Policy Synthesis for Resource Allocation in Clouds

Sathish Gopalakrishnan

Abstract—**Most resource allocation and job scheduling problems are NP-hard. The standard approach to addressing such problems is to propose approximation algorithms or heuristics and, in some situations, derive bounds on the performance of these approaches. Depending on the complexity of the optimization objective, we may not be able to easily infer good heuristics. One example of such an objective is when we have to provide *percentile guarantees*: $p\%$ of jobs in a job set must complete within some deadline. We present an approach to *synthesizing resource allocation policies* using optimal solutions to a small number of problem instances as examples. Our approach relies on supervised learning via decision trees. We use this approach as part of a framework we call Pennyworth and apply this framework to some realistic workloads. Our results also suggest that we may be able to obtain policies that have good statistical behaviour and worst-case bounds. More generally, we make connections to work on program synthesis using input-output examples.**

I. Introduction

Resource allocation and scheduling problems are concerned with assigning jobs to processors and with imposing an order for the execution of these jobs. A resource allocation policy helps us determine where and in what order jobs should be scheduled. In developing such policies, we seek to minimize the use of resources while satisfying certain constraints such as job deadlines. We consider such problems in the context of cloud computing and present a framework for *synthesizing resource allocation policies*.

Cloud computing has commoditized computation but users need to solve other problems to make optimal use of these resources. We list three related problems that *users of cloud computing platforms* need to solve from this perspective:

- **Resource provisioning.** Cloud computing requires the instantiation of virtual machines, and virtual machines (VMs) may be of different "sizes" or capacities. How many VMs are needed for an application? What should the sizes of these VMs be?
- **Job placement.** How should we assign work to the VMs?
- **Scheduling.** In what order should jobs be executed on a VM?

We answer these three questions so as to manage a group of related jobs with performance (timeliness) requirements and with the objective of minimizing the cost of cloud computing resources. We use the term *resource allocation policy* to capture the answers to the three questions enumerated above for a particular application (set of jobs).

The innovation in our work is the application of statistical learning and the ability to accommodate a variety of performance objectives. We have developed cost-aware techniques to answer these resource allocation problems. We evaluate these techniques in detail, and these techniques have been embedded in a tool, called **Pennyworth**, that can generate resource allocation policies. *Our work addresses systems where jobs are expected to meet certain timeliness requirements and violations of these requirements result in a (monetary) penalty.*

As an example, let us consider a user who is collecting data, at a data center, from a variety of sensors that are part of the Internet of Things. This user may want to process the data gathered every half-hour, and the data processing can be represented as a set of jobs/programs that needed to be executed. The performance requirement could be that 95% of such jobs be completed within a deadline or latency bound *after* the data has reached the data center. The user may not want to be directly concerned with the choice of VMs and scheduling decisions (i.e., the resource allocation policy) but the user is interested in achieving the latency goal at a low cost. We develop a suite of techniques for helping such a user.

Identifying policies to meet deadlines requires some knowledge of the execution times of jobs. Job execution times may be obtained using worst-case execution time analysis, which is specific to a physical/virtual machine, and is useful for worst-case guarantees. Statistical guarantees require job execution time distributions, and we propose sampling for obtaining some execution time information when the performance needs are expressed as statistical guarantees.

Given a certain workload or application and a particular performance objective, such as minimize mean latency, the problem we tackle can reduce to the classic discrete optimization problem of bin packing. On the other hand, for a requirement such as percentile metrics on latency or response times, we may need another tactic. Our approach, as embodied in Pennyworth, is to identify effective resource allocation policies *by learning heuristics* that are suited to an application's requirements. We use supervised learning, and the training phase uses templates (programs + representative inputs) as training data to construct decision trees.

For the rest of this discussion, we will refer to the techniques we have developed as *Pennyworth* although that is the name for the tool that represents an implementation of the ideas. We will outline some of the design objectives that inform Pennyworth and then we enumerate our contributions before we elaborate on our work. *The work we present is preliminary and is restricted to scheduling batches of jobs. Similarly, we sketch key ideas for brevity of exposition.*

A. Design Principles

Pennyworth is a resource provisioning tool, and we articulate some of the design objectives that are reflected in our work.

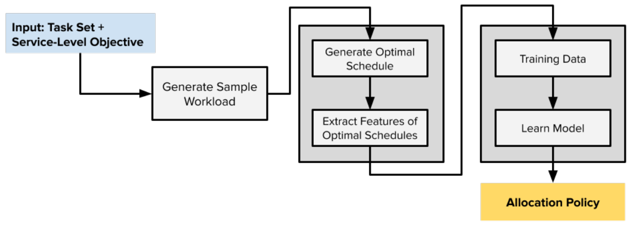

Input: Task Set + Service-Level Objective
Generate Sample Workload
Generate Optimal Schedule
Extract Features of Optimal Schedules
Training Data
Learn Model
Allocation Policy

Fig. 1: **The Pennyworth Framework**

1) **Comprehensive.** We have designed Pennyworth to provide answers to many of key questions related to resource allocation on a cloud computing platform. A cloud computing platform may offer a variety of VM types so Pennyworth should be able to indicate what types of VMs are needed, how many VMs of each type are needed, and how jobs are assigned to these VMs.
2) **Cost-informed.** Cloud computing platforms offer different VM types at different price points (often, cost/hour of using a VM). Pennyworth uses this information in identifying resource allocation policies. Further, if a Pennyworth user deploys a particular application on a cloud computing platform and then sells that application as service to others, we consider the cost implications of missing performance requirements (as captured in a service-level agreement between the Pennyworth user and her customers).
3) **Handle varied applications.** Different applications will have different performance needs. Pennyworth supports a variety of performance metrics. A performance metric may pertain to a specific job (such as response time) or be defined over a batch of jobs (such as maximum response time). The techniques used in Pennyworth intentionally rely on generic application features so that we can handle a range of applications and different performance metrics.

B. Pennyworth Workflow

To better understand our contributions (Section I-C), it is useful to consider the workflow involved in using Pennyworth.

Pennyworth's users will provide the tool with a task set, performance requirements and some representative inputs for the tasks. We can think of a task as being a program binary and a job as a combination of the program and program inputs; in other words, a job is a particular instance of a task. This information is used to train Pennyworth and develop a resource allocation policy. These inputs are also used to generate policies for alternative performance requirements.

To help a user decide on a suitable strategy, Pennyworth can also produce a tradeoff function that can be used to estimate the cost associated with each strategy. We use the term *workload* to refer to the set of jobs that need to be executed. A concrete set of jobs may include multiple instances of the same task (instantiations of the same program but with different inputs).

At runtime, Pennyworth can also use the synthesized policies to determine the resources needed as work arrives. A policy explicitly indicates the number and the types of VMs that need to be started on the cloud computing platform, the allocation of jobs to VMs, and the execution schedule.

C. Contributions

The principal contribution of our work is to demonstrate that it is possible to synthesize resource allocation strategies from examples and that this approach can yield solutions that are near-optimal from a cost perspective.

We have built Pennyworth to learn good strategies using features of optimal allocations for sample workload (Section III). The optimal allocations for some representative workload essentially provides *examples* of good decisions. We rely on cost and performance metrics that are easily observed in optimal allocations, and these observations/features allow for learning near-optimal allocations when the workload and the performance goals change (Section **??**). The approach we propose can handle batch workload, where all jobs are known in advance, and online job arrivals (Section IV).

Using database workload as well as synthetic workload, and working with Amazon's EC2 cloud computing service, we demonstrate the effectiveness of our ideas (Section V).

Our primary contribution can be broken down into several important sub-aspects:

1) We have identified key features of optimal resource allocations (Section III-D) that can be used by Pennyworth to synthesize resource allocation policies for job batches. These policies result in costs that are no more than 10% in excess of optimal costs, when we can find optimal policies (Section V-B). An insight that is noteworthy is that the best path found using A* search can be used to inform the construction of decision trees. The use of decision trees results in policies that can be understood and also analyzed for their worst-case behaviour. *The critical insight is that the use of A* search allows us to determine decision paths that can be translated to a comprehensible policy. The use of graph-based search allows us to extract features that can be used to train a decision tree.*
2) Pennyworth can generate new policies for job batches within one minute for a variety of workload and performance goals (Section V-C).

II. System Model and Problem Formulation

A. System Model

Our goal is to enable simple resource provisioning for application that rely on cloud computing infrastructure. We assume that a Pennyworth user relies on a cloud service provider who can rent virtual machines (VMs) of different types. VM types correspond to the resources associated with a VM and these resources include the main processing unit (CPU), memory and disk capacities, and possibly computational accelerators (such as graphics processing units and specialized processors for applications such as machine learning).

In our model of the system, we assume that a job can be executed on any VM. This system property can be attained by using replicated storage or a separate storage service that is globally visible (such as Amazon S3). We also focus on *non-preemptive scheduling of jobs* on a VM.

B. Formal Definitions and Problem Statement

JOBS, TASKS AND WORKLOAD We are primarily interested in allocating resources for, and scheduling, a set of jobs. A **job** is an instance of a **task**. As an example, *face recognition* is a task that may be implemented using a specific algorithm. A job of this task will involve recognizing faces in a specific image. A task T, therefore, represents a computational activity or program. $\tau = \{T_1, T_2, \dots, T_n\}$ represents a set of tasks. A *job*, J_j, is a particular instance of a task (a specific invocation of a program). To indicate that J_j is an instance of T_i, we will use the notation $(T_i)J_j$ when necessary. We will use the term **workload** to refer to a set of jobs $\{J_1, \dots, J'_n\}$.

MACHINES The user also provides a set of (virtual) machines types, $\Pi = \{\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_m\}$, that can be used to schedule the tasks. $\mathcal{M}_k$ is associated with a monetary startup cost $\$^s_k$, which is incurred when the VM instance is launched. There is monetary cost of operation, $\$^o_k$, which is incurred per time unit.

EXECUTION TIMES For each task, T_i, and each machine type, $\mathcal{M}_k$, we obtain an execution time estimate for the task on that machine type, and denote this estimate as $\hat{e}_{i,k}$. We will use the notation $e_{j,k}$ to denote the actual execution time of $(T_i)J_j$ on $\mathcal{M}_k$.

ALLOCATION An allocation $\mathcal{A}$ for a job set $\mathcal{J} = \{J_1, \dots, J_{n'}\}$ is represented by a set of tuples $\{(\mathcal{M}_1, \mathcal{L}_1), (\mathcal{M}_2, \mathcal{L}_2), \dots\}$ where $\mathcal{M}_k$ represents a machine of a type in Π and $\mathcal{L}_k$ is a list that contains jobs to be executed in order on an instance of $\mathcal{M}_k$. For $\mathcal{A}$ to be an allocation, we need: (i) $\bigcup_k \mathcal{L}_k = \mathcal{J}$, and (ii) $\mathcal{L}_{k_1} \bigcap \mathcal{L}_{k_2} = \emptyset \ \forall (k_1, k_2),\ k_1 \neq k_2$.

PERFORMANCE GOALS AND COSTS We can express a variety of performance goals with Pennyworth:

- **Per-task guarantee.** Each job/instance of a task is associated with a specific response time guarantee;
- **Maximum response time guarantee.** No job (across all tasks) can exceed a maximum permitted response time;
- **Mean response time guarantee.** The mean response time for a set of jobs is bounded by a given d;
- **Percentile guarantee.** At least $p\%$ of tasks in a task-set must be completed within a given deadline d.

Such performance criteria are specified in the service level agreement (SLA) that a service provider and the user deploying applications enter into. An SLA would include specifications for the workload, the performance goals and penalties incurred by the service provider when an application's goals are not met. We will use notation $\mathcal{G}$ to refer to a specific performance goal that is selected. A performance goal is also referred to as a *service level objective* (SLO).

We use a monetary cost model that accounts for starting up (virtual) machines, for the operational cost incurred, as well as penalties when a service level objective is not met. We will use the notation $l(\mathcal{A}, \mathcal{G})$ to refer to the loss incurred when an allocation $\mathcal{A}$ violates performance goal $\mathcal{G}$.

The cost of an allocation can be expressed as the sum of start-up costs and operational costs for each machine instance as well as the cost of SLO violations:

$$cost(\mathcal{A}, \mathcal{G}) = \sum_{(\mathcal{M}_k, \mathcal{L}_k) \in \mathcal{A}} \left[\$^s_k + \sum_{J_j \in \mathcal{L}_k} \{\$^o_k \times e_{j,k}\} \right] + l(\mathcal{A}, \mathcal{G}). \tag{1}$$

PROBLEM STATEMENT Using the notation that we have set up, **our goal is to take as input a task set τ, a performance goal P, a set of machine types Π, and synthesize an allocation policy π that minimizes the allocation cost for any workload generated from τ.** The policy π, in turn, takes a job set $\mathcal{J} = \{J_1, \dots, J_{n'}\}$ as input, where every $J_j \in \mathcal{J}$ is an instance of some task $T_i \in \tau$. The output of the policy is a resource allocation $\mathcal{A}$ that minimizes the total cost (Equation 1).

C. Computational Complexity

The general problem includes the case when the loss function is such that $l(\mathcal{A}, \mathcal{G}) = \infty$ and $\$^s_i = \$^s_j, \forall (i, j)$. Consequently, the problem reduces to the bin packing problem, which is NP-Hard problems [1]. In some situations we may be able to leverage well-known approximation algorithms or heuristics with acceptable guarantees. *But*, we do not have any approximation guarantees currently for performance goals such as the tail latency guarantee (or other percentile-based metrics).

III. SYNTHESIZING RESOURCE ALLOCATION POLICIES

The central functionality that Pennyworth provides is *policy synthesis* for resource allocation. The resource allocation policies are *synthesized* from examples and we use *supervised learning* to extract a policy from the examples.

We now detail the steps involved in policy synthesis.

A. Sketch of Policy Synthesis

From a given application definition, which in our case is a set of tasks and a service-level objective (SLO), Pennyworth generates a set of resource allocation policies that can be utilized to achieve the SLO at a low cost.

Our objective is to minimize the total cost of operation (Equation 1). To achieve this, our approach involves the following main steps (Figure 1):

1) **Generating workload samples.** We create a large set of workloads by sampling from the task-input pairs that are user-provided.
2) **Optimal allocation solutions.** We identify optimal resource allocations for the workload samples, and these optimal allocations become examples. This step can be computationally expensive. We use a search algorithm to identify optimal allocations for each sample workload. To gain some efficiency, we model this search using

a graph with the vertices representing specific allocations, edges representing incremental allocations, and edge weights corresponding to the cost involved with corresponding allocation decision.

3) **Feature extraction and model generation.** We characterize the example solutions using specific *features* of the allocation. We then use these features to train a *decision tree* that represents the synthesized policy.

In the subsequent subsections, we will discuss each of the steps in greater detail.

B. Generating Sample Workload

An input to Pennyworth is the set $\tau = \{T_1, \ldots, T_n\}$ that represents the tasks that may need to be executed. A task may be thought of as a program and a task instance, or job, would then be the program coupled with external inputs. A sample workload would then be a set of jobs, where each job is an instance of a task in τ.

We generate M_s sample workloads using τ. In each workload, we set the number of jobs to n_s. The choice of M_s and n_s affect the quality of policy synthesis; they must both be sufficiently large so that we can understand task-level interactions. On the other hand, for all sample workloads we will identify optimal solutions (by searching the space of solutions). n_s has to be suitably small to keep this step of the process time-efficient. We use uniform sampling from the space of tasks to generate workloads. This decision around how we sample is significant. If we do not sample uniformly then the decision tree model that we construct may not offer good decisions because it has little information about certain tasks or certain task combinations. We generate a large number of workload samples so that the examples will include cases where the number of unique tasks is small and this allows the model to deal with imbalance in a workload ("workload skew").

Different jobs of a task may have different execution times. When generating the sample workload we use information about the specific jobs to obtain execution time estimates for these jobs on different VM types. These estimates are based on prior profiling of the jobs. We then use these execution time estimates to obtain an optimal resource allocation for the sample workload.

We consider sample workloads where more than one workload involves the same mix of tasks as long as these workloads differ in the execution times of specific jobs. If $\tau = \{T_1, T_2, T_3\}$, sample workloads W_1 and W_2, with 6 jobs, could both contain three jobs of T_1, two jobs of T_2, and one job of T_3 as long as the job of T_3, for example, has a different execution time in the two samples. Doing so allows us to capture the impact of execution time variations across jobs.

C. Constructing Optimal Allocations

Having generated sample workloads, Pennyworth needs to find optimal resource allocations for these problem instances. The general problem is NP-Hard. This step uses a search-based strategy.

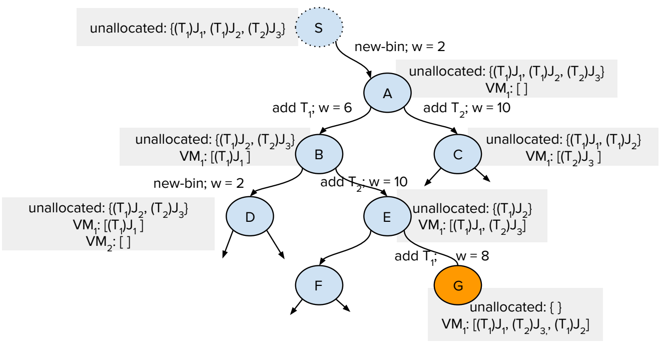

unallocated: {(T1)J1, (T1)J2, (T2)J3}
S
new-bin; w = 2
unallocated: {(T1)J1, (T1)J2, (T2)J3} VM1: [ ]
A
add T1; w = 6
add T2; w = 10
unallocated: {(T1)J2, (T2)J3} VM1: [(T1)J1]
B
C
unallocated: {(T1)J1, (T1)J2} VM1: [(T2)J3]
new-bin; w = 2
add T2; w = 10
unallocated: {(T1)J2, (T2)J3} VM1: [(T1)J1] VM2: [ ]
D
E
unallocated: {(T1)J2} VM1: [(T1)J1, (T2)J3]
add T1; w = 8
F
G
unallocated: { } VM1: [(T1)J1, (T2)J3,, (T1)J2]

Fig. 2: Example of an Allocation Graph. The initial job set has three jobs; two jobs correspond to instances of T_1 and one job corresponds to an instance of T_2. S is the initial vertex and G represents a final vertex.

We use a weighted directed graph representation of allocations because this representation captures the search-based strategy. A sequence of decisions is represented by a path in the graph we construct, and an optimal allocation is found through a *best path search*. The path captures both the allocation and the decisions leading to the allocation.

CONSTRUCTING AN ALLOCATION GRAPH A sample workload is a set of jobs $\mathcal{J} = \{J_1, \ldots, J_{n_s}\}$. To find the optimal resource allocation for this job set, we construct an *allocation* graph (see Figure 2 for an example) with a vertex set and an edge set as follows: a vertex captures a partial allocation and a list of jobs that need to be processed, and an edge represents a decision such as assigning a job to a virtual machine or adding a new virtual machine. With each edge we can associate a cost (such as the cost of starting up a new VM or the expected cost of adding a job to a VM because of how that decision may affect response times).

Each vertex v captures allocation decisions made: a list of VMs that have been rented as well as their types, the placement of jobs on VMs and the order in which jobs have to be scheduled, and the set of jobs that need to be processed. The *initial* vertex $v_0 \in V$ maps to the state when none of the jobs have been allocated a VM and the set of unprocessed jobs is $\mathcal{J}$. If the set of unprocessed jobs is not empty at some vertex v then that vertex represents an incomplete allocation. A vertex $v_f \in V$ that has an empty set of unprocessed jobs would be a *final* vertex and the allocation it represents would be a *complete* allocation. Suppose vertex v represents the allocation $\mathcal{A}$ with performance goal $\mathcal{G}$, we can use the notation $cost(v, \mathcal{G})$ in place of $cost(\mathcal{A}, \mathcal{G})$.

Every edge of an allocation graph corresponds to an allocation-related decision:

1) A *new-bin* decision edge $(u, v, \mathcal{M}_k)$ that connects vertex u to v by adding a VM of type $\mathcal{M}_k$ to the available VMs (or bins) to use. Such an edge will have weight $w(u, v, \mathcal{M}_k) = \$^s_k$, which is the cost of purchasing a type-$\mathcal{M}_k$ bin.
2) An *add-item-to-bin* decision edge (u, v, T_i) that con-

nects u to v by adding a job of T_i to a bin that was available in the allocation represented by u. An edge of this type will have weight that corresponds to cost incurred by adding a job to type-$\mathcal{M}_k$ bin. The [expected] cost of the decision includes the operational cost *and* any losses that may arise from violating SLO P: $w(u, v, T_i) = (\$_k^o \times \hat{e}_{i,k}) + [l(v, \mathcal{G}) - l(u, \mathcal{G})]$. The difference between the allocations at u and v [uniquely] identify the bin that was chosen for the item.

The weight of a path in an allocation graph from v_0 to some v_f (in the example shown in Figure 2 this is the weight of, say, the path from S to G) corresponds to the cost of a complete allocation for a given SLO P. All complete schedules terminate at some final vertex; we are, therefore, interested in the least expensive path from the start vertex v_0 to any of the final vertices in the graph.

In our example, the path $S \rightarrow A \rightarrow B \rightarrow E \rightarrow G$ represents the resource allocation decision wherein jobs J_1, J_3 and J_2 are assigned to the same VM and are to be executed in that order.

GRAPH PRUNING One can easily envision a very large graph of allocation decisions, and processing such a graph will be time consuming. We can perform some pruning to reduce the time required to find an optimal path.

First, we consider the option of adding a new VM only if the VM that was added most recently has at least one job allocated to it. This restriction eliminates the sub-optimal decisions that may involve provisioning but not using a VM.

Second, in our allocation approach, tasks are added to the VM that was most recently provisioned. This constraint eliminates some redundancy in the allocation graph; a particular combination of virtual machine types and job schedules can be traversed by exactly one path in the graph and not multiple paths. This constraint does not result in sub-optimality.

SEARCH ROUTINE The optimal allocation corresponds to a path of minimum weight that begins with the start vertex and ends with *any* final vertex. In Pennyworth, we use A* search [2]. A* search is *complete*: it is guaranteed to find a minimum weight path, if such a path exists. A* search can also utilize a given heuristic to find the minimum weight path efficiently. Heuristics that A* uses must be *admissible*: in other words, we can use $h(v)$ as a heuristic if this function represents an *estimate* of the path weight from the initial vertex to a final vertex *that is no larger than the true weight*. A* search is optimal from a time efficiency perspective in the sense that no other complete search routine will explore fewer vertices than A* search given the same heuristic.

The heuristic that the A* search routine uses is problem specific. In Pennyworth, the heuristic function would be $cost(v, \mathcal{G})$. The heuristic is, as one would expect, different for different SLOs.

We further have to distinguish between performance objectives or SLOs that are *non-decreasing with respect to subsequent allocation decisions* and those that are not. A performance objective is non-decreasing if adding a new job will not reduce the value of the objective. Maximum response time, for example, is non-decreasing with new jobs because additional jobs may increase the maximum response time but not decrease this metric. On the other hand, average response time is not non-decreasing because a new job with a short execution time may reduce the metric.

In the case of performance metrics that are *non-decreasing* the loss function is also non-decreasing. If there is a directed edge $v_i \rightarrow v_j$ in the allocation graph, then $l(v_j, \mathcal{G}) \geq l(v_i, \mathcal{G})$. We can then use the heuristic function $h(v) = \sum_{(T_i) J_j \text{ is unassigned}} \min_{\mathcal{M}_k} [\$_k^o \times \hat{e}_{i,k}]$.

The heuristic function we have defined ignores the cost of starting new VMs and accounts for minimizing only the operational cost involved in scheduling the remaining jobs. We can show that this function is admissible for A* search.

For performance metrics without a suitable heuristic function, we can still use A search without the use of the heuristic; the search will need to explore more states but will find a suitable solution if one exists.*

D. Characterizing Optimal Solutions

Using A* search, we can obtain optimal allocations for each of the workload samples. These examples of optimal allocations are hard to work with directly, so we need to process them and identify specific characteristics or *features* that are representative of the optimal allocations.

We can think of the A* search as making decisions at each vertex of the allocation graph. The decision being made is the edge to follow from one vertex to the next until a final vertex is reached. More importantly, at each vertex the decision is made based on the allocation at that vertex and the set of jobs that are un-allocated. Thus the decisions made by the A* search are independent of parent vertices and child vertices but are dependent on the state that is captured by each vertex. We can, therefore, select key features at each vertex along the optimal path identified by A* search.

WHAT ARE GOOD FEATURES OF OPTIMAL ALLOCATIONS? One can enumerate numerous features that affect resource allocation decisions. There are task specific features such as execution times, program/code and program inputs as well as features of the chosen VMs such as processing power, available memory, and non-volatile storage capacity. Moreover, some combination of task- and VM-specific features may have a significant impact on the allocation decisions. Given this extremely large space of features, *we sought features that can be extracted efficiently and are sufficient to capture the performance metrics that influence allocation decisions.*

We used a few high-level principles in seeking features of optimal allocations:

1) The features should be independent of the specifics of tasks – such as source code or binary code – and should be independent of the performance objectives. This principle allow the policy synthesis aspect of Pennyworth to be separable from details of specific applications.
2) Good features should not depend on the number of jobs in the workload. This principle allows the synthesizer

to generalize from the examples, which may have fewer jobs relative to actual workload.

3) Features that we utilize should be orthogonal to each other to avoid redundancies. For example, the response time of a job on a particular VM depends on the jobs that are already scheduled on that VM. We do not need to capture the estimated response time and how many jobs are allocted to a VM. (Empirically, we did observe that using any one of these metrics is sufficient to obtain good allocations.)

Using the principles outlined above, we selected the following metrics or features for every vertex along an optimal path in the example allocation graphs:

- **Expected wait time.** The response time of a job is the sum of the wait time and the execution time of the job. Thus, for each bin or VM, the wait time is simply the sum of the execution times of the jobs that have already been scheduled on that VM. If there is a long wait time at a VM then it may make sense to assign only short jobs to that VM or to assign jobs with relaxed latency requirements.
- **Proportion of tasks.** For the bin that was most recently opened, or equivalently the VM that was started most recently, we extract a vector that represents the fraction of jobs of each task assigned to that bin. As an example, suppose the current bin has been assigned 12 jobs and 3 of these jobs are instances of task T_1 then the fraction of jobs of T_1 would be $3/12 = 0.25$. We only need to maintain this information for the bin that was opened most recently because that is the bin that would see new items being added (or a new bin will be opened).
- **Task instance cost.** This metric represents that cost that we expect to incur by adding an instance of a task to the most recently opened bin. If we were to add, for instance, job J that is an instance of task T_2 then this metric would correspond to the weight of the edge that represents that decision in the allocation graph. Maintaining this vector of costs allows any policy to decide whether it is better to allocate a job to the open bin or whether a new bin should be opened (equivalently, a new VM rented).
- **Unassigned tasks.** This feature is a bit vector that indicates if an instance (job) of a particular task is yet to be allocated. This information is relevant to optimal allocations: if an instance of, say, task T_3 is still unallocated then it may be better to allocate it next and only if such a job is not pending should we allocate a different job.

These features allow us to construct a decision tree model for resource allocation (Figure 3 illustrates an example).

We acknowledge that the features that we have just discussed do not allow us to identify, *unambiguously*, a vertex in an allocation graph. Consequently, we cannot learn or synthesize a policy that is based on the exact conditions in an example. Nevertheless, these features are [empirically] sufficient to illuminate the trade-offs that are made along optimal paths in the allocation graphs.

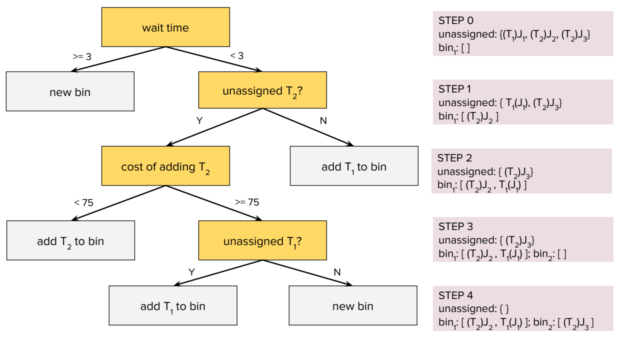


Fig. 3: Example of a Resource Allocation Decision (RAD) Tree

We do not claim that the features we have identified are *always* sufficient but we find, based on our experiments, that they do help Pennyworth synthesize policies for a reasonable range of workloads *and* provide expressive power to accommodate several common performance objectives.

E. Resource Allocation Policy

From a set of examples that are transformed into a training set using the features we discussed (Section III-D), Pennyworth relies on decision trees to generate a policy for resource allocation. We call the decision tree constructed a **Resource Allocation Decision Tree** or **RAD-tree**.

We can discuss how a decision tree is used with an example (illustrated in Figure 3). Consider a simple job set with three jobs: $\{J_1, J_2, J_3\}$ where J_1 is an instance of T_1 and jobs J_2 and J_3 are instances of T_2. An instance of T_1 has an expected execution time of 3 seconds and a latency requirement of 4.5 seconds. An instance of T_2 has an expected execution time of 1.5 seconds and a latency requirement of 1.5 seconds. For simplicity, we only consider a single type of virtual machine, and that jobs are executed in sequence and with isolation.

In the corresponding RAD-tree, the initial vertex indicates that no job has been scheduled. A RAD-tree may be parsed as follows: initially a bin/VM is created because none are available at the start. After a bin is opened, one would check what the wait time is for a job in that bin; this would be zero for a new bin. We would then check if some job (an instance of T_2 in the example) is unassigned, and then compute the cost of adding the job to the open bin. If the cost is less than a threshold (75 units), then an instance of T_2 is assigned to the open bin/VM. As there are two jobs still to be mapped to virtual machines, we process the RAD-tree again. After having allocated an instance of T_2 to the open bin, the (expected) wait time is now 1.5 seconds. We still have an instance of T_2 that is unassigned but now the cost of adding this instance to the open bin may be more than the threshold to satisfy performance goals so the decision is to check if there is an unassigned instance of T_1 and add it to the open bin. For the last job, we follow the same RAD-tree and add this job to a new virtual machine.

Every RAD-tree represents a resource allocation policy. For a set of jobs in our example, our RAD-tree policy will start with adding a job of T_2 and then a job of T_1 to a virtual machine; this policy will create new virtual machines, as needed, for subsequent jobs and repeat the process until all jobs have been assigned to virtual machines.

IV. Using Synthesized Policies

Pennyworth's policy synthesis is largely an *offline* activity. If one were to use the exploratory analysis then one would obtain a set of policies, with each policy representing a different trade-off between cost and service-level objectives. A Pennyworth user would then select one these policies for job scheduling.[1] A selected policy can be used for scheduling when the jobs are batched or for *online* scheduling when jobs arrive one by one. During the policy synthesis phase, we used execution time estimates for each based on prior profiling. At run-time, we would not have precise information about the jobs – although we would know the task that a job is an instance of – so we have to use generic estimates for execution time. When we apply a policy, we will need an execution time estimate for each job on the available VM types, and we use the mean execution time for jobs of the corresponding task as the execution time estimate.For now, we consider non-preemptive job execution (no parallelism between jobs on the same VM) and so we ignore interference between jobs.

After a user has a chosen a policy, Pennyworth suggests a resource allocation for a set of jobs. A user can then rent the VMs, and deploy jobs to VMs in the order prescribed by Pennyworth. This process was discussed earlier in more detail (Section III-E). Occasionally, we may encounter instances of tasks that were not part of the synthesis phase. In these situations, as long as the new tasks can be approximated – in terms of execution times – by a task that was part of the synthesis phase, Pennyworth can still produce resource allocation recommendations. In our approach, two tasks with similar execution time distributions – characterized by the Wasserstein distance – can be considered identical.

V. Evaluation

In evaluating Pennyworth, we seek to understand the cost of the allocations that are suggested (how close to optimal are they?) and the overhead of finding the allocations (how much time does it take?).

A. Setup

We implemented Pennyworth in Java, and we ran it on an Amazon Web Services (AWS) [3] t2.xlarge EC2 instances. AWS EC2's t2 instances are general-purpose computing virtual machine instances, and the t2.xlarge instances are provisioned with 16 GiB memory and have four virtual CPUs (vCPUs). *The principal step in the policy synthesis phase is the use of A* search for different example job sets. To speedup this step, we use 20 of these EC2 instances in parallel, each performing A* search for a different job set.*

[1]In certain situations, a user may desire a policy suite with policies that offer different performance trade-offs. The process of producing a policy suite is discussed in Section **??**.

VM types and costs We used Pennyworth to allocate and schedule work on AWS EC2's t2.small (2 GiB memory, 1 vCPU), t2.medium (4 GiB memory, 2 vCPUs) and t2.large (8 GiB memory, 8 vCPUs) instances. The cost of operation, *co*, for these VM types was \$0.023/hour, \$0.0464/hour and \$0.0928/hour, respectively. We estimated the startup cost as \$0.00076, \$0.0015, and \$0.0031, respectively. These estimates were based on repeated measurements of startup time for EC2 instances, where we found an instance being available for connections and initiating jobs at most 2 minutes after its status changed to *running*.

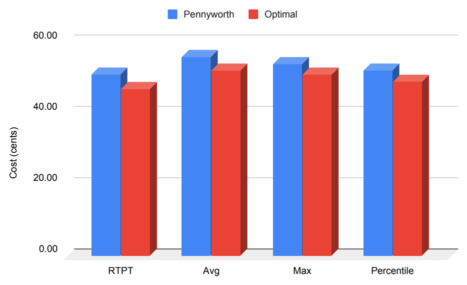
Generating Tasks and Jobs

We used database queries from the TPC-H decision support benchmark [4]. For these experiments, we used a 10GB setup of the benchmark running over PostgreSQL [5]. The TPC-H benchmark uses a query generator, QGEN, to generate queries using query templates. From our perspective, a query template is a task and a specific query generated for a template becomes a task instance or job. We used the first 10 templates from the TPC-H benchmark. These templates had average computation times between 90 seconds and 6 minutes.

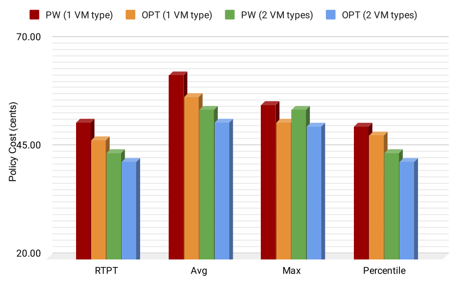
Policy Generation To generate the RAD-trees that represent policies, we used the J48 decision tree algorithm that is implemented as part of the Weka3 workbench [6]. The sample workload that we used to produce RAD-trees used 15,000 example job sets with 20 jobs per job set. We varied the number of tasks (TPC-H query templates) from 1 to 20 to generate different workload mixes. In our evaluation, we found that using a larger number of job sets or a larger number of jobs per job set did not lead to better RAD-trees; we therefore limited these parameters to obtain effective RAD-trees without increasing the policy synthesis time more significantly.

Service-Level Objectives We used several SLOs to evaluate Pennyworth:

1) **Maximum Response Time (Max)**: This SLO provides an upper-bound on the response time for each job. We set the response time requirement for each job to 12 minutes, which is $2\times$ the maximum computation time of any single job. The penalty for violating this bound was 1¢ per second beyond the response time requirement.
2) **Response Time Per Task (RTPT)**: This SLO requires that each job of a task not exceed a set response time requirement for that task. We set the response times for each task to be $3\times$ the mean response time of an instance of that task. The penalty for violating the response time requirement or deadline was 1¢ per second of tardiness.
3) **Average Response Time (Avg)**: This SLO provides an average response time for a given workload of $\overline{r}$. In our experiments with the TPC-H benchmark, we set $\overline{r}$ = 10 minutes, which is $2.5\times$ the mean computation time of a job. We set the penalty for violating this SLO as



(a) vs. Optimal Policies



(b) Impact of VM Types

$2(r_{ave} - \bar{r})$¢, where r_{ave} is the actual average response time after jobs are allocated and scheduled on VMs.

4) **Percentile Guarantee (Percentile)**: This SLO mandates that an α fraction of jobs complete within r time units. In our evaluations, we used $\alpha = 0.95$ and $r = 10$ minutes. The penalty for violating the requirement was 2¢ for every second beyond the response time guarantee.

B. *Policy Costs*

To determine the effectiveness of policy synthesis, we compared the resource allocations produced by Pennyworth with optimal solutions and heuristics that have been shown to be good policies.

WHAT IS PENNYWORTH'S OPTIMALITY GAP? The resource allocation problem is NP-Hard so we cannot compute optimal solutions for all problem instances. Nevertheless, we can use exhaustive search to identify optimal allocations for small problem instances. To find optimal solutions, we assumed "perfect" knowledge of job execution times although we did not make this assumption when applying Pennyworth's policies (where we used estimates). We used problem instances with 32 jobs (queries) distributed over 10 tasks (TPC-H query templates). We compared the cost of the optimal resource allocation with the cost of Pennyworth's policies (Figure 4a) and found that Pennyworth was within 9% of the optimal cost for all SLOs. The results were similar for smaller job sets, even with more restrictive or relaxed SLOs.

HOW DOES PENNYWORTH ADAPT TO A MULTIPLICITY OF VM TYPES? To assess Pennyworth's synthesis, we provided it with the option of using multiple virtual machine types. In our experiments, we used three types of AWS EC2 instances [7]: t2.small, t2.medium and t2.large. The VM types correspond to different amounts of memory and compute power, and these differences result in variations in job execution time depending on the type of instance a job is scheduled on. For instance, jobs that use less memory may perform similarly on all VM instances but a job that needs more memory will perform better on a t2.large instance. The use of different VM types increases the number of edges in the allocation graph, nevertheless Pennyworth was able to identify good allocations. We found that the cost of an allocation was, on average, no more than 7% more than the optimal allocation (Figure 4b). Further, allowing more VM types strictly improved the costs. The experiments using multiple types of VMs indicates that Pennyworth can exploit the choices available and generate policies that lead to good ("low cost") job allocations.

C. *Overheads*

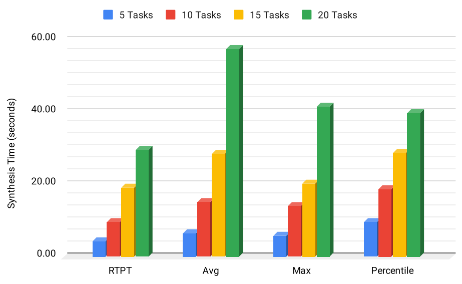
POLICY SYNTHESIS Pennyworth's policy synthesis phase is largely offline. The policy synthesis involves: (i) generating examples and (ii) constructing RAD trees. The time overhead associated with policy synthesis depends on the number of tasks that may need to be allocated and the number of VM types (bin types) we are allowed to choose from.

When we kept the number of VM types to 1 and varied the number of tasks, we found that Pennyworth could synthesize policies in no more than 1 minute with 20 different tasks, with much of the time being spent in generating example problems and solutions to the example problems. When the number of possible tasks is small (5 tasks), Pennyworth could synthesize polices in a few seconds. Recall that the number of tasks is not directly related to the number of jobs that Pennyworth can handle; jobs are instances of tasks and we could have many instances of the same task to allocate. The results we report here (Figure 5a) are based on the TPC-H benchmark, and for these experiments we used more than the first ten TPC-H query templates as tasks.

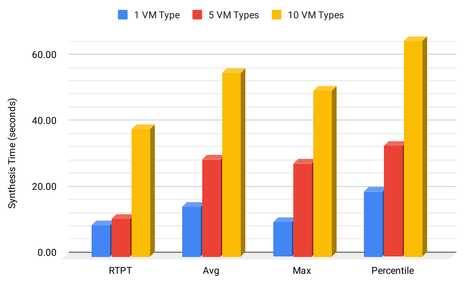
We also evaluated Pennyworth by varying the VM types and holding the number of tasks at 10 (Figure 5b). In this experiment too, we found that policy synthesis times ranged from a few seconds to a minute. (For this experiment, we used additional AWS EC2 instances as possible VM types: t2.micro, t2.xlarge, t3.micro, t3.small, t3.medium, t3.large, m5.large.) *Our conclusion is that, because policy synthesis is performed offline, Pennyworth can generate resource allocation policies within usable time frames.* From a cost perspective, the most expensive policy to synthesize cost no more than 20 cents.

D. *Efficiency of Bulk Allocation*

When used for scheduling a batch of jobs, Pennyworth's policies lead to good outcomes (Section V-B) and our evaluation also found that the policies generated are runtime-efficient: for a batch of up to 30,000 jobs, Pennyworth's policies can complete the resource allocation within 2 seconds. The RAD trees are not very deep (the tree height is typically less than 20) so decisions per job can be made quickly. If h is the height of the RAD tree and a job set contains N jobs, the



(a) Impact of Number of Tasks



(b) Impact of VM Types

Fig. 5: Pennyworth's Time Overheads

runtime complexity of allocating jobs is $O(Nh)$. After a policy has been synthesized, the allocation times do not depend on the number of tasks or VM types.

VI. Related Work

Much of the work in the area of resource allocation and scheduling takes the form of analysis. For a specific problem, an optimal policy is presented with proofs of correctness and runtime analysis. When optimal policies are impractical (because the underlying problem is NP-Hard), approximation algorithms are derived, or heuristics are presented (with some analysis or empirical evidence). The process of arriving at suitable policies, whether they be optimal or whether they be heuristics, is usually a combination of insight, experience and trial-and-error. As examples from a specific domain, work by Liu et al. [8] and Lang et al. [9] present approximation algorithms or frameworks for policy exploration to maximize profit (for a service provider) while meeting service-level agreements in the database-as-a-service model. In the area of job scheduling for large clusters, Tetris [10] is a framework that balances resource consumption along multiple dimensions to place and schedule jobs on a fixed cluster. Tetris aggregates a set of heuristics to improve cluster scheduling but does not explicitly address server provisioning or cost functions of the nature that we consider. Tetris is primarily focused on reducing makespan or maximizing throughput.

What we have sought to do is *synthesize policies* from examples of solutions to specific problem instances, which is a systematic computational approach. In this regard, our work is related to the work on *program synthesis*. Program synthesis is concerned with deriving computer programs automatically given a specification for the program, and there is a significant body of work that relates to this topic [11]. One approach to synthesis is through the use of input-output examples where a program's behaviour is specified by a set of input-output pairs and then the program is synthesized. This approach is exemplified by the FlashFill feature in Microsoft Excel where a formula is derived given the inputs and outputs [12]. Following the success of work on formula synthesis, the idea of program synthesis from examples has been extended to more sophisticated problems such as the generation of a context-free parser from input-output examples using deep learning as the underlying method [13]. Our view is that a resource allocator is a particular type of program, and that we can synthesize such programs from examples of allocations. Work by Mok, Tsou and de Rooij [14] as well as Altisen et al. [15] has explored scheduler synthesis for jobs with timing constraints. Such work uses techniques like constraint satisfaction (Mok et al.) or Petri Nets (Altisen et al.), which run into scalability bottlenecks. Moreover, such prior work, is better suited to deriving policies when constraints are hard (deadlines *have to* be met) but not when constraints are related to mean response times or tail latency guarantees.

Other work, such as such as work by Nguyen and Pingali in the context of parallel programs [16], does not explicitly synthesize new scheduling policies but involves selecting good policies from a portfolio or performing a composition over pre-defined polices. Carastan-Santos and de Camargo use machine learning to assign priorities to jobs in a high-performance computing setting [17]. Their work is similar to ours because it brings together statistical learning and scheduling. Their problem formulation is limited to priority assignment in HPC applications but there may be opportunities to integrate that insight with our work. Decima [18] uses deep learning to learn scheduling policies for data processing clusters and may be a promising approach to achieve goals similar to Pennyworth but Decima's design goals are very similar to Tetris's [10], and therefore server provisioning and alternative objectives are not addressed. A survey by Goodarzy et al. discusses resource management techniques for cloud computing that use machine learning and provides a snapshot of the ideas that have been explored so far [19]. Pennyworth fills a gap in the problem space. The work we have presented builds on the insights from program synthesis and seeks to extend the capabilities of scheduler synthesis to more general settings.

VII. Conclusions

Thinking about a resource allocator or scheduler as a program allows us to use ideas related to program synthesis. We have shown that it is possible to synthesize resource allocation policies using examples of optimal allocations. To obtain optimal allocations we, generally, have to solve NP-hard problems but we can obtain policies that are *good enough* using a relatively small number of examples. Our evaluation of this approach indicates that we obtain policies that are both effective (solutions are near optimal) and efficient (the cost of synthesis and policy use is practical). We reiterate that although the TPC-H queries, used for a majority of our evaluations, have execution times that are of the order of minutes, Pennyworth is not sensitive to the scale of the execution times (offline A* search would have the same running time whether the jobs have execution times in the order of minutes or in the order of milliseconds) except when performing online re-synthesis.

We use decision trees in our policy synthesis step and we find that this model is sufficient for our purposes. The advantage of this choice is that we can explain the policies that we obtain. For other workload models, such as those that involve recurrent tasks, we can synthesize policies for different task combinations up-front, thereby limiting the need for re-synthesis at run-time.

A challenge for program synthesis using input-output examples, and generally for the use of supervised learning, is the need for suitable training data. *For policy synthesis such as what we have described, one can generate a suitable training data set by expending computation power*; this method is usually less expensive compared to requiring human intervention in curating a training data set.

We have only scratched the surface of the research subfield of using statistical learning for synthesizing resource allocation policies. We intend to extend our work to handle online job arrivals and we will analyze the impact of errors in execution time measurements. We will also study the effectiveness of Pennyworth when applied to other scheduling problems. *Our approach works well with "soft" constraints coupled with the type of [continuous] optimization objectives that we have considered.* To synthesize policies when the objective function has discontinuities, we will need additional machinery. One possible argument against such policy synthesis is that we cannot derive performance bounds. This is not completely true: given the use of decision trees, if worst-case execution time estimates are available then *we should be able to identify the worst-case behaviour of these policies* because such bounds can be obtained by simple analysis of the decision tree. We see this as work for the near future (along with extensions to handle preemptive scheduling). Pennyworth's policies have good statistical properties; combining those guarantees with worst-case bounds would improve decision making.

Pennyworth does need example workload to synthesize policies. Obtaining example workload may be a challenge on occasion but traces are often available and the success of techniques such as Decima [18] suggests that this is a viable direction. Our analysis of robustness also indicates that variations between the examples used for synthesis and workload encountered during deployment can be tolerated. The ability to automatically identify good policies for a few cents allows for better use of (more expensive) human engineering time.

References

[1] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on System Science and Cybernetics*, vol. 4, July 1968.

[3] Amazon.com, "Amazon Web Services." http://aws.amazon.com/.

[4] Transaction Processing Performance Council, "TPC-H Decision Support Benchmark." http://www.tpc.org/tpch/.

[5] PostgreSQL Global Development Group, "PostgreSQL." https://www.postgresql.org/.

[6] E. Frank, M. A. Hall, , and I. H. Witten, "The WEKA Workbench." https://www.cs.waikato.ac.nz/ml/weka/.

[7] Amazon.com, "AWS EC2 Instance Types." https://aws.amazon.com/ec2/instance-types/.

[8] Z. Liu, H. Hacundefinedgümüş, H. J. Moon, Y. Chi, and W.-P. Hsiung, "PMAX: Tenant placement in multitenant databases for profit maximization," in *International Conference on Extending Database Technology (EDBT)*, pp. 442–453, 2013.

[9] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan, "Towards multitenant performance SLOs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 6, pp. 1447–1463, 2014.

[10] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proceedings of ACM SIGCOMM (SIGCOMM)*, August 2014.

[11] S. Gulwani, O. Polozov, and R. Singh, *Program Synthesis*. Now Publishers, 2017.

[12] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan 2011.

[13] X. Chen, C. Liu, and D. Song, "Towards synthesizing complex programs from input-output examples," in *International Conference on Learning Representations (ICLR)*, May 2018.

[14] A. K. Mok, D.-C. Tsou, and R. C. M. de Rooij, "The msp.rtl real-time scheduler synthesis tool," in *IEEE Real-Time Systems Symposium (RTSS)*, p. 118, 1996.

[15] K. Altisen, G. Gossler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine, "A framework for scheduler synthesis," in *IEEE Real-Time Systems Symposium (RTSS)*, pp. 154–163, Dec 1999.

[16] D. Nguyen and K. Pingali, "Synthesizing concurrent schedulers for irregular algorithms," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS XVI, pp. 333–344, 2011.

[17] D. Carastan-Santos and R. Y. de Camargo, "Obtaining dynamic scheduling policies with simulation and machine learning," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[18] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of ACM SIGCOMM (SIGCOMM)*, August 2019.

[19] S. Goodarzy, M. Nazari, R. Han, E. Keller, and E. Rozner, "Resource management in cloud computing using machine learning: A survey," in *Proceedings of the International Conference on Machine Learning and Applications (ICMLA)*, 2020.