S_2, S_3\}$ and $r = 3$. For the node $\{S_1, S_2\}$, $r' = 2$, thus, the number of neighbors is $2^3 - 2^{3-2} - 1 = 5$.

After a node is selected, we compute the remaining candidates by leveraging the fact that nodes are adjacent only if they share at least one service. Thus, the remaining candidates are nodes that do not share any service with the selected node v, i.e., the power set of all services in R minus the services in v. In our example, when $\{S_1, S_2\}$ is selected, the remaining nodes are formed by all the combination of S_3, which is the combination $\{S_3\}$ itself.

Execution Engine. This component is responsible for performing the runtime experiments and collecting the performance data for each service combination π on a VM type v (line 18 in Algorithm 1). To accurately collect such data, we must deploy π on v in isolation. Yet, services in π interact with the rest of the system, i.e., $\mathbb{S} \setminus \pi$. To ensure the performance of the services in π is not negatively affected by "lagging" services of the rest of the system, we deploy each remaining service in $\mathbb{S} \setminus \pi$ in isolation, on a separate instance of the least expensive VM type (VM_1). The *Controller* component in Figure 3 takes care of such deployment.

We assume as input a set of tests that exercise the input application. To collect response times of APIs of the services in π, *Controller* executes these tests and, for each API of a service in π, captures the incoming request and the response times. Since the response time of an API depends not only on its own execution time but also on the response times of the outbound service it triggers, we measure and subtract the response times of such calls, as was also done in earlier work [31]. For the example in Figure 1, when measuring the response time of API of the Order service, we subtract from its execution time the response times of the outbound calls to the Payment and Shipping services.

D. Implementation

We use a private cluster with three physical machines. Two of the machines have an Intel Xeon E5-2640 v4 @ 2.40GHz processor with 40 cores, 128 GB of RAM, 25 MB cache, and 63 GB/s Memory bandwidth. The third machine has an Intel Xeon E5-2680 v4 @ 2.40GHz processor with 56 cores, 256 GB of RAM, 35 MB Cache, 76 GB/s Memory bandwidth.

We create and manage VMs using the OpenNebula cloud computing platform [32] deployed on a separate machine. We use Kubernetes cluster manager [33] and Istio monitoring system [34] to deploy and monitor microservices. We use Istio's logging functionality to store the execution time of each API in a time series database.

Finally, our implementation of the *Combination Optimizer* and *Execution Engine* components is written in Python and takes around 5000 lines of code. Our system implementation is publicly available to facilitate further research in this area [19].

IV. Evaluation Setup

The goal of our evaluation is to answer the following research questions:

TABLE I: VM Types

VM Type	AWS VM Type	CPU Cores	RAM (GB)	US$ / Hour
VM_1	A1.medium	1	2	0.0255
VM_2	M6g.medium	1	4	0.0385
VM_3	A1.large	2	4	0.051
VM_4	M6g.large	2	8	0.077
VM_5	A1.xlarge	4	8	0.102
VM_6	T3.micro	2	1	0.1104
VM_7	T3.small	2	2	0.1208
VM_8	M6g.xlarge	4	16	0.154
VM_9	T3.large	2	8	0.1832
VM_{10}	A1.2xlarge	8	16	0.204
VM_{11}	M6g.2xlarge	8	32	0.308

RQ1 (Selection Strategies): How effective are the configuration and VM selection strategies applied by KUBER?

RQ2 (Sampling vs. Prediction): How effective is KUBER when compared with a baseline prediction-based approach?

We now discuss our experimental setup, including our selection of VM types, subject applications, and baseline approaches for comparison. To facilitate reproducibility, our experimental package is available online [19].

VM Types. We used the three physical machines in our private cluster to simulate a number of VM types from Amazon EC2. Specifically, we choose three different families of VMs suggested for microservice-based applications: the basic A1 family, which provide cost savings for CPU-intensive workloads; the more expensive T3 family, which provides burstable general-purpose instances, thus increasing the price of each VM type; and the higher-performance M6g family.

We picked four VM types from each family (12 VMs in total), starting from a VM type on which all services of our subject applications can boot and run individual. That excluded the smallest VM type from the T3 family: T3.nano with only 0.5 GB of RAM. We could not simultaneously simulate two of the selected VM types in our cluster: T3.medium and A1.large, because they both have 2 CPU cores and 4 GB of RAM. We thus excluded T3.medium from our analysis. The resulting 11 VM types, together with their mapping to the corresponding Amazon EC2 instance, the number of CPU cores, RAM size, and the cost per hour (as of January 2020) are given in Table I.

We deployed all the VMs corresponding to the same VM type family onto the same physical machine, allocating our largest physical machine (56 cores, 256 GB of RAM) to the M6g family and the remaining two machines (40 cores, 128 GB of RAM) to the T3 and A1 families. The obtained size and the capacity of our cluster is similar to prior work [7].

Subject Applications. We used a recent benchmark of microservice-based applications, called DeathStarBench [31]. It consists of three applications: Hotel Reservation, Media Service, and Social Network. In addition, we used a popular open-source microservices demo application called Sock Shop [14]. We selected these applications because they are explicitly designed to represent real-world systems, are deployable onto a Kubernetes cluster, and include test suites allowing us to effectively trigger services/APIs.

TABLE II: Subject Applications

Benchmark	#Services	#APIs	Avg. #APIs/Service
Hotel Reservation	8	14	2
Media Service	11	29	3
Social Network	12	27	2
Sock Shop	7	42	6

Table II shows, for each application, the number of services it contains, the total number of APIs, and the average number of APIs per service. Overall, our applications contain between 7 and 12 services, with 14 to 42 APIs in total, and 2 to 6 APIs per service, on average. As the performance of an application varies based on the number of requests it receives (the API load provided by the test) and the volume of data stored in its associated database(s), we applied the following strategy to populate applications with realistic data.

For Hotel Reservation, which allows the users to obtain information and rates of nearby hotels, check hotels' availability during a given time period, make reservations, and also obtaining recommendations for hotels matching their selection criteria, we populate the hotel information database with real-world data from Yelp's Hotels Dataframe [35]. It contains 438 hotels and 172,159 hotel reviews. Similarly, for Media Service, which allows users to browse movie information, and then rent, stream, review, and rate movies, we use data from a real movies database, TMDB [36], which contains information about 5,000 movies and 5,000 casts.

Similar to Twitter, in the Social Network application, users can create posts embedded with text, media, and links, can tag users, and broadcast posts to their followers. The application uses three separate databases for persisting user profiles, posts, and media. We load the profiles database from existing social network data [37] with 962 users and 18,800 relations (representing followers). The volume of posts and media databases does not affect the performance of the application and we thus only use them for data generated at runtime.

Finally, for Sock Shop, an e-commerce application allowing users to browse and buy socks, we first searched for all socks sold by Amazon [38]. We learned that Amazon sells around 40,000 types of socks at the time of writing; we thus loaded the database with the same number of items.

Each of our subject applications contains a test suite provided by the developers, which simulates its typical usage scenario. For example, the test suite of Hotel Reservation simulates the scenario where the user logs in into the application, searches for a hotel, gets hotel recommendations, and reserves a hotel. We set the number of concurrent users served by each application to 165, as specified by DeathStarBench. We define a workload for an application as a set of API calls made by concurrent users under the test.

Performance Targets. We use API execution time to represent API performance, with high performance translating to low execution time. To set the performance target for an API, we follow existing work that typically selects targets within a certain percentage of the best possible performance [39]. We thus assumed that the largest VM type (in all dimensions) has the best performance [40] and, without loss of generality, set the targets to be 50% of that performance. That is, we set the individual API performance targets to be twice their execution time on VM_{11}. Such selection ensures that performance target can be reached on some but not all VM types.

Runtime Environment. Given 165 concurrent users and our database load, all user requests terminate within two minutes of execution on any VM type. We thus picked two minutes execution time for each test. Deploying and booting services on the right configuration of VMs takes another five minutes. We reset the VMs and repeat each experiment three times, to avoid performance variability due to underlying infrastructure. Thus, the total execution time of each experiment is 21 minutes. To avoid any performance bottlenecks, we make sure to deploy the test scripts and all external dependencies of each microservice, including databases.

As the WID algorithm is evolutionary, it requires a time limit to stop performing iterations. We experimented with the algorithm, running it on the largest set of combinations in our subjects set for 30 minutes. Our experiments showed that the best solution is achieved within one minute and minimal to no improvements are achieved afterwards. We thus set one minute as a time limit for the algorithm.

Baseline Approaches. To answer **RQ1**, we implemented the basic SF approach described in Section III-B. We then augmented it with each of the three conditions described in Section III-C one-by-one, producing three different implementations, which we refer to as SF_1, SF_2, and SF_3. We compared these approaches with KUBER, which uses a combination of all three conditions simultaneously.

To answer **RQ2**, we implemented an approach that borrows and adapts ideas from a prominent prediction-based approach, PARIS [8], making it work in the microservices context. The goal of PARIS is to predict the performance of a service on a VM type. It does so by profiling a set of benchmarks that are assumed to be similar to the real applications of interest. For each benchmark, PARIS collects resource utilization (e.g., CPU usage) and performance information on all VM types, scaling it relatively to a few reference VM types (typically two). Then, to predict the performance of a service, PARIS collects features of the service by running it on the reference VM types and uses an ML-based model to predict performance on the remaining VM types based on the service similarity with the benchmarks.

To directly apply PARIS for predicting the best VM type for a service combination, we would need profiling information from all various combinations of benchmark services, which is untenable in our setting. We thus use individual benchmarks to predict the performance of combinations. To fairly evaluate the prediction properties of the approach, without relying on our ability to chose benchmarks similar to services in our dataset, we opted to use individual services themselves as benchmarks to train the model. That is, we run single services in isolation on each VM type and collect resource utilization and performance data. We use this data to train an ML-based model similar to the one used by PARIS. Then, to predict the performance of a combination of services π, we execute π on only two VM types and use the model to predict the performance on the remaining VM types. To validate the prediction, we execute a runtime experiment on the cheapest VM where π is predicted to work and continue to the next predicted VM type, if the performance target is not met. Further, we apply conditions 1-3 to make sure we do not disadvantage this approach when comparing it with KUBER. We refer to the obtain approach as **P (for prediction)**.

Measures and Metrics. For each approach, we calculate the cost of the deployment configuration Λ it finds. While AWS prices VMs per hour, microservice-based applications run for several days, months, or even years. Thus, without loss of generality, we calculate the cost of deployment per month. That is, when comparing the deployment cost found by the approaches, we multiply the hourly cost of each VM type in Λ by 24 hours and 30 days.

As the quality of the solution identified by each of the approaches improves as a function of the number of experiments it performs, we calculate the deployment cost identified by each approach as functions of: (1) the *search cost*, which represents the amount of money (in US dollars) spent in finding a solution and includes the cost of VMs used during the experiments, and (2) the *total execution time*, which represents the time (in hours) taken by an approach and includes the time of runtime experiments and WID execution.

V. RESULTS

Figures 4a-4d show, for each subject, the deployment cost achieved by each of the evaluated approaches as a function of the invested search cost. The baseline for the graphs, i.e., point x=0, is a deployment that places each individual service on the most expensive VM type. We do not depict this solution in the figure to avoid clutter, starting from the point where each approach found the cheapest working VM for each individual service. For example, for the Sock Shop application in Figure 4d, the cost of such deployment is \$312 and it takes \$2 to find this solution.

We mark with a cross the point on each graph where the corresponding approach terminates and we list the (search cost, deployment cost) values at this point, for clarity. E.g., for the Sock Shop application, KUBER terminated after spending \$6, identifying a deployment that costs \$238. SF_1 spends \$13 to find the same deployment, and SF_3 spends \$27. While SF found the same deployment after spending \$27, this approach continues to run and explore additional combinations. We stopped approaches which take substantially longer to terminate and do not show their termination points in the figure. Figures 4e-4h show similar information: the deployment cost achieved by each approach as a function of its execution time.

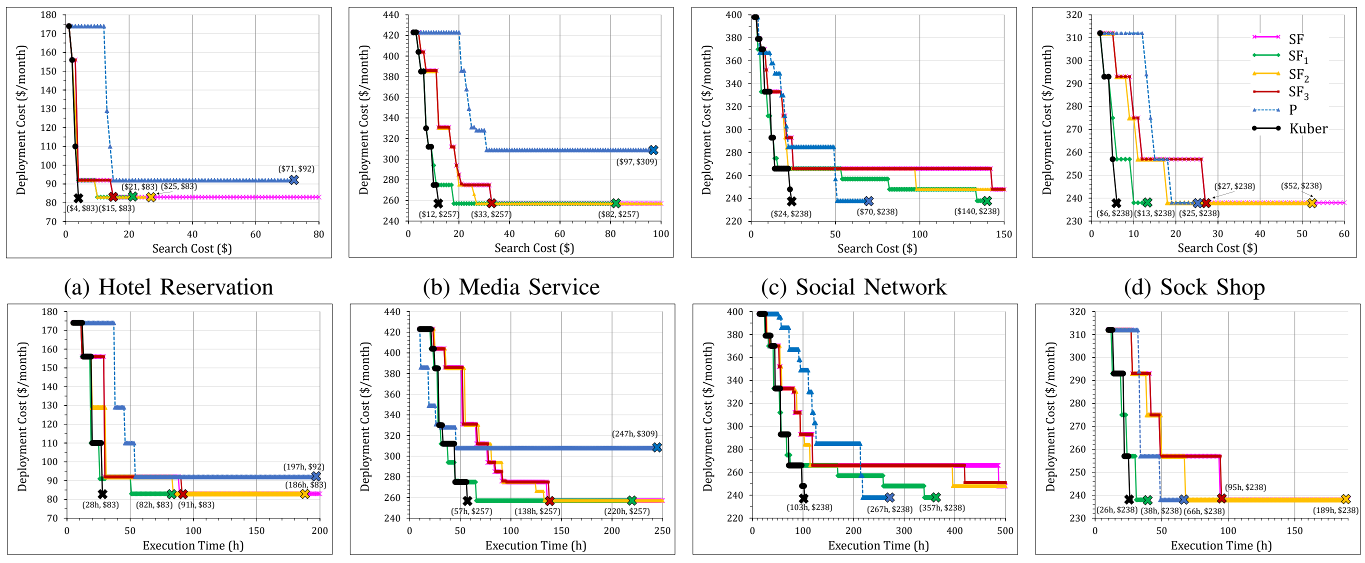
RQ1 (Selection Strategies). All approaches evaluated in this research question perform exhaustive search over the space of combinations. Thus, given enough time and budget, they all arrive at the optimal solution. Yet, comparing KUBER with SF_1, SF_2, SF_3, and SF shows that the combination of conditions that KUBER employs is the most beneficial for finding lowest-cost deployment at minimal search cost and execution time: KUBER spends \$12 on average (min: 4, max: 24) and runs for 54 hours on average (min: 26, max: 103). In comparison, SF_1 spends \$64 on an average (min: 13, max: 140) and runs for 174 hours on average (min: 38, max: 357); SF_2 spends more than \$94 on an average (min: 25, max: >150); SF_3 spends more than \$57 on an average (min: 15, max >150); and SF – more than \$144 on an average (min: 126 for Sock Shop, not shown in the figure to avoid clutter, max >150). These three approaches also execute for hundreds hours on average. In fact, the total execution time of all the experiments is more than four months. A detailed breakdown of time spent by each approach in each of the phases (setting up VMs for the experiments, executing the experiments, and running the WIP algorithm) is available online [19].

The differences between the approaches are more pronounced as the size of the applications grows. For example, for Sock Shop, which is the smallest subject application with only seven services, the search cost of KUBER is 53% lower than that of its closest competitor, SF_1; for Social Network, the largest application with 12 services, the difference is 82%. Similarly, the execution time of KUBER is lower than that of SF_1 by 32% for Sock Shop and by 71% for Social Network.

Our experiments show that without any termination condition, SF continues executing experiments that do not improve the overall deployment cost, even if it arrives at the optimal solution, like in the case of Hotel Reservation, Media Service, and Sock Shop applications. SF_3 mitigates this issue by inducing a stopping condition (Condition 3 in Section III-C) when no better solution is possible, which, in fact, stopped the executing in all these cases. For the Social Network application, the largest in our dataset, SF does not reach the optimal solution within the allocated time. Even though this application is larger than Media Service by only one service, it has double the number of combinations (4096 vs. 2048 combinations for Social Network and Media Service, respectively), which increases search cost and execution time.

SF_1 is the only approach besides KUBER that reaches the optimal solution for Social Network, demonstrating that the pruning technique preventing SF_1 from running combinations that are expected to fail (because their subset already failed on the same VM type; Condition 1) is the most effective strategy to reduce the number of unnecessary experiments.

SF_2 (Condition 2), by itself, performs worse than SF_1, but it helps eliminate a small number of experiments on very costly VM types while SF_1 eliminates a large number of lower-cost non-working VM types via propagation of negative results. For example, the cheapest VM type for which the *Order* service in Sock Shop can meet its performance target is VM_{10}; the compilation of *Payment* and *User* services can work on VM_1; and the combination of all three services together does not meet the performance target on any of the given VM types. For that combination, SF_1 will not perform runtime experiments for VM_1-VM_9 and will check VM_{10} and VM_{11}. While SF_2 will execute experiments on VM_1-



(a) Hotel Reservation (b) Media Service (c) Social Network (d) Sock Shop

(e) Hotel Reservation (f) Media Service (g) Social Network (h) Sock Shop

Fig. 4: Search cost (a-d) and execution time (e-h) comparison.

VM_{10}, it will determine that placing the combination of these three services on VM_{11} (which costs \$0.308 per hour) is more expensive than placing them on VM_1 and VM_{10} separately (\$0.0255+\$0.204=\$0.2295) and will eliminate this experiment.

Interestingly, the search cost for SF_1 is lower than that of SF_3 for the Social Network and Sock Shop applications but is higher than SF_3 for Hotel Reservation and Media Service. That is because in Social Network and Sock Shop, there are a few highly interfering services. Placing them on the same VM would require an expensive VM type to ensure they meet their performance target. In fact, the optimal solution for both of these applications involves a combination of three services placed on VM_{10}. SF_1 thus has an advantage due to its ability to skip executing many combinations of size three or more that contain pairs of these services, as such pairs are already known to interfere on VM types cheaper than VM_{10}. As a result, SF_1 reaches the optimal solution faster than SF_3, which continues exploring such non-working combinations. On the contrary, in Hotel Reservation and Media Service, many service pairs works well on cheaper VM types but larger combinations require costlier VM types. For example, the optimal deployment for Hotel Reservation includes four pairs of services placed on a VM_1 and three instances of VM_3. Thus, SF_3 can quickly determine that additional experiments increase the cost of deployment and stop the execution while SF_1 will keep running these experiments.

Answer to RQ1: Conditions employed by KUBER allow it to arrive at optimal deployments with the minimal search cost and execution time for all our subject applications. This is because its Condition 1 (SF_1) helps eliminate many relatively cheap experiments, Condition 2 (SF_2) helps eliminate a few relatively expensive experiments, and Condition 3 (SF 3) provides a global stopping condition. The savings achieved by KUBER increase as the number of services grows.

RQ2 (Sampling vs. Prediction). Comparing the performance of KUBER to that of P shows that P found a costlier solution for two out of four subjects: Media Service and Hotel Reservation. This is due to inaccuracies in predicting optimal VM type for combinations. For Media Service, P provided incorrect predictions for 20% of combinations it tried, resulting in (a) unnecessary executions and (b) missing some VM types that could have worked in practice. In fact, for combinations that were not predicted correctly, P made two wrong predictions on average, with only a third being a successful one. It also missed 14 correct placements and, as a result, missed the optimal deployment cost by 20% (\$309 vs. \$257). For Hotel Reservation, the obtained solution was 11% more expensive than the optimal one found by KUBER (\$92 vs. \$83). In this case, P provided incorrect predictions for 7% of combinations, making one wrong predictions on average.

In all four case studies, including Sock Shop and Social Network applications where P was able to identify the optimal solutions, the search induced higher cost and longer execution time: P spent \$66 vs. \$12 for KUBER, on average (450% increase) and executed for 194 hours vs. 53 hours for KUBER, on average (266% increase). The increase in search cost is more substantial than in execution time because incorrect predictions lead the approach to skip less expensive and execute costlier experiments. For example, in Sock Shop, P incorrectly predicts the combination of the Order and User services not to meet its performance target on the cheaper VMs, VM_2 and VM_3, and executes on a costlier VM_{10}.

Answer to RQ2: Prediction errors cause P to both execute unnecessary experiment and miss experiments that can lead to optimal deployments. As a result, KUBER is able to find a substantially less costly deployment for one of the subject applications. KUBER converges on a solution with lower execution time and search cost for all subject applications.

VI. Limitations and Threats to Validity

For **external validity**, our results may be affected by the selection of applications that we used and may not generalize beyond our subjects. We attempted to mitigate this threat by using a set of benchmark applications provided by a highly cited related work on microservices. As we used applications of reasonable size and complexity, we believe our results are reliable. Yet, while in all our subject applications KUBER was able to find the optimal deployment, more experiments are needed to decide on the appropriate search budget for larger applications, e.g., with hundreds of services, as reaching the optimal deployment in these cases in infeasible. Moreover, our selection of performance targets could influence our results. We mitigated this threat by applying the same target calculation method to all applications and compared approaches.

For **internal validity**, our implementation of KUBER, the WID algorithm, and our re-implementation of PARIS as part of building the P solution could have deficiencies. We controlled for the threat by having two authors of this paper reviewing each other's code. Two authors of the paper also manually and independently analyzed the obtained results, discussing their findings and any possible inconsistencies. We also make our implementation and evaluation setup publicly available [19] to encourage validation and replication of our results.

The main **limitation** of our approach is that it currently finds the optimal deployment for a fixed workload produced by an application test. In practice, the workload can fluctuate over time and, thus, autoscaling is used to automatically increase/decrease the number of instances of each service [41]. As re-running KUBER from scratch every time autoscaling takes place, future work on identifying the optimal deployment incrementally, during autoscaling, is needed. Moreover, when evaluating the performance of a particular service combination, KUBER deploys other services on separate machines, increasing the costs. Future work could investigate approaches to mitigate this problem, e.g., by mocking of other services.

VII. Discussion and Related Work

We discuss existing work for efficiently finding a cost-effective VM type where an application (task/job/service) satisfies its performance target along three main categories:

Prediction-based approaches [9], [42], [43], [8], [44], aim to infer performance of an application by assessing its similarity with previously profiled benchmarks. For example, AROMA [9] extracts resource consumption patterns from a set of benchmark jobs runs them in a staging cluster of low-capacity VMs using a reduced workload, and further clusters the jobs by the extracted patterns. It then runs a new job in the staging cluster, uses the obtained signature to determine the similarity of the job with a particular cluster, and applies the cluster's trained ML model to predict performance of that job. PARIS [8], which was extensively discussed in Section IV, uses a prediction-based approach but, instead of running a job on a staging cluster with a smaller workload, runs it on a subset of VM types and infers its performance on other VM types. Prediction-based approaches heavily rely on similarity of the profiled workloads with each other. As our evaluation shows, assuming such similarity can lead to erroneous predictions. Moreover, in the context of microservice-based applications, profiling various possible combinations of services becomes a challenging and expensive task by itself.

White-box sampling-based approaches [45], [46], [47], [10] do not rely on similarity with other, previously profiled benchmarks but rather assume certain application properties, e.g., that computation scales linearly with data. They mainly work by building analytical performance models specific to an application, executing workloads in a carefully selected subset of VMs and estimating performance of the same workload for different configurations (e.g., on other VMs or for larger input data). For example, Ernest [10] builds a mathematical performance model of a job based on the behavior of a job on small samples of data and then predicts its performance on larger data and cluster size. Such approaches are not easily extendable beyond applications with fixed internal structure, e.g., Spark jobs, and thus have limited applicability for general-purpose applications, like microservices.

Black-box sampling-based approaches [7], [11], [48] do not assume any application properties or similarity to existing benchmarks. Instead, they typically rely on VM similarity metrics (e.g., CPU cores, frequency, memory specifications) to predict the execution time of a workload on a new VM type using data collected from already executed VMs. These approaches iteratively update their prediction models by selecting the next best VM type to sample. For example, CherryPick [7] uses an ML-based optimization technique to predict the VM type where the cost of running the job is minimized. It then samples this VM type, collects runtime data, and updates the ML model. Scout [12] combines prediction- and sampling-based techniques. It uses performance predictions similar to that of PARIS [8] to improve CherryPick [7] by avoiding running experiments on VM types that are predicted not to work. Our work largely falls into this category. Yet, while most black-box sampling-based approaches focus on predicting execution time given the specification of a job and a VM, we focus on addressing an orthogonal scalability problem induced by a large number of service combinations.

VIII. Conclusion

As cloud providers typically offer a variety of virtual machine (VM) types, each with its own hardware specification and cost, and microservice-based applications contain multiple services that can be co-located on different VM types, selecting the cheapest VM types for deploying a microservice-based application becomes a time-consuming and costly task. This paper formally defined the problem of identifying an optimal deployment for a microservice-based application and proposed a scalable solution that addresses this problem, implemented in a tool named KUBER. We empirically evaluated KUBER on four open-source microservice-based applications and showed that it can identify the desired deployment faster and with lower search cost than other possible alternatives.

References

[1] "Microservices: a Definition of This New Architectural Term," https://martinfowler.com/articles/microservices.html, 2014.

[2] "How Uber Monitors 4,000 Microservices," https://www.cncf.io/case-studies/uber/, 2019.

[3] "Netflix Architecture: How Much Does Netflix's AWS Cost?" https://www.cloudzero.com/blog/netflix-aws, 2021.

[4] M. Weinberger, "Lyft Has to Pay Amazon's Cloud at Least $8 Million a Month Until the End of 2021," https://www.businessinsider.com/lyft-ipo-amazon-web-services-2019-3, 2019.

[5] "Byte Down: Making Netflix's Data Infrastructure Cost-Effective," https://netflixtechblog.com/byte-down-making-netflixs-data-infrastructure-cost-effective-fee7b3235032, 2020.

[6] "Our Journey Towards Cloud Efficiency," https://medium.com/airbnb-engineering/our-journey-towards-cloud-efficiency-9c02ba04ade8, 2021.

[7] O. Alipourfard, H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics," in *Proc. of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 469–482.

[8] N. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. Katz, "Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach," in *Proc. of the Symposium on Cloud Computing (SoCC)*, 2017, p. 452–465.

[9] P. Lama and X. Zhou, "Aroma: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud," in *Proc. of the International Conference on Autonomic Computing (ICAC)*, 2012, pp. 63–72.

[10] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics," in *Proc. of the Symposium on Networked Systems Design and Implementation (NSDI))*, 2016, pp. 363–378.

[11] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies, "Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM," in *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 660–670.

[12] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh, "Scout: An Experienced Guide to Find the Best Cloud Configuration," Tech. Rep., 2018.

[13] C.-J. Hsu, V. Nair, T. Menzies, and V. Freeh, "Micky: A Cheaper Alternative for Selecting Cloud Instances," in *Proc. of International Conference on Cloud Computing (CLOUD)*, 2018, pp. 409–416.

[14] "A Microservices Demo Application: Sock Shop," https://microservices-demo.github.io/.

[15] "Amazon Elastic Compute Cloud," https://aws.amazon.com/ec2.

[16] Amazon, "Amazon EC2 Instance Types - Amazon Web Services," https://aws.amazon.com/ec2/instance-types/.

[17] S.-C. Chang, J.-J. Liu, and Y.-L. Wang, "The Weighted Independent Domination Problem in Series-parallel Graphs," *Intelligent Systems and Applications*, vol. 274, pp. 77–84, 2015.

[18] P. P. Davidson, C. Blum, and J. A. Lozano, "The weighted independent domination problem: Ilp model and algorithmic approaches," in *Proc. of the European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP)*, 2017, pp. 201–214.

[19] H. Kadiyala, A. Misail, and J. Rubin, "Supplementary Materials." https://resess.github.io/artifacts/Kuber/.

[20] Microsoft, "Chapter 1: Service Oriented Architecture (SOA)," https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx, 2016.

[21] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and Challenges of Microservices: An Exploratory Study," *Empirical Software Engineering*, vol. 26, no. 63, 2021.

[22] "The Psychology of Web Performance," https://blog.uptrends.com/web-performance/the-psychology-of-web-performance/.

[23] H. Jayathilaka, C. Krintz, and R. Wolski, "Service-level Agreement Durability for Web Service Response Time," in *Proc. of the International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 331–338.

[24] Amazon, "Azure Virtual Machine Series," https://azure.microsoft.com/en-au/pricing/details/virtual-machines/series/.

[25] Microsoft, "Azure Linux Virtual Machines Pricing," https://azure.microsoft.com/en-ca/pricing/details/virtual-machines/linux/.

[26] "Instance Performance Variability," https://forums.aws.amazon.com/thread.jspa?threadID=22830.

[27] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An Analysis of Performance Interference Effects in Virtual Environments," in *Proc. of the International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2007, pp. 200–209.

[28] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring Interference between Live Datacenter Applications," in *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–12.

[29] Docker, "Docker," https://www.docker.com/.

[30] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A Holistic Evaluation of Docker Containers for Interfering Microservices," in *Proc. of the International Conference on Services Computing (SCC)*, 2018, pp. 33–40.

[31] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An Open-source Benchmark Suite for Microservices and their Hardware-software Implications for Cloud & Edge Systems," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 3–18.

[32] "OpenNebula," https://opennebula.io/.

[33] "Kubernetes," https://kubernetes.io/.

[34] "Istio Distributed Tracing," https://istio.io/latest/docs/tasks/observability/distributed-tracing/.

[35] "The Yelp Dataset," https://hanlululu.github.io/SocialGraphYelp.io/Page1_Dataset.html.

[36] "TMDB 5000 Movie Dataset," https://www.kaggle.com/tmdb/tmdb-movie-metadata.

[37] R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*, 2015.

[38] "Amazon Socks," https://www.amazon.com/s?k=socks&crid=2VSAR07AQ7QLX&sprefix=socks%2Caps%2C182&ref=nb_sb_noss.

[39] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 77–88.

[40] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao, "Rhythm: Component-distinguishable Workload Deployment in Datacenters," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2020, pp. 1–17.

[41] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling Web Applications in Clouds: A Taxonomy and Survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[42] X. Li, M. A. Salehi, M. Bayoumi, and R. Buyya, "CVSS: A Cost-efficient and QoS-aware Video Streaming Using Cloud Services," in *Proc. of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 106–115.

[43] N. Zacheilas and V. Kalogeraki, "ChEsS: Cost-Effective Scheduling Across Multiple Heterogeneous Mapreduce Clusters," in *Proc. of International Conference on Autonomic Computing (ICAC)*, 2016, pp. 65–74.

[44] A. Chung, J. Park, and G. Ganger, "Stratus: Cost-aware Container Scheduling in the Public Cloud," in *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, 2018, pp. 121–134.

[45] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *Proc. of the International Conference on Autonomic Computing (ICAC)*, 2011, pp. 235–244.

[46] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proc. of the european conference on Computer Systems (EuroSys)*, 2012, pp. 99–112.

[47] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan, "Perforator: Eloquent performance models for resource optimization," in *Proc. of the Symposium on Cloud Computing (SoCC)*, 2016, pp. 415–427.

[48] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwaenepoel, and D. Garlan, "Lynceus: Cost-efficient Tuning and Provisioning of Data Analytic Jobs," in *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 56–66.