# IMITA: Imitation Learning for Generalizing Cloud Orchestration

Kamal Hakimzadeh[†], Patrick K. Nicholson[‡], Diego Lugones[‡], Amir H. Payberah[†]
[†]KTH Royal Institute of Technology, Sweden
[‡]Nokia Bell Labs, Ireland
[†]{mahh,payberah}@kth.se [‡]{pat.nicholson,diego.lugones}@nokia-bell-labs.com

***Abstract*—Operating large scale and feature-rich applications is becoming increasingly complex as engineers need to deploy highly configurable software releases on distributed cloud stacks while managing ever-shorter production cycles. Although recent proposals attempt to streamline cloud resources orchestration, there is still a significant challenge in making such solutions generalize to unseen cloud stacks. In other words, the behavior of application-specific Key Performance Indicators (KPIs) and resource configurations, crafted for specific stacks, may differ on heterogeneous deployments, requiring time-consuming policy adjustments. We introduce IMITA, a system that leverages *imitation learning* to create models by imitating an expert behavior that can be generalized seamlessly to new cloud stacks. To make a generalized model, IMITA maps expert actions taken based on the application KPI space to the space of resource utilization metrics that are universally available in cloud platforms. This mapping enables the model to trigger actions, mimicking expert behavior, upon the occurrence of similar resource utilization footprints across deployments. We demonstrate IMITA by learning to scale-out Cassandra deployments with diverse configurations and workloads. Our results show IMITA can replicate expert actions across deployments and extrapolate to unseen environments by achieving $50 - 94\%$ fewer false positives actions than traditional threshold-based policies while still adhering to Service-Level Objectives (SLO) and avoiding under-provisioning of resources. Moreover, since collecting data in clouds is costly, IMITA gathers data only for representative configurations to train the imitator model. This approach reduces the size of the collected data to $50\%$.**

***Index Terms*—Cloud Orchestration, Imitation Learning, Generalization**

## I. Introduction

Large companies have drastically changed how software functionality is deployed and offered to customers. Nowadays, new software features and product releases are delivered on a daily basis, or even more frequent [1]. The change towards continuous software development is creating significant challenges in engineering operations as the pace of deployment reaches unprecedented speed and scale [2]. As a consequence, many companies create specialized teams to design strategies for configuring, deploying, and maintaining cloud services at scale [3], and also to collaborate with developers in deploying new software functionality to large scale production environments (cloud environments) [4]–[6].

Whenever any change occurs across these distributed deployments, it is required to tune plenty of configuration parameters. However, due to the complexity of cloud stacks, this is a time-consuming process. These changes can be caused by workload variations, containers or virtual machine modifications, new software features, or underlying middleware updates, to name a few. These changes usually require manual adjustments that limit the frequency at which a production system can be adapted. The need for human expertise is mainly because applications have different cloud stack-specific configurations and Key Performance Indicator (KPI) requirements, which may not be directly applicable in another setting, given the multiplicity of variables that affect the application performance.

Recent researches show that Machine Learning (ML) is a promising approach to reduce human intervention in *orchestrating* cloud stack [7]–[9]. For example, [10], [11] present ML-based application-agnostic solutions to orchestrate applications effectively for a given set of resource utilization metrics and application KPIs. However, these solutions require long offline training phases to create accurate performance models. Another group of solutions opts to calibrate the systems by monitoring the applications executions online [12]–[14]. Nevertheless, this approach requires a rich representative set of expert actions to train the models.

Both of these ML-based solutions suffer from the same problem; that is, the constructed models are implicitly tied to the environments from which the training data is collected. Thus, it is infeasible to use them directly in new environments without either retraining them with data from the new environment or modifying them on-the-fly using such data. As an example, consider an *autoscaling* system configured to take action based on throughput exceeding a certain threshold. Any change in the underlying container resources, such as increasing the number of available cores and memory, can render the initial threshold sub-optimal.

In this paper, we introduce IMITA, a solution that leverages *imitation learning* (IL) [15] to train a generalized model, called *imitator*, by associating expert actions (either human-based or automated) with resource usage footprints, which are generic platform metrics (e.g., CPU and memory utilization, I/O rate, etc.) universally available in cloud stacks. The imitator, then, mimics the expert actions in new environments, even if the application-specific KPIs behave very differently. IMITA is a system that (i) automates and reduces the calibration and training phases by learning how to imitate expert actions in scenarios and configurations that are different from those known by the expert, and (ii) incorporates IL techniques for automating data collection across cloud stacks.

To demonstrate IMITA, we create a software environment that enables expert mechanisms to precisely annotate their actions with respect to underlying application KPIs and resource utilization metrics. Therefore, IMITA learns the sequence of actions taken by the expert to keep application KPIs in

the desired state. These actions are correlated to resource utilization footprints; thus, the imitator can model the expert actions with respect to normalized resource utilization rather than application-specific KPIs. Since the imitator encodes expert actions regarding resource usage (learned across a variety of representative environments), it is generic enough to perform in new unseen environments accurately.

We evaluate IMITA in an autoscaling scenario of a real setup of Apache Cassandra [16] in a cluster, and we train and test the imitator on different environments. We show that IMITA's data collection heuristic reduces the amount of required data for training accurate and robust models (i.e., imitators) to $50\%$ compared to the random heuristic as the baseline. Moreover, we show that IMITA trains the imitator using only $22\%$ of a given configuration space; however, it can effectively extrapolate to unseen configurations with $F1$-score greater than $0.8$ for the majority of the space. The results also show that the trained imitator prevents resource saturation. Furthermore, the imitator makes $50\% - 94\%$ less false-positive (that causes over-provisioning) compared to threshold-based policies.

In summary, the contributions of this work are:

- IMITA, an IL-based solution that provides a systematic method for mimicking expert's actions in cloud orchestration and generalizing their capabilities to new deployments.
- A novel data collection heuristic to reduce the time required to find a set of representative stack/workload configurations to train the model. An exhaustive exploration of the entire configuration space is infeasible; thus, our heuristic tries to reduce the amount of data collected. To do so, we use distance metrics over the configuration and resource utilization spaces and take only the most difficult cases to learn as samples.
- A software environment as a sandbox for data collection, elasticity-aware workload generators, and executing controllers.

## II. Background

In this section we first briefly explain the autoscaling function as a well-known use case of cloud orchestration, and then recall some basic concepts from IL and show how it can be used in autoscaling operations.

### A. Autoscaling

*Autoscaling* is a use case of cloud orchestration that we built our system upon. However, our proposed solution is general enough to be applied in other use cases, such as migrating virtual machines across physical machines because of noisy neighbors. An orchestration system can be modeled as a *feedback control loop*, where the autoscaling function, as a *controller*, scales out the resources of the *target system* (i.e., by increasing the number of servers), when the performance of the system exhibits a bottleneck. The controller continuously gives control input to the target system, and based on the receiving *feedbacks* (e.g., KPI, utilization, etc.) from the system, it scales the resources. The feedback determines if the target system's performance deviates from the defined Service Level Objective (SLO). If the performance goes below the defined threshold, due to any bottlenecks, then the controller takes action to resolve the deviation.

Performance bottlenecks can stem from different sources, such as high resource utilization (a.k.a *saturation*), interference, deadlocks, failures, or any other anomalies. In this work, we focus on resource saturation as a source of bottlenecks, but any of the aforementioned reasons can also be considered. We define the *saturation footprint* as the normalized values of a set of utilization metrics when the application resources are saturated. Saturation can occur in a single resource or multiple resources [17]–[19]. Single resource bottlenecks simply can be identified by analyzing sub-linear behaviours [17]; however, this approach does not scale in multiple bottlenecks in large systems with dozens or hundreds of metrics. Therefore, we consider a data-driven approach to identify saturation footprints regardless of their type. Our goal is to identify as many saturation footprints as possible, given constraints on data collection time.

### B. Imitation Learning

*Imitation Learning* (IL) is a family of ML algorithms, in which a learning model (*imitator*) tries to learn a *policy* $\pi$ by mimicking an *expert* policy $\pi^*$ in the environment [15]. The environment is a set of *states* $S$, such that in each state a number of *actions* $A(S)$ are feasible, and the policy $\pi$ is a function from a state $s \in S$ to an action $a \in A(S)$. At each point in time $t$, the imitator receives a state $s_t$, and takes an action $a_t$, i.e., $a_t = \pi(s_t)$. A sequence of pairs $(s_t, a_t)$, observed by applying the policy $\pi$ on the states of the target system over time is called a *trajectory* $\tau$. Similarly, we define the *expert trajectory* $\tau^*$, a trajectory generated by the expert policy $\pi^*$, which is considered as a reference behaviour, and consists of a set of pairs $(s_t, a_t^*)$, where $a_t^* = \pi^*(s_t)$.

*Dataset Aggregation (DAgger)* [20] is an IL algorithm to train an imitator by re-training the model iteratively, based on its failures. Algorithm 1 illustrates how DAgger works. It first trains an initial policy (model) $\pi_0$ on a set of the expert trajectories $\tau^*$. To train the policy, we use the learning algorithm $A$ that can be any supervised learning algorithm. Then, in each iteration $i$, the trained policy $\pi_i$ controls the system and annotates the states visited by the model using the expert policy $\pi^*$. As a result, a new policy $\hat{\pi}_i$ is created based

**Algorithm 1:** DAGGER$(\pi^*, A, N)$

```
Train ← ∅
τ* ← GenerateTerajectory(π*)
π_0 ← A(τ*_0)

for i = 1..N do
    π̂_i = β_i π* + (1 − β_i) π_{i−1}
    τ_i ← GenerateTerajectory(π̂_i)
    Train = Train ⋃ τ_i
    π_i ← A(Train)

return best π_i
```

on the trained policy from the previous iteration $\pi_{i-1}$ and the expert policy $\pi^*$. The parameter $\beta$ (Line 5) defines how much the expert is allowed to generate new trajectories. If $\beta$ is one, the expert is in control all the time. Afterward, iteratively a new trajectory $\tau_i$ is generated using the policy $\hat{\pi}_i$, and will be added to the training set. A new model $\pi_i$ is, then, trained on the aggregated trajectories. Finally, after several iterations, the best $\pi_j$ (not necessarily the last one) is returned, based on its performance on the entire aggregated set of trajectories.

## III. Our Algorithm (IMITA)

In this section, we explain IMITA, our proposed IL algorithm for autoscaling problems.

### A. Autoscaling with Imitation Learning

IMITA incorporates IL to the autoscaling control loop. We aim to train the imitator to behave like the expert when the target system runs in new deployments. In this case, a state $s \in S$ consists of a vector of input *feedback metrics* that demonstrates if the target system is saturated, and action $a \in A(S)$ is a *resource scaling* action to ensure the target system has enough resources for its tasks.

IMITA considers two types of feedback metrics: (i) $y_p$, the application-specific KPIs, which is given as input to the expert, and (ii) $y_z$, the resource utilization metrics, which is given as input to the imitator. We define the expert's actions $a^* \in \{True, False\}$ as a binary value to identify saturation footprints based on the performance feedback $y_p$. In other words, the expert policy $\pi^*$ is a mapping from $y_{p_t}$, at time $t$, to a binary action $a_t^*$ as an indication of saturation. We also consider two types of actions for the imitator: (i) *actuation*, denoted by $U^{+1}$, to add more resources to the system, and (ii) *no-actuation*, denoted by $U^0$, to do nothing, i.e., $a \in \{U^{+1}, U^0\}$.

The trajectories are collected in a controlled environment; thus, if a bottleneck occurs at time $t$, the resource utilization $y_z$ is mapped to an intended expert action $a_t^*$. Thus, given $a_t^*$, we define the imitator policy as:

$$\pi(y_{z_t}) = \begin{cases} U^{+1}, & \text{if } a_t^* = True \\ U^0, & \text{if } a_t^* = False \end{cases}$$

Our aim in IMITA is to find the policy $\pi$ to associate resource utilization $y_z$ to scale-out actions, either $U^{+1}$ or $U^0$, whenever saturated footprints are observed in run-time. This approach's benefits are relieving cloud operators from frequently configuring software stacks per application separately while producing accurate saturation examples.

### B. The IMITA Algorithm

IMITA is a novel IL algorithm, inspired by DAgger [20], for the autoscaling problem. IMITA iteratively collects data, aggregates them, and re-trains the model on the entire collected dataset. We have an expert (either human-based or automated) in the control loop during the training that we can query in all the collected states in each iteration. However, after the training, we have no access to the expert. To collect more effective samples in the training phase, we focus on collecting additional data only for the *difficult states* that cause the model to fail at its task. The intuition is that the training on the difficult states will lead to a more robust model within fewer iterations compared to the random training. This is reminiscent of existing re-weighting techniques for boosting, as we sequentially add difficult trajectories to our training set [21].

IMITA slightly differs from DAgger, that is, in DAgger, the model policy $\pi$ controls the system by observing states and taking actions, and the expert policy $\pi^*$ is used to annotate incorrect actions taken by the model policy. In IMITA, however, the expert policy is used for demonstrating the right actions of a trajectory in the training process, but the model policy is used to forecast which unexplored configurations are the most effective ones (i.e., the configurations that the model policy produces the most number of wrong actions) to explore in subsequent training. This procedure is formalized in Algorithms 2 and 3.

Algorithm 2 (the *boosting heuristic*) effectively explores the environment configuration space. As input, it gets (i) $\pi^*$, the expert policy, (ii) $A$, a supervised learning algorithm, (iii) $C$, a set of possible environment configurations, (iv) $\theta$, the minimum target score, (v) $\Delta$, a threshold on the difficulty of a trajectory, and (vi) $N$, the total number of iterations

**Algorithm 2:** BoostingHeuristic$(\pi^*, A, C, \theta, \Delta, N)$

```
1  Train ← ∅
2  c ← PopFirst(C)
3  for i = 1..N do
4      τ_i^* ← GenerateTerajectory(π^*, c)
5      Train = Train ⋃ τ_i^*
6      π_i ← A(Train)
7      δ ← CalcDifficulty(A, τ_i^*, θ)
8      if δ ≤ Δ then
9          c ← PopFurthest(C, c)
10     else
11         c ← PopNearest(C, c)
12 return best π_i
```

**Algorithm 3:** CalcDifficulty$(A, \tau, \theta)$

```
1  len ← ∞
2  Train ← ∅
3  D : ⟨d_j⟩_{j=1}^{|τ|} ← τ
4  for each d ∈ D do
5      Train ← {d}
6      sc_min ← 0
7      while sc_min < θ and D \ Train ≠ ∅ and |Train| < len do
8          π ← A(Train)
9          d̂ ← arg min_{i∈D\Train}(score(π, i))
10         sc_min ← score(π, d̂)
11         Train ← Train ⋃ {d̂}
12     if |Train| < len then
13         len ← |Train|
14 δ ← len/|D|
15 return δ
```

(the number of trajectories to explore). First, the POPFIRST function returns a configuration from the extreme points of the available environment configuration space (i.e., minimum or maximum settings from each dimension of configuration space). Then, for the selected configuration $c$, we call GENERATETERAJECTORY that queries the expert to collect data and make the expert trajectory $\tau_i^*$ in each iteration $i$ (Line 4). The expert trajectory $\tau_i^*$ consists of tuples of states (i.e., both performance $y_p$ and utilization metrics $y_z$) and expert actions $a^*$. We then append $\tau_i^*$ to the list of training trajectories (Line 5), and retrain the imitator's policy $\pi_i$ on the aggregated data (Line 6).

Next in Lines 7-11, first we invoke CALCDIFFICULTY, (described in Algorithm 3) to estimate the difficulty of trajectory $\tau_i^*$ collected by the expert. If $\tau_i^*$ is not difficult enough (i.e., its difficulty metric is less than the defined threshold $\Delta$), then we call the POPFURTHEST to get a new configuration that is estimated to be *highly distant* (we can use different distance metrics, but in our experiments we use Manhattan distance - see Section V) from the explored configurations $c$. Otherwise, we call POPNEAREST to get a configuration that is *near* to the current configuration $c$. Then, we proceed to the next iteration using this new configuration. Finally, after $N$ iterations, we return the best imitator $\pi$.

The goal of Algorithm 3 is to indicate the *difficulty* of the input trajectory $\tau$. For analysis purposes, we decompose trajectories into a set $D$ of disjoint *sub-trajectories* $d$, such that each sub-trajectory represents a maximum trajectory subsequence, where the system stays in the same state. To be more precise, a sub-trajectory consists of a sequence of any number of pairs with $U^0$ actions, followed by a single pair with $U^{+1}$ action. We can compute the difficulty of a trajectory by finding the smallest set of its sub-trajectories, such that if a model trained on them and tested on the rest of the sub-trajectories, the accuracy metric ($F1$-score in our work) becomes greater than $\theta$. The difficulty is then defined as the ratio of the number of sub-trajectories in the training-set over the total number of sub-trajectories in $D$.

Algorithm 3, as input, gets (i) $A$, a supervised learning algorithm, (ii) $\tau$, a trajectory, and (iii) $\theta$, a scoring threshold, and returns $\delta$ as the difficulty of $\tau$. We first create $D$ as the list of sub-trajectories of $\tau$, and then iterate through all the sub-trajectories in the outer-loop in the algorithm (Line 4-13). For each sub-trajectory $d$, we train the model $\pi$ using $d$, and test it on the rest of the sub-trajectories one by one. We then select the sub-trajectory $\hat{d}$ that generates the lowest score and append it to the training set $Train$. The lowest score indicates that $\hat{d}$ is the most difficult sub-trajectory among the rest of the sub-trajectories. Thus, including it into the training set means that we try to narrow down our training to use only the most difficult sub-trajectories, rather than all. In Line 14 we return the difficulty value $\delta$ as the ratio of the size smallest training set over the total number of sub-trajectories.

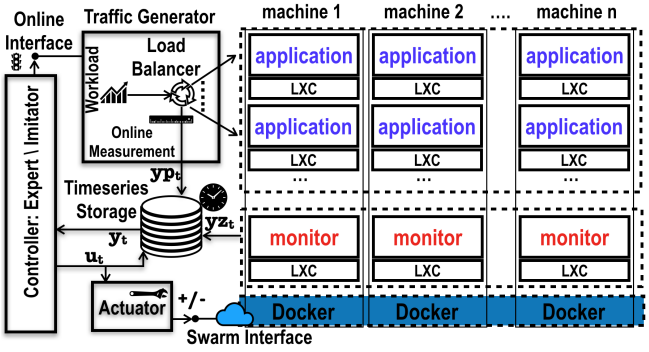


Fig. 1: The architecture of the deployment environment. In this framework, IMITA controls the applications (e.g., Cassandra) running in LXCs.

## IV. IMPLEMENTATION

In this section, we describe how we use IMITA to create an imitator for autoscaling an Apache Cassandra cluster [16], as a use case. Below, we first describe the use case implementation and then present how the expert indicates the minimum requirements to run the target application and generates the trajectories.

### A. Use Case

As a use case, we show how IMITA manages a cloud stack for Apache Cassandra [16]. Our design objective is to provide a framework to facilitate training a generalized model and testing it across different deployments. The expert in this use case is a meta-algorithm that describes how one would manually learn when to take action for given application deployment.

*Architecture and Technologies:* Figure 1 illustrates the deployment environment, where we use to generate the expert trajectories and to test the trained imitators. In our design, we can launch various applications (e.g., Cassandra, Hbase, etc.), with variable amounts of computing resources (e.g., 2-8 CPUs 1-8 GB RAM), and a configurable number of instances for running applications (e.g., 1-20 Cassandra servers). The applications run on Linux Containers (LXC) for lightweight management and a portable orchestration. We use Docker as a management layer for containers and Docker Swarm for treating all machines as one cluster of resources [22]. We also rely on *control groups (cgroups)* [23] to isolate resources (e.g., CPU or memory) allocated to containers. The cgroups performs a fine-grained resource allocation and controls containers not to interfere with each other.

We use *cadvisor* [24], an LXC-based monitoring agent, per machine to collect the utilization metrics $y_z$ at the container level. The performance metrics $y_p$ (i.e., throughput and latency) are also measured and collected by the traffic generator's load-balancer (explained in the next sub-section). The traffic generator allows online control of traffic intensity. Both $y_z$ and $y_p$ are stored in Prometheus time-series database [25]. At each control interval, the controller queries the suitable feedback metrics ($y_p$ or $y_z$) from Prometheus

to make scaling decisions. During the training sessions, the expert controls the traffic generator in order to produce representative training patterns. Expert decisions are enforced by an actuator that uses the Docker Swarm to control the number of instances. It also allows changing the system configuration parameters, such as cool-down time, concurrent scaling, etc.

*KPI Metrics $y_p$:* We take advantage of a traffic generator to (i) generate data labeled by the expert, (ii) test the system in a controlled environment, and (iii) characterize the performance metrics $y_p$ under heavy loads. In this deployment, we use *Yahoo! Cloud System Benchmark (YCSB)* [26] as the traffic generator with key adaptations to allow it to scale with the application. YCSB generates high-intensity traffic with accurate precision for task assignment among running threads.

The existing implementation of YCSB does not scale to multiple instances, and it can only run workloads against applications with a fixed setup during the execution of a workload. Moreover, it only outputs the calculated performance statistics when the workload is finished. Therefore, to have a fine-grained evaluation, we need to add a few features to YCSB to work with elastic systems. To do so, we modified YCSB, such that it can reproduce time-varying traffic intensities from external sources to stress the application. Synthetic intensity functions and real-world workload traces can be executed remotely by the controller. Moreover, our modified YCSB can measure the performance statistics in configurable time-windows.

As the performance metric $y_p$ we consider two metrics: (i) *throughput*, the number of the *started*, *finished*, and *failed* operations for different operation types in YCSB (i.e., `READ`, `UPDATE`, `INSERT`, and `DELETE`), and (ii) *latency*, various percentiles of the latency per operation. If $P_k$ denotes the $k$th percentile, we collect $P_k$ for $k \in \{25, 50, 75, 90, 99, 100\}$ of latency per operation. At run-time, the traffic generator pushes micro-batches of calculated throughput and latency metrics to our time-series storage. Thus, in summary, we collect and process three throughput metrics (i.e., for started/finished/failed operations), and seven latency metrics (i.e., $P_k$) per operation, and given four query operations in YCSB, in total we have $(3 + 7) \times 4 = 40$ metrics as $y_p$.

*Utilization Metrics $y_z$:* We use the cadvisor monitoring tool [24] to collect $44$ utilization metrics $y_z$: $10$ CPU-related, $7$ memory-related, $17$ filesystem-related, and $10$ network-related metrics per container. These metrics should be pre-processed in order to be used by the controllers. First, we normalize them by the total amount of available resources in the application container. For example, CPU metrics are normalized to the number of CPU cores in the container. We do the normalization when the metrics are fetched from Prometheus [25]. Here, we use the rate of values instead of the absolute values as they preserve an *indication of trend* in the final magnitude. The indication of trend is a metadata that allows the model to distinguish between the same absolute values occurring in increasing and decreasing traffic trends. In the end, the pre-processing pipeline aggregates each container's metrics into a single vector by summarizing them using seven quantile measurements, i.e., $P_k$ for $k \in \{0, 25, 50, 75, 90, 99, 100\}$. Thus, each controller receives $y_z$ as a vector with $44 \times 7 = 308$ metrics.

### B. Indicating the Minimal Requirements

To train an imitator, IMITA first requires the expert intervention to determine the minimum system requirements for running the target application. We need to conduct this task only once per application; thus, it is not a time-consuming task in the whole process. In our use case, we achieve this by exploring the configuration space for CPU intensive workloads. We empirically find the minimal container requirements for Apache Cassandra to work without failures and then verify if horizontal scaling relieves resource saturation for such containers in the given workloads.

If Apache Cassandra is under-provisioned, then two types of errors are dumped into the YCSB log: errors due to (i) connection failure, and (ii) timeouts. So, to better understand the system behavior, we include these errors as additional metrics to YCSB and measure the correlation between them and resource utilization. We figure out that these errors are correlated to the initial caching and the amount of data in the Apache Cassandra database. Cassandra rejects connections when the caching effect begins. The duration of the effect is proportional to the amount of data in the database.

To overcome this problem, we decrease the amount of stored data, as long as the system is CPU-bounded, and set the minimum container's RAM to $3GB$ to remove timeouts and excessive memory swaps. The reason we focus on CPU is that the horizontal scaling is not able to resolve resource saturation involving RAM limitations. After running the experiments, we find out the minimum CPU-bounded container flavor requires $2$ CPUs and $3GB$ of memory for $40000$ randomly generated database rows, each 1KB (the YCSB's default setting).

Given this flavor, we inject an increasing ramp-shaped workload (with $100\%$ `READ` operations) and reach a throughput bottleneck. We repeat the same experiment for clusters with more nodes (horizontal scaling) and observe that the saturating throughput increases with the number of nodes, though not always monotonically. The average latency of reads is around $1$ms in unsaturated states, but it increases to $100$ms near saturation. We also scale the system vertically by adding more CPUs in each container and increasing it to $4$, $8$, and $16$. We observe that experiments with $4$ and $8$ CPUs give higher throughput, while the $16$ CPUs flavor does not provide any noticeable benefit compared to the $8$ CPUs flavor for our workloads. In all these experiments, the requests are uniformly distributed over the records in the database.

### C. Generating Expert Trajectory

We set up the training sessions based on the trajectories collected using the boosting data collection heuristic (Algorithm 2). In most of the trajectories, the fraction

of time that the system is not saturated is much larger than in saturation because the expert takes action to avoid resource saturation. However, the lack of enough training data exhibiting saturation is highly problematic for the learning methods, as it produces a class imbalance in the training examples. To circumvent data skewness, we let the expert controller manage the training sessions for detecting performance degradation. So, the expert injects traffic in such a way to avoid data skewness. The expert splits the training session into a sequence of two-phases: *saturation identification* and *data sampling*.

*Saturation Identification:* The purpose of this phase is to estimate the capacity of the current system configuration so that the expert can gather relevant data during the subsequent data sampling phase. To perform saturation identification, the controller saturates the application for several epochs, with a constant and high traffic rate that exceeds the limit that the system can serve. Then, it calculates the capacity of the system over that window.

*Sampling Phase:* The sampling phase follows saturation identification without changing the system settings. The expert leverages the computed system capacity from the first phase to adjust the range of traffic used in this phase. More precisely, it creates a ramp traffic starting from 0, and gradually increases to twice the computed capacity value at a constant rate until when the system becomes saturated. The expert, then, labels all samples that occur after the first time the system exceeds the estimated capacity as saturated, and all samples prior that as unsaturated. The ramp traffic modeling and labeling by the expert allow generating a balanced set of samples of both saturated and unsaturated states. In our experiments in Section V, the expert generates 200 samples, such that 100 samples are taken in the saturated state, and 100 samples are in unsaturated.

## V. Evaluation

In this section, we evaluate the performance of IMITA and present the experimental results for synthetic and real-world workloads. In particular, we answer the following questions: (i) how much data should IMITA collect in order to achieve a certain level of accuracy?, (ii) to what extent can IMITA generalize models to unseen environments?, and (iii) how does the imitator perform as an elastic controller?

For all experiments, we use our sandbox system (Figure 1). We deploy the sandbox on a cluster of seven servers, each with

TABLE I: Exploration space for generating trajectories.

| Dimension | Range |
|---|---|
| Container Flavor | $\{C_2, C_4, C_6\}$, where $C_i$ has $i$ cores, and 8GB RAM |
| Workload Mix | {100R (or R), 50R, 0R (or U)} where $x$R is $x$% READ, $(1-x)$% UPDATE |
| Number of Nodes | [1-20] |

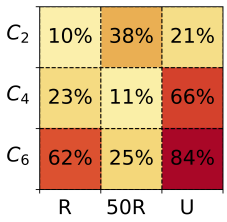


Fig. 2: This difficulty matrix shows the values of the difficulty metric for configurations in our configuration matrix according to Algorithm 3.

32-core AMD 6378 processors, $132GB$ RAM, Linux CentOS 7, and a $10Gbps$ network. Containers are launched on all servers that are swarm members, connected through an overlay network in Docker Swarm for container communication. We use Apache Cassandra [16], as the test application in the scaling scenario since it is a representative and widely used distributed key-value store usually offered by cloud providers. As described in Section IV, we use YCSB [26] as the traffic generator. We use Random Forest as the learning algorithm to train the imitator ($A$ in Algorithms 2 and 3), but any other ML algorithm could be used to implement our solution.

The environment configuration space is shown in Table I. The first two dimensions (container flavor and workload mix) define nine distinct configurations ($3 \times 3$) used for collecting nine trajectories. We consider 20 scaling levels, such that each one corresponds to one sub-trajectory; thus, we have $9 \times 20 = 180$ sub-trajectories. Given these sub-trajectories, the expert generates $180 \times 200 = 36000$ samples, where each sample is collected at a 6-second interval. As explained in Section IV-C the 200 samples are generated by the expert, such that the number of saturated and unsaturated samples are equal, i.e., 100 samples. Moreover, the configuration space is defined according to Section IV-B to satisfy the minimum required resources.

To have a concrete definition of *distance* between configurations, we map the configuration space into a two dimensional matrix (as depicted in Figure 2), such that the rows represent different container flavors sorted by number of cores (i.e., $C_2$, $C_4$, and $C_6$), and the columns show different workload mixes (i.e., R, 50R, and U), sorted as in Table I. Thus, we have a $3 \times 3$ matrix with nine cells in total, and we use the Manhattan distance with respect to this matrix to compute the distance between configurations. The value in each cell shows the difficulty value of the trajectory generated on that configuration (as computed in Algorithm 3). For example, as Figure 2 shows, the difficulty of $C_6$_50R is 25%.

### A. *How many subtrajectories must* IMITA *explore?*

The models' training time is affected by several factors, including the size of the exploration space (i.e., configuration space), the variation of saturation footprints inside that space, and the target accuracy. To present how much time is required to explore our configuration space, we need to show how we quantify the different configurations' difficulty.

The difficulty values for the nine trajectories in our configuration matrix is shown in Figure 2. We observe that the difficulty of trajectories increases when we change the configuration from $C_2$ towards $C_6$ (i.e., increasing the number

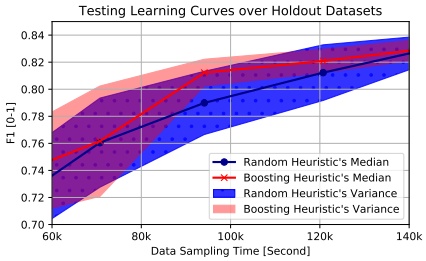


Fig. 3: The learning curve shows that the boosting heuristic compared to the random heuristic achieves higher $F1$-score faster with lower variances.

of cores). Moreover, the trajectories generated in only-update `U` configurations are more difficult than their only-read `R` counterparts. For $C_2$ the read-update mix `50R` is more difficult than the only-read `R` and only-update `U`, whereas in $C_4$ and $C_6$ this is the other way round. Based on our observation, the throughput signal of $C_2$ is less noisy than that of $C_6$ for `R` and vice versa for `U`. The noise in the throughput signal, which is correlated to the throughput volume, also explains the difficulty. We conclude that the amount of required training data depends not only on the size of the environment configuration space, but also on the difficulty of different dimensions.

We showed in Algorithm 2 the boosting heuristic that IMITA uses to collect training data and train the imitator. The data collection heuristic goal is to collect as few samples as possible due to the cost of collecting data in different configurations. We compare the $F1$-score of the model trained using data collected by the boosting heuristic with the model trained using randomly selected samples. Figure 3 shows the result, where the x-axis is the sample collection time, and the y-axis is the $F1$-score of the models on the test set. The test set contains $20\%$ of samples randomly selected from all trajectories, and it is fixed in all the experiments. We repeat the experiment 10 times, and we show the median and variance of the $F1$-scores in Figure 3. Here, we use the $3\times3$ configuration space presented above, in which it takes $36000 \times 6 = 216K$ seconds to collect all samples (considering that each sample is collected at a 6-second interval). We set the accuracy threshold of $F1$-score to be greater than $85\%$ ($\theta = 0.85$ and $\Delta = 0.5$ in Algorithm 2).

In both of the data collection heuristics (random and boosting), initially, the $F1$-scores of the models are low because the size of the training set is small. On the other hand, the $F1$-scores of the models in the final iterations tend to the same value for both data collection heuristics since both of them finally explore the whole configuration space. Thus, in Figure 3 we only focus on where we can see the impact of the heuristics on data collection, i.e., from $60K$, where the $F1$-scores start taking off, to $140K$, where both heuristics start converging again.

As we can see in Figure 3, the variance of the boosting heuristic monotonically decreases: from $0.1$ in $70K$, $0.02$ in $93K$, to $0.015$ in $140K$. In this period, the median $F1$-score of the boosting heuristic is always superior to the random heuristic's median, and its variance is about $1/3$ of

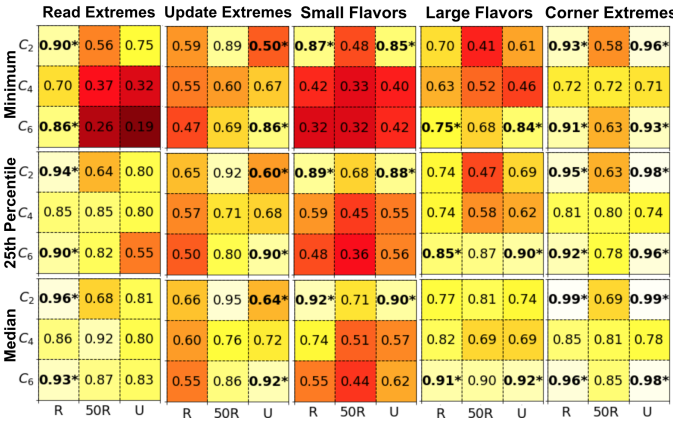


Fig. 4: Each small cell represents an environment configuration, and the cell-value is the $F1$-score of the models trained in that configuration. The bold values highlight the training configurations for that block.

the random's. Moreover, the boosting heuristic achieves $0.81$ $F1$-score with $0.02$ variance by $93K$ seconds, whereas the random heuristic's median gets to $0.81$ with $0.04$ variance by the time of $125K$ seconds. The variance of the random heuristic converges to $0.02$ around $180K$ seconds. Therefore, the random heuristic requires $34\%$ more data to get to the median score, and it needs $100\%$ more data to become robust compared to the boosting heuristic. Practically, the data collection in the boosting heuristic can be stopped after $26$ hours ($93K/3600$). In contrast, the random heuristic needs to collect data $\approx 36$ hours and $\approx 50$ hours for the same median and the same variance, respectively.

### B. How well do models generalize?

We run an experiment to illustrate how well the trained model (i.e., imitator) performs on unseen configurations. As Figure 4 shows, from the nine existing configurations ($3 \times 3$ configuration space), we train the model using only two configurations (shown in bold and with *). The trained model is then tested on the next seven configurations of that block. Figure 4 has five block-columns, shown as Read-Extremes, Update-Extremes, Small-Flavors, Large-Flavors, and Cornet-Extreme. Each of the first four block-columns indicates the configurations used for training the model. For example, in the Read-Extreme, only the configurations with only-read `R` are used for training, and in Large-Flavors, only the configurations with $C_6$ are used for training. In the Corner-Extreme, unlike the first four block-columns, four configurations are used for training. Moreover, given that we collect one trajectory that contains 20 sub-trajectories for a configuration, we present the minimum, the 25th percentile, and the median of $F1$-scores in distinct block-rows separately.

As a general trend, we see that in all the blocks, the further away a cell is located from the training cells (by Manhattan distance), the darker color or, the lower value the cell has: the training cells have the highest $F1$-scores, then their direct neighbors, and the forth. Intuitively, in this two-dimensional configuration space, the model that is trained on a configuration, such as $C_2$_`R`, performs better on a configuration that varies only on one dimension, such as $C_2$_`50R`, than

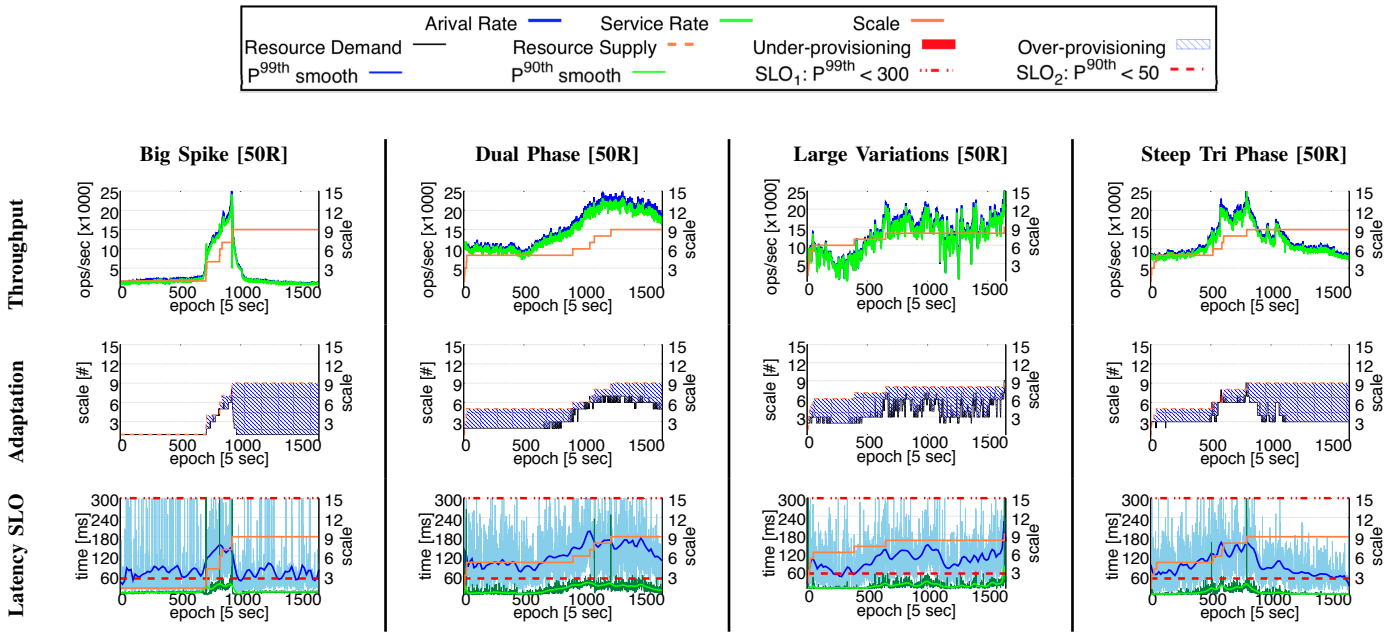


Fig. 5: Testing the imitator on an unseen configuration ($C_6$_50R). We execute real traffic traces from the web-site access dataset for the 1998 World Cup.

another configuration that varies on both dimensions, such as $C_4$_50R. As we change the training configurations in different block-columns, we still observe the same trend regardless of training the model on only-read traffic, only-update traffic, small container flavors, or large container flavors. However, this general trend has some exceptions that are due to different difficulty levels in different configurations (as Figure 2 shows). For example, the top-3 most difficult trajectories, $C_6$_U, $C_4$_U, and $C_6$_R according to Figure 2, also tend to yield low $F1$-scores when they are not part of the training set. The $C_2$_U and $C_6$_U also behave differently, as $C_2$_U has a low $F1$-score even when part of the training set.

### C. How efficient is IMITA as an elastic controller?

To test the imitator under various workloads, we inject traffic traces based on web-site access patterns for the Worldcup98 [27], as these patterns capture important phenomena like traffic surges. We aggregate requests on a one-minute timescale, scale them up to saturate our cluster, and stretch the traces to a time-horizon of 1600 samples. We selected four traces (Big Spike, Dual Phase, Large Variations, and Steep-Tri-Phase) resembling some important patterns presented in the literature [28], [29]. As the test configuration, we choose the 50R workload, and a five cores container, $C_5$. We launch Cassandra with one seed node and one data node per run. Figure 5 shows traffic throughput, resource adaptation, and SLO violations for the mentioned runs.

Each column of Figure 5 shows the system behavior per traffic pattern. The first row compares the request arrival rate against the request service rate, and as we see, the two plots match almost entirely for all the traffic patterns as the imitator scales out the system as required. The second row compares the amount of resource demand against the amount of supplied resource by the imitator. According to a measurement that we carried out based on the elasticity metrics from BUNGEE [30], the imitators have zero under-provisioning on all runs except the Steep-Tri-Phase, which incurs under-provisioning with an average of 2% resource units in the entire run. As expected, since our imitator does not scale-in, there is significant over-provisioning at the end of the run for these traces. Interestingly, in the last three traffic patterns, IMITA appears to over-provision at the start of the trace. This is actually due to non-monotonic behavior in the throughput capacity (for example, the expert finds out that four nodes have lower throughput capacity than three nodes). Therefore, the amount of over-provisioning is not the product of false-positive actions.

TABLE II: The false-positive benchmark of autoscaling policies shows that the imitator makes $50-94\%$ fewer mistakes compared to threshold-based policies over the traffics of Figure 5.

| Policies | Big S. | Dual Ph. | Large Var. | S. Tri Ph. |
|---|---|---|---|---|
| Single Metric | 10 | 37 | 6 | 16 |
| Rule-Based | 18 | 81 | 13 | 31 |
| DT Imitator | 11 | 20 | 10 | 8 |
| RF Imitator | 1 | 6 | 3 | 3 |

TABLE III: Fulfilment analysis of Cassandra's SLO targets when orchestrated with the imitator. We set two target SLOs to obtain 99th and 90th percentile latency of 300ms and 50ms, respectively. We calculate SLO violations, $V_{300}^{99}$ and $V_{50}^{90}$ these SLOs, using the runs in Figure 5.

| Latency SLO | Big S. | Dual Ph. | Large Var. | S. Tri Ph. |
|---|---|---|---|---|
| $V_{50}^{90}$ READ | 0.69% | 1.25% | 0.36% | 0.30% |
| $V_{300}^{99}$ READ | 4.14% | 1.08% | 0.60% | 0.72% |
| $V_{50}^{90}$ UPDATE | 0.46% | 0.48% | 0.12% | 0.18% |
| $V_{300}^{99}$ UPDATE | 4.77% | 2.15% | 0.91% | 1.09% |

False-positive actions are one of the main reasons for over-provisioning, mainly related to the accuracy of controllers. We evaluate the imitator created in IMITA by comparing its performance in four different autoscaling policies, presented in Table II. We calculate the false-positive decisions that the

imitator makes out of the 1600 actions of the presented runs. The benchmark is based on true labels that are produced by the saturation identification procedure, explained in Section IV-C. Here, we use four different learning algorithm ($A$ in Algorithms 2 and 3) to build the imitator as below:

1) A single-metric threshold-based controller that takes actions based on the CPU usage.
2) A rule-based controller that considers CPU usage and received network packets to take actions.
3) A decision tree controller with depth of 16 that takes into account all the resource utilization metrics $y_z$.
4) A random forest controller that consists of 128 decision trees with depth 8, which are built using all the resource metrics $y_z$.

The first two policies result from training depth bounded decision trees on all the resource metrics $y_z$; thus, the used metrics in those policies are the most important ones. Surprisingly, the single metric policy seems to be more generic than the simple rule-based, revealing that more complex rules do not necessarily generalize better. Similarly, the decision tree model is inferior to the single metric policy in Big Spike and Large Variation traffics. However, it has almost $50\%$ fewer false-positives in Dual Phase and Steep-Tri-Phase. The random forest model performs very accurately for these unseen configurations, having only $1-6$ deviation in decisions than the expert.

Moreover, to quantify how well the imitator adheres to an SLO, we define two SLO targets in tail latency (a short tail latency is a desirable SLO). We do not use throughput, as it is the expert's KPI metric to identify resource saturation. The two SLO targets are: SLO1: $P_{99}(latency) < 300ms$, and SLO2: $P_{90}(latency) < 50ms$. We calculate the percentage of SLO *violations* ($V$ in Table III) as the proportion of epochs violate the total number of epochs for each run. The third row of Figure 5 shows the percentiles latency along with a smoothed bezier curve of their values to give an idea about their distribution, as the x-axis is quite dense. The scale-out actions taken by the imitator keep both SLOs consistent under their bounds. As Table III shows, IMITA has an average of $2.23\%$ of violations for SLO1, with the worst-case being $4.77\%$ for the Big Spike. The number of violations for SLO2 is $0.31\%$ on average, with the worst case for Dual-Phase, that has $0.48\%$, which is promising since latency was not considered at training time. Thus, results show that tail latency issues are quite infrequent for the imitator deployed in an unseen environment, even during large traffic variations.

## VI. Related Work

We classify the autoscaling experts by the type of metrics used, as listed below.

*Autoscaling experts based on KPI metrics:* KPIs used in autoscaling scenarios are typically the round-trip time [13], the response time [31], [32], the throughput [33], and the deadline [34]. KPI-based autoscalers are accurate when designed by domain experts but are also limited to the specific application environment. KPI-dependent models with fixed run-time parameters are dependent on their design-time setup (e.g., state-space modeling [35]), whereas solutions with tunable parameters are slightly adaptive to new operational setups [33], [36], [37]. ML-based solutions [28], [34], [38], or fuzzy logic-based [28], [39] have the most precise models. The downside, however, is the need for re-training in new environments. Hybrid controllers and control switching solutions are suggested in [40]–[42] and [43], respectively. Both groups come short in generalizing to new environments and can be unstable during workload surges.

*Autoscaling experts based on platform metrics:* These solutions can be optimized for performance [44] or resource utilization objectives [45]. Queuing [46] and blackbox models [47], [48] are used in fixed-gain autoscalers, where the model parameters are estimated offline, and therefore remain fixed at runtime. The self-tuning blackbox controller in [49] and the parameter estimation based on Kalman filter in [45] can adapt to varying workloads but are application-specific. AGILE [9] uses online CPU prediction by curve-fitting on wavelet signals. Although it is accurate, it requires re-training to be used in new environments.

*Autoscaling experts based on both KPI and platform metrics:* Compared to the previous two classes, this class of autoscaler can better relate performance degradation to saturation, resulting in more accurate scaling actions [50], [51]. Techniques include ML [10], [11] and fuzzy systems [39], [52]. However, as mentioned above, the use of KPIs introduces environment-dependency issues that complicate software provisioning/operation.

## VII. Conclusions

In this paper, we have introduced the concept of imitation learning (IL) for cloud orchestration. It enables the domain experts to train machine learning (ML) models that can be easily generalized across deployments, simplifying operations. As IL is based on expert examples (trajectories), it requires less data collection to converge to an appropriate model compared to other approaches. To demonstrate this concept, we implemented IMITA, a controller model based on IL, to imitate the expert autoscaling functions. IMITA relies on platform metrics (instead of application-specific KPIs) that allow the learned model (imitator) to extrapolate the expert behavior to new environments. Experiments show that IMITA effectively learns to autoscale the application in an environment for which it was not trained – avoiding resource under-provisioning with fewer false-positive scale-out actions than baseline approaches. As future work, we plan to include more orchestrator actions such as vertical scaling and migration to achieve a generic solution for a broader class of applications.

## Acknowledgment

This work was partly supported by the EC H2020 DataCloud[1] project under Grant Agreement No. 101016835.

[1]http://datacloudproject.eu

## REFERENCES


[1] C. Rossi. (2017) Rapid release at massive scale. https://code.fb.com/web/rapid-release-at-massive-scale/.

[2] B. Beyer et al., *Site Reliability Engineering: How Google Runs Production Systems*, 1st ed. O'Reilly Media, Inc., 2016.

[3] R. Boone. (2016) A history of site reliability engineering at uber. https://eng.uber.com/sre-talks-feb-2016/.

[4] N. Mische, "From engineering operations to site reliability engineering." San Francisco, CA: USENIX Association, 2017.

[5] D. Rensin, "Building successful SRE in large enterprises—one year later." Santa Clara, CA: USENIX Association, 2018.

[6] A. Tobey, "Breaking in a new job as an SRE." Santa Clara, CA: USENIX Association, 2018.

[7] A. Gandhi et al., "Adaptive, model-driven autoscaling for cloud applications." in *Autonomic Computing, 2014. (ICAC'14). IEEE International Conference on*, vol. 14, 2014, pp. 57–64.

[8] ——, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, pp. 14:1–14:26, Nov. 2012.

[9] H. Nguyen et al., "Agile: Elastic distributed resource scaling for infrastructure-as-a-service." in *Autonomic Computing, 2013. (ICAC'13). IEEE International Conference on*, vol. 13, 2013, pp. 69–82.

[10] U. Syed et al., "A reduction from apprenticeship learning to classification," in *Advances in Neural Information Processing Systems*, 2010, pp. 2253–2261.

[11] M. Wajahat et al., "Mlscale: A machine learning based application-agnostic autoscaler," *Sustainable Computing: Informatics and Systems*, 2017.

[12] A. Kontarinis et al., "Cloud resource allocation from the user perspective: A bare-bones reinforcement learning approach," in *International Conference on Web Information Systems Engineering*. Springer, 2016, pp. 457–469.

[13] E. Barrett et al., "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[14] C. Xu et al., "Url: A unified reinforcement learning approach for autonomic cloud management," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 95–105, 2012.

[15] A. Hussein et al., "Imitation learning: A survey of learning methods," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 21:1–21:35, 2017.

[16] A. Lakshman et al., "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[17] O. Ibidunmoye et al., "Performance anomaly detection and bottleneck identification," *ACM Comput. Surv.*, vol. 48, pp. 4:1–4:35, 2015.

[18] Q. Wang et al., "An experimental study of rapidly alternating bottlenecks in n-tier applications," in *IEEE International Conference on Cloud Computing*. IEEE, 2013, pp. 171–178.

[19] ——, "Lightning in the cloud: A study of very short bottlenecks on n-tier web application performance," in *USENIX Conference on Timely Results in Operating Systems*, 2014.

[20] S. Ross et al., "A reduction of imitation learning and structured prediction to no-regret online learning," in *International Conference on Artificial Intelligence and Statistics, AISTATS*, 2011, pp. 627–635.

[21] Y. Freund et al., "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[22] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[23] Linux cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[24] cAdvisor. https://github.com/google/cadvisor.

[25] Prometheus. https://prometheus.io/, [Accessed Nov. 2019].

[26] B. Cooper et al., "Benchmarking cloud serving systems with ycsb," in *ACM symposium on Cloud computing (SoCC'10)*. ACM, 2010, pp. 143–154.

[27] M. Arlitt et al., "A workload characterization study of the 1998 world cup web site," *IEEE network*, vol. 14, no. 3, pp. 30–37, 2000.

[28] P. Jamshidi et al., "Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures," in *Quality of Software Architectures (QoSA'16), ACM SIGSOFT Conference on*. IEEE, 2016, pp. 70–79.

[29] A. Gandhi, "Dynamic server provisioning for data center power management," *PhD diss., Intel*, 2013.

[30] N. Herbst et al., "Bungee: an elasticity benchmark for self-adaptive iaas cloud environments," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2015, pp. 46–56.

[31] X. Liu et al., "Adaptive entitlement control of resource containers on shared servers," in *IFIP/IEEE International Symposium on Integrated Network Management (IM'05), 2005.* IEEE, 2005.

[32] P. Padala et al., "Automated control of multiple virtualized resources," in *ACM European conference on Computer systems (EuroSys'09)*. ACM, 2009, pp. 13–26.

[33] A. Ali-Eldin et al., "An adaptive hybrid elasticity controller for cloud infrastructures," in *Network Operations and Management Symposium (NOMS'12), IEEE*. IEEE, 2012, pp. 204–212.

[34] S. Park et al., "Self-tuning virtual machines for predictable escience," in *IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2009.

[35] A. Arman et al., "Elasticity controller for cloud-based key-value stores," in *Parallel and Distributed Systems (ICPADS), IEEE International Conference on*. IEEE, 2012, pp. 268–275.

[36] S. Farokhi et al., "Performance-based vertical memory elasticity," in *Autonomic Computing (ICAC'15), IEEE International Conference on*. IEEE, 2015, pp. 151–152.

[37] J. Ho et al., "Model-free imitation learning with policy optimization," in *International Conference on Machine Learning*, 2016, pp. 2760–2769.

[38] S. Farokhi et al., "A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach," *Future Generation Computer Systems*, vol. 65, pp. 57–72, 2016.

[39] P. Jamshidi et al., "Autonomic resource provisioning for cloud-based software," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEASM'14)*. ACM, 2014, pp. 95–104.

[40] E. Klein et al., "Batch, off-policy and model-free apprenticeship learning," in *European Workshop on Reinforcement Learning*. Springer, 2011, pp. 285–296.

[41] A. Al-Shishtawy et al., "Elastman: elasticity manager for elastic key-value stores in the cloud," in *ACM Cloud and Autonomic Computing Conference (CAC'13)*. ACM, 2013, p. 7.

[42] S. Farokhi et al., "Coordinating cpu and memory elasticity controllers to meet service response time constraints," in *Cloud and Autonomic Computing (ICCAC), International Conference on*. IEEE, 2015, pp. 69–80.

[43] J. O. Iglesias, J. A. Aroca, V. Hilt, and D. Lugones, "Orca: an orchestration automata for configuring vnfs," in *ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 81–94.

[44] C. Barna et al., "Cloud adaptation with control theory in industrial clouds," in *Cloud Engineering Workshop (IC2EW'16), 2016 IEEE International Conference on*. IEEE, 2016, pp. 231–238.

[45] E. Kalyvianaki et al., "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 117–126.

[46] I. Gergin et al., "A decentralized autonomic architecture for performance control in the cloud," in *2014 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2014, pp. 574–579.

[47] H. Lim et al., "Automated control for elastic storage," in *International conference on Autonomic computing*. ACM, 2010, pp. 1–10.

[48] ——, "Automated control in cloud computing: challenges and opportunities," in *Workshop on Automated control for datacenters and clouds*. ACM, 2009, pp. 13–18.

[49] P. Padala et al., "Adaptive control of virtualized resources in utility computing environments," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 289–302.

[50] A. Moulavi et al., "State-space feedback control for elastic distributed storage in a cloud environment," in *International Conference on Autonomic and Autonomous Systems (ICAS 2012)*, 2012, pp. 589–596.

[51] P. Saikrishna et al., "Stability analysis of cloud computing systems under uncertain time delays," in *International Symposium on Mathematical Theory of Networks and Systems*, 2014.

[52] P. Lama et al., "Efficient server provisioning with end-to-end delay guarantee on multi-tier clusters," in *International Workshop on Quality of Service, 2009. (IWQoS'09)*. IEEE, 2009, pp. 1–9.