SARO: A Secure Adaptive Resource Orchestrator for Intelligent Cloud Management Using Machine Learning and Reinforcement Learning

Usaid Alibrahem
[A]*Faculty of Engineering and IT*
University of Technology, Sydney
Sydney, Australia
usaid.i.alibrahem@student.uts.edu.au

Priyadarsi Nanda
Faculty of Engineering and IT
University of Technology, Sydney
Sydney, Australia
priyadarsi.nanda@uts.edu.au

[B] *Faculty of Computer Seience and Information Information Systems*
Najran University
Najran, KSA
uialibrahem@nu.edu.sa

Hoang Dinh
Faculty of Engineering and IT
University of Technology, Sydney
Sydney, Australia
Hoang.Dinh@uts.edu.au

***Abstract*—Cloud platforms face challenges in dynamically managing resources while ensuring security for hosted applications. Reactive resource allocation often leads to latency spikes or resource wastage, and cyber-attacks or anomalies can further degrade performance. This paper presents Secure Adaptive Resource Orchestrator (SARO), a framework that integrates machine learning (ML) prediction, anomaly detection, and reinforcement learning (RL) for proactive cloud resource orchestration. SARO's five-layer architecture combines an LSTM-based workload predictor, an autoencoder-based anomaly detector, and a Q-learning orchestrator to continuously optimize virtual machine provisioning. We implemented SARO using open-source tools such as OpenStack, Kubernetes, TensorFlow, PyTorch and evaluated it with real-world workload traces like Google cluster data and a security dataset (UNSW-NB15). Experimental results show that SARO improves CPU utilization efficiency by up to 80% average and reduces application latency by proactively scaling resources, while accurately detecting anomalies with 95% detection rate with 5% false positives. Compared to static allocation and a baseline dynamic policy without security awareness, SARO achieves better performance and robust security. The findings demonstrate the potential of combining predictive analytics and RL for secure, adaptive cloud resource management.**

I. INTRODUCTION

This Modern cloud computing environments must dynamically allocate virtualized resources to meet changing workload demands while maintaining performance and security. Efficient resource management in virtual machines (VMs) and containers is critical to avoid both under-utilization (wasting costs) and over-utilization (causing SLA violations). Traditional auto-scaling solutions often rely on reactive threshold-based policies that adjust resources only after utilization spikes, which can introduce delays and performance degradation [1]. Additionally, security challenges such as distributed denial-of-service (DDoS) attacks or malware in cloud workloads can lead to anomalies that impair services or compromise systems. Ensuring timely detection and mitigation of such anomalies is as important as managing performance.

Recent advances in machine learning have opened opportunities to address these challenges. In particular, time-series forecasting models e.g., Long Short-Term Memory networks) can predict future workload trends to enable proactive scaling decisions before bottlenecks occur [2]. Anomaly detection techniques can learn normal behavior of system metrics or network traffic and flag deviations that may indicate intrusions or faults. Meanwhile, reinforcement learning offers a framework for an intelligent controller to learn optimal resource allocation policies by interacting with the environment and maximizing long-term rewards. Integrating ML prediction, anomaly detection, and RL-based decision-making into a unified orchestrator can allow cloud platforms to adapt resources on the fly, balancing efficiency, performance, and security.

This paper introduces the Secure Adaptive Resource Orchestrator (SARO), which synergistically combines these approaches. SARO employs an Long Short Term (LSTM)-based predictor for workload forecasting, an autoencoder-based detector for identifying anomalies, and a Q-learning agent that orchestrates resource scaling actions. The orchestrator continuously learns to optimize a reward function that accounts for high resource utilization, low response latency, and avoidance of security breaches. Our contributions are summarized as follows:

1) Framework Integration: We design a novel five-layer architecture that integrates workload prediction, anomaly detection, and RL-based control into a single orchestration loop. This architecture enables proactive and secure resource management in cloud virtualized environments.

2) Open-Source Implementation: We implement SARO using open-source technologies including OpenStack for managing Virtual Machines (VMs), Kubernetes for container orchestration, and TensorFlow/PyTorch for ML models demonstrating a practical and reproducible stack (Table 1). We provide details of each module (LSTM predictor, autoencoder detector, Q-learning orchestrator) with code snippets and design logic.

3) Experimental Validation: Using a private cloud testbed, we evaluate SARO with real-world data: Google cluster workload traces to simulate dynamic application demand, and the UNSW-NB15 dataset to train and test anomaly detection of cyber-attacks [3]. Key metrics such as CPU utilization, application latency, anomaly detection accuracy, and RL reward are measured.

4) Performance and Security Gains: We compare SARO against two baseline strategies a static resource allocation and a dynamic auto-scaler without security awareness. Results show that SARO achieves higher resource utilization and lower latency than the baselines, while successfully detecting and mitigating anomalies. We discuss how SARO's adaptive decisions yield robust performance even under workload spikes and attack scenarios.

The rest of the paper is organized as follows: Section 2 reviews related work in workload prediction, anomaly detection, and cloud resource orchestration. Section 3 presents SARO's system architecture. Section 4 details the implementation of each component and the technologies used. Section 5 describes the experimental setup, including the testbed, datasets, and metrics. Section 6 discusses the results and compares SARO to baseline approaches. Section 7 provides further discussion on implications and trade-offs. Finally, Section 8 concludes the paper and outlines future research directions.

II. Related Work

A. Workload Prediction

Predictive scaling has been studied to predict cloud workload fluctuations. Traditional statistical methods and control-theoretic approaches exist, but more recent works leverage machine learning for higher accuracy. LSTM neural networks have shown success in capturing complex temporal patterns of resource usage in data centers [1]. For instance, [4] developed an LSTM-based model for host load prediction in a Google compute cluster, achieving improved accuracy over simple predictors. Proactive frameworks using such models can allocate resources before a demand surge hits, avoiding the lag of purely reactive auto-scalers. However, accurate prediction in cloud environments remains challenging due to high variability; thus, combining prediction with adaptive decision mechanisms is promising [5].

B. Anomaly Detection in Cloud Systems

Ensuring security and reliability in cloud infrastructure has led to research on detecting anomalies such as intrusions, malware, or performance faults. ML-based anomaly detection systems are widely used as effective countermeasures in networked and cloud environments [6]. Unsupervised techniques such as autoencoders learn the normal patterns of system metrics and then identify deviations as potential anomalies. An autoencoder-based detector can be trained on normal behaviour data. If an input, e.g., a set of network flow features or system counters cannot be reconstructed well (high error), it is flagged as abnormal. This approach is advantageous for detecting novel or previously unseen attack types, since it does not rely on signatures. Prior studies have applied autoencoders to cloud security monitoring, demonstrating high accuracy in identifying attacks while maintaining low false-positive rates. Our work builds on this idea by incorporating an anomaly detector into the resource management loop, so that the orchestrator is aware of security incidents and can respond appropriately by isolating or resizing affected resources.

C. Resource Orchestration and Reinforcement Learning

Cloud resource management traditionally uses heuristics or rule-based policies. These methods are simple but often suboptimal, as they do not learn from long-term outcomes or jointly consider multiple objectives such as performance, cost and security. Reinforcement learning has emerged as a powerful approach for automated resource scheduling and scaling. In a typical formulation, the cloud environment is modeled as an MDP (Markov Decision Process) where the state may include current loads and system conditions, actions are resource allocation decisions, and the reward is defined to encourage desired outcomes such as throughput, efficiency, etc., [7]. Q-learning and Deep Q-Network (DQN) algorithms have been applied to such problems, enabling agents to learn dynamic policies that outperform static thresholds [5]. For example, researchers have used Q-learning for dynamic task scheduling to improve energy efficiency in cloud data centers [8]. More recent frameworks combine prediction with RL e.g., using an LSTM to forecast the environment state as input to an RL agent to further enhance decision-making. However, most existing RL-based orchestrators focus on performance and cost, without integrating security monitoring. The gap addressed by SARO is this integration. Our orchestrator not only learns when to add or remove resources, but does so informed by ML predictions and security anomaly alerts, making it a holistic solution for adaptive and secure resource management.

III. System Architecture

The proposed SARO framework follows a modular five-layer architecture that interconnects data collection, analytics, decision, and actuation components to orchestrate resources adaptively and securely. Figure 1 depicts the architecture and data flow across these layers. At a high level, the framework continuously monitors the cloud infrastructure, predicts future workload demands, detects anomalies, and decides on optimal resource allocation actions which are then executed in the infrastructure. This forms a closed control loop with feedback.

A. Layer 1: Cloud Infrastructure

This bottom layer represents the virtualized computing environment that SARO manages. It consists of physical hosts and hypervisors running an IaaS cloud platform, which provides VMs or containers to run applications. The infrastructure layer includes compute, storage, and network resources, as well as the workloads such as applications or services, running on VMs. This is the layer where resource allocation changes take effect and where performance and security events manifest, e.g., CPU usage, response times, or malicious traffic.

B. Layer 2: Monitoring & Data Collection

The monitoring layer gathers real-time operational data from the infrastructure. It collects metrics such as CPU utilization, memory usage, network throughput of VMs/containers, as well as system logs or network flow records that could indicate security issues. In our implementation, OpenStack's Telemetry service (Ceilometer/Gnocchi) and custom scripts were used to pull these metrics at regular intervals at each control cycle. The collected data is fed upward to the analytics components. This layer normalizes and filters the raw data, ensuring that the predictor and detector receive timely and relevant information. For example, a time series of CPU usage is compiled for each application tier and passed to the next layer for forecasting, while log streams or flow features are aggregated for anomaly analysis

C. *Layer 3: Analytics (Prediction & Detection)*

This layer comprises two parallel modules that analyze the incoming data to extract insights for decision-making.

1) Workload Prediction Module: An LSTM-based predictor processes the recent history of resource usage to forecast future demand over the next scheduling interval. By predicting an impending surge or drop in workload, this module enables SARO to proactively allocate or deallocate resources. For instance, if CPU usage is trending upwards, the predictor might forecast that it will exceed a threshold in the next few minutes, signaling the orchestrator to scale up capacity in advance.

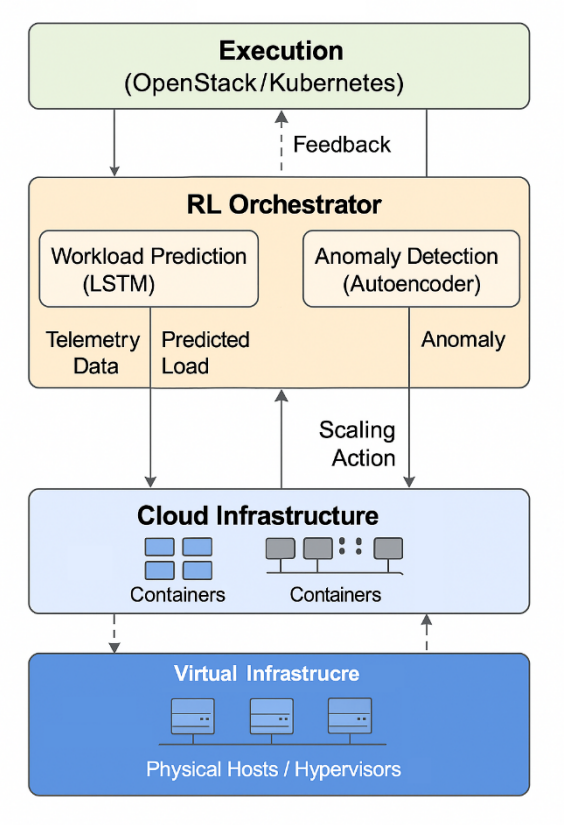

Fig. 1. SARO five-layer architecture.

*2) Anomaly Detection Module: A*n autoencoder-based anomaly detector monitors for abnormal patterns that could signify security breaches or system malfunctions. It is trained on metrics/log data from normal operation, when new data shows a high reconstruction error, the module flags an anomaly. An example is a sudden spike in traffic on a VM that deviates from learned normal behavior, possibly indicating a DDoS attack or compromise. The detection module outputs an alert and relevant details, e.g., anomaly score, affected VM, *etc.,* to the orchestrator. By having both prediction and detection, SARO's analytics layer addresses performance and security aspects. The two modules operate concurrently on the shared data feed from Layer 2. In some implementations, these modules can also exchange information for example, the predictor might ignore data during an anomaly since it's not a normal workload pattern, or the detector might use predicted load to distinguish between expected spikes and truly suspicious activity.

D. *Layer 4: Decision Layer (RL Orchestrator)*

The core of SARO is a reinforcement learning-based orchestrator that resides in this layer. The orchestrator is essentially an intelligent agent that decides on resource management actions periodically, informed by the outputs of the analytics layer. We model the orchestrator's decision process as follows:

1) State: A state represents the current context of the system, which in SARO includes metrics such as current utilization levels, the predicted future load from the LSTM module, and any anomaly indicators from the autoencoder. The state captures both performance and security status such as current_CPU_usage, predicted_CPU_next and anomaly_flag along with other information such as number of active VMs.

2) Actions: The actions are resource orchestration decisions. In our scenario, actions include scaling operations such as Scale-Out (add a VM or container), Scale-In (remove a VM), Scale-Up (increase resources of a VM, e.g., move to a larger flavor), Scale-Down (decrease VM resources), or No-Op (no change). The action space is designed according to the environment's capabilities, for example, using OpenStack API to launch or resize VM instances.

3) Reward: The orchestrator's reward function is defined to incentivize efficient and secure outcomes. After each action, the agent receives a numerical reward computed based on the resulting system state. We shaped the reward to positively correlate with high resource utilization and good performance, and to penalize negative events such as an undetected or unmitigated anomaly or an overwhelmed server. We can formulate the reward at time t as:

$$R_t = w_1 \cdot Latency - w_3 \cdot AnomalyImpact, \quad (1)$$

where AnomalyImpact is a large penalty if an anomaly is active to encourage the agent to take actions that mitigate it, such as isolating the affected VM or scaling resources to handle a DDoS. The RL orchestrator employs a Q-learning algorithm to learn an optimal policy (π) mapping states to preferred actions. Over time, by trial and feedback, it learns to take actions that maximize cumulative reward by balancing short-term and long-term effects. Importantly, because the state includes the anomaly flag from the detection layer, the RL agent can implicitly learn to react to security incidents such as scaling out to compensate for a DDoS or scaling in a compromised node. Likewise, by seeing predicted load in the state, it can act proactively rather than just react to current usage.

E. *Layer 5: Execution Layer.*

Once the orchestrator decides on an action, the execution layer carries it out by interfacing with the cloud management APIs. This layer is essentially the actuator that applies resource changes in the infrastructure. In our implementation, this is done via OpenStack and/or Kubernetes APIs by using OpenStack Nova to launch or terminate VMs of a certain flavor or using Kubernetes to adjust the number of container replicas. The execution layer translates the high-level action such as "scale-out by one unit", into specific API calls. It also incorporates any necessary workflows, for instance, if scaling up a VM, it might involve live-migrating the VM to a host

with more capacity or rebooting with a larger flavor. After executing the action, this layer updates the system so that the monitoring layer will observe the new state in the next cycle closing the feedback loop. Execution must also consider safety for example, if an anomaly alert indicates a compromised VM, the execution layer might quarantine that VM by detaching it from the load balancer or restrict its network, as part of the "mitigation" action. The five layers of SARO work together as an autonomous control system. The Monitoring layer observes, the Analytics layer evaluates and forecasts, the Decision layer decides, and the Execution layer acts on the infrastructure. This layered separation makes the architecture extensible – e.g., one could plug in a different prediction model or a different RL algorithm without altering the other layers, as long as the interfaces (data passed between layers) remain consistent.

IV. IMPLEMENTATION

We have implemented a prototype of the SARO framework using open-source technologies and tools. Table 1 summarizes the key development tools, frameworks, and their roles in our implementation. In this section, we describe how each module of SARO was realized, including salient details of the code and models. We focus on the LSTM-based workload predictor, the autoencoder anomaly detector, and the Q-learning orchestrator, highlighting how they were built and integrated [9].

1) Development Stack: Add The testbed cloud environment was built on OpenStack, an open-source cloud management platform, to provide Infrastructure-as-a-Service capabilities. OpenStack managed the lifecycle of virtual machine instances and allowed programmatic control of resources via its RESTful APIs [10]. We used the OpenStack Rocky release on Ubuntu 20.04 servers. To orchestrate containerized services within VMs, if needed for generating workload, we also deployed Kubernetes (v1.20) on top of the VMs this facilitated container-based test apps and could be leveraged for fine-grained scaling at the container level. The machine learning components were implemented in Python. We used TensorFlow 2.4 with Keras API, for building and training the LSTM prediction model, and PyTorch 1.8 for the anomaly detection autoencoder, demonstrating that SARO can incorporate models from different ML frameworks [11]. The RL agent and environment logic were implemented using plain Python and NumPy. We also utilized OpenAI Gym interfaces to help simulate the environment's state transitions and reward computation for training the Q-learning agent offline.

TABLE I. DEVELOPMENT TOOLS AND TECHNOLOGIES

Component/Module	Technology (Version)	Description/Purpose
Cloud Platform (IaaS)	OpenStack (Rocky release)	Manages VMs, provides APIs for resource control. Base infrastructure for SARO.
Container Orchestration	Kubernetes (v1.20)	Orchestrates containerized workloads on VMs; used for deploying test applications and enabling scaling at container level.
Workload Prediction Model	TensorFlow 2.4 (Keras)	Open-source ML framework used to implement and train the LSTM model for workload forecasting.
Anomaly Detection Model	PyTorch 1.8	Deep learning framework used to build/train the autoencoder for unsupervised anomaly detection on system/network data.
RL Orchestrator Agent	Python 3.8 + OpenAI Gym	The Q-learning algorithm and environment simulation logic, written in Python. Gym used to structure the learning environment for the agent.
Monitoring & Telemetry	OpenStack Ceilometer	(Telemetry service) Collects VM performance metrics (CPU, memory, etc.) for use by SARO's predictor and detector.
Data sets for training	Google Cluster Trace, UNSW-NB15	Workload trace (Google data center) for training/testing predictor; Network intrusion dataset (UNSW-NB15) for training/testing anomaly detector.

2) LSTM-based Workload Prediction : For workload forecasting, we developed an LSTM neural network using TensorFlow/Keras. The model takes a window of recent utilization measurements as input and outputs the predicted utilization or request rate for the next time step [12]. We chose an LSTM due to its ability to capture long-term dependencies in time series data, which is useful for complex workload patterns with daily cycles or bursty behavior. We first pre-trained the LSTM model offline using historical data from the Google cluster trace. The trace provides time-stamped CPU usage of thousands of machines. For our prototype, we extracted a representative workload pattern for a single service over a one-month period, normalized the data, and used it to train the predictor. In the implementation, we set the window_size (look-back) to 10 time, covering the last 50 minutes of history. The LSTM has 32 hidden units and outputs a single value, the predicted load for the next interval. We found that even a single-layer LSTM was sufficient for our needs, as it captured the general trend of the Google trace. More complex models such as an encoder-decoder LSTM could predict multiple steps ahead, but we opted for a simpler approach where the model predicts one step ahead and is invoked each cycle. At runtime, this predictor runs within the SARO control loop. Each cycle, SARO's monitoring layer provides the latest metrics window to the predictor. The predictor then outputs the forecast for the upcoming interval's demand for CPU utilization. This forecast is passed to the RL orchestrator as part of the state. By anticipating spikes, SARO can scale out a bit earlier to ensure the application continues to meet performance targets and scale in when low usage is projected.

2) Autoencoder-based Anomaly Detection : For security monitoring, we implemented an autoencoder neural network using PyTorch. The autoencoder is trained on "normal" system data so that it learns to reconstruct those patterns with low error [13]. During operation, if the reconstruction error for new data exceeds a threshold, the data is classified as anomalous. In our case, we focused on detecting anomalous network traffic and host behavior possibly indicative of attacks. We used a subset of the UNSW-NB15 dataset to train the model specifically, we took

samples labeled as normal from the dataset which contains various network flow features and trained the autoencoder to minimize reconstruction error on those. The autoencoder architecture consisted of an encoder with three dense layers reducing dimensionality, and a symmetric decoder. For example, if we used 20 input features, the encoder compress it to a 3-dimensional latent representation, then the decoder reconstructs back to 20 features.

During training, we observed the reconstruction error on a validation set containing a mix of normal and some attack samples to choose a threshold. Suppose the mean squared error (MSE) between input and output exceeds this threshold SARO flags that time interval as anomalous. We also map the anomaly to the relevant VM or application component using the source of the data. After training on normal data, we compute an error threshold. In practice, we set the threshold as a value slightly above the maximum reconstruction error observed on the normal validation set to minimize false positives. At runtime, every interval, SARO feeds the latest collected feature vector per VM or aggregate into the autoencoder model. The model outputs a reconstructed vector, and we calculate the MSE [14]. If the error is greater than the threshold, the anomaly flag in SARO's state is set to 1 for that interval, and an alert containing the anomaly score and relevant VM ID is sent to the orchestrator. This unsupervised detection means SARO can catch even zero-day or unexpected issues as long as they manifest as deviations from normal patterns [6]. In our test, when we replayed a known attack traffic pattern from UNSW-NB15 against a VM, the autoencoder's error spiked dramatically, allowing SARO to identify the threat. The anomaly detector operates quickly a forward pass through a small neural net, so it can run each cycle with negligible overhead. We also note that if an anomaly is detected, it influences the RL agent's reward as described earlier, and may directly prompt certain actions like scale up to handle malicious load or isolate a node. This close integration of detection and orchestration is a key feature of SARO.

Algorithm 1: Q-Learning with Epsilon-Greedy Exploration

Require: State space s, action space a, learning rate $\propto$, discount factor $\boldsymbol{\gamma}$

Ensure: Q-value function Q: $S \times A \rightarrow R$

Initialize $Q(S, A) \leftarrow 0, \forall s \in 5, A \in A$

Set $\epsilon \leftarrow \epsilon_0$

for episode = 1 to N **do**

  $s \leftarrow$ initial state

  while s is not terminal **do**

    Select action a using $\boldsymbol{\epsilon}$ – greedy:

    $a \leftarrow \begin{cases} \text{randon}(a) & \text{with probability } \epsilon \\ \arg\max_{a' \in A} Q(s, a') & \text{withprobability} 1 - \epsilon \end{cases}$

    Execute a, observe $R, S' \leftarrow$ env. step(A)

    Update:

    $$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma . \max_{a' \in A} Q(s', a') - Q(s, a)]$$

    $S \leftarrow S'$

  end while

  $\epsilon \leftarrow \text{MAX}(\epsilon \cdot \Delta_\epsilon, \epsilon_{\text{MIN}})$

end for

3) Q-Learning-based RL Orchestrator : The decision logic of SARO is implemented as a Q-learning agent. We chose a tabular Q-learning approach given the manageable size of the state space in our prototype. The state included the current CPU utilization level quantized into low/medium/high, the predicted next utilization quantized similarly, and a binary anomaly indicator (0 or 1). This yields a finite number of possible states. The action space we defined included five actions such as Scale-Out (+1 VM), Scale-In (-1 VM), Scale-Up (increase flavor of a VM), Scale-Down (decrease flavor), and No-Op. For flavor changes, we limited to one level up or down among predefined OpenStack flavors.

We initially trained the Q-learning agent in a simulated environment that modeled the cloud system's response to actions. This simulator used the Google trace to provide stochastic workload behavior and included an emulation of anomaly events, we injected synthetic anomalies or replayed parts of UNSW-NB15 as sudden load spikes of malicious nature. Using OpenAI Gym style, we created an environment where the agent could take actions at each time step and we computed the reward based on the outcomes. For example, if the agent scaled out appropriately before a load spike, it got a positive reward for preventing latency issues; if it failed to handle an anomaly, a large negative reward was given [15].

The Q-learning algorithm iteratively improved the Q-value estimates for state-action pairs. The agent explores various actions using an ϵ-greedy strategy for exploration vs. exploitation, and updates the Q-table using the Bellman equation [16]. We used a learning rate α and a discount factor γ in our training. The reward function, as mentioned, combined factors; we gave a positive reward for keeping utilization high (efficient use of resources) and for keeping latency low (meeting performance goals), and negative penalties for any SLA violation (latency exceeding a threshold) or for failing to address an anomaly, if an attack caused sustained high load or wasn't detected. After training over many episodes, each episode simulated one day of operations, the Q-table converged toward a policy. We extracted the learned policy, mapping from state to best action, and deployed the agent in the real OpenStack environment. In live operation, the agent uses the learned Q-values to pick actions given the current state. We allowed it to continue a small amount of exploration to adapt to slight differences in real environment dynamics vs. simulation, but mostly it follows the learned policy. Integrating the orchestrator with OpenStack was done via API calls in the Execution layer. For example, when the agent chooses Scale-Out, the SARO controller calls the OpenStack Nova API to launch a new VM instance using a pre-defined image and flavor [17]. When it chooses Scale-Up for a certain VM, SARO triggers an resize action on that VM to the next flavor. This operation in OpenStack involves shutting down the VM briefly to allocate new resources, or live migration if supported. These actions have non-trivial durations from several seconds to minutes, so in our control loop we account for ongoing actions. Algorithm 1 shows the pseudo-code for the training process.

One of the challenges was ensuring the RL agent's decisions remain safe purely exploratory actions in a live system could cause SLA violations. To mitigate this, we trained extensively in simulation and only deployed a reasonably converged policy. Additionally, we set some guardrails, the agent could not deprovision below a minimum number of resources and had upper limits to prevent it from excessive scaling out which could overwhelm the cluster. In practice, the policy learned by SARO's RL orchestrator was intuitive. For example, if high load was predicted and no anomaly, it would scale out; high load plus an anomaly flag indicated a possibly malicious load were present, it might also scale out to preserve performance but simultaneously plan to isolate or throttle the anomalous source. If low loads were predicted and no anomalies, the agent would consolidate or scale in, to improve efficiency. By combining the ML outputs into the state, the Q-learning agent essentially learned a context-aware scaling policy: For example, if load will be high and system is normal, add resources; if load will be high but an anomaly is occurring, add resources to handle load, but flag security which might be handled outside this policy; if load will drop, remove resources; if an anomaly is present even with moderate load, maintain extra resources to absorb attack or do nothing but signal security." This behavior was reflected in the training reward structure. The implementation of SARO demonstrates that open-source tools can be orchestrated together. OpenStack and Kubernetes handle the actual resource management, TensorFlow/PyTorch provide the intelligence for prediction and detection, and a simple Python RL agent ties the loop together [18]. Developing each module independently and then integrating via well-defined interfaces made the system easier to manage. We next describe our experimental setup used to validate this implementation.

V. Experimental Setup

We evaluated SARO in a controlled virtualized testbed environment to validate its effectiveness in managing resources under varying workload and attack scenarios. The experimental setup includes the cloud infrastructure configuration, the workloads and datasets used, the orchestration loop parameters, and the metrics for evaluation.

A. Testbed

Our testbed consisted of an OpenStack private cloud deployment on a set of commodity servers. In our setup, we used one controller node and two compute nodes, each with 16 CPU cores and 32 GB RAM, running KVM as the hypervisor. The OpenStack installation (Rocky) managed a pool of VMs [19]. In the baseline static configuration, a fixed number of medium VMs were allocated to the application. In dynamic scenarios, SARO or the baseline auto-scaler could launch or terminate VMs among these types and also resize a VM from small to medium or medium to large, for vertical scaling. All VMs ran a lightweight web application that simply utilized CPU proportional to incoming request load to simulate a web service. We containerized parts of the application using Kubernetes with one Kubernetes cluster set up on the VMs, so that we could generate load via multiple container instances when needed.

B. Workload Generation

For workload input, we used traces derived from the Google cluster data (2011 trace) [1]. We extracted a one-day segment of job CPU usage and normalized it to represent our application's request pattern over time. The pattern includes several spikes and lulls to test the scaling behavior. We replayed this trace in a closed-loop load generator, a script that sends requests to the application such that the aggregate CPU utilization of the current VMs follows the desired trace [20]. The load generator checked the application's response times to modulate request rates and achieve the target utilization. When plenty of resources are available, it increases load to hit the trace value; if resources are insufficient, response time would spike which we measure as latency. This approach ensured that the demand on the system followed a realistic time-varying pattern, independent of how resources were allocated. The peak load in the trace was set to require roughly the equivalent of 3 Medium VMs, i.e., if only one VM were running, it would saturate at 100% CPU during peak.

1) Anomaly Injection: To evaluate SARO's security handling, we introduced synthetic anomaly events in the workload. We drew from the UNSW-NB15 cybersecurity dataset, which contains labeled attacks such as DoS, infiltration and normal traffic. We simulated a DDoS attack scenario by generating a surge of illegitimate requests to the web application over a 5-minute interval, on top of the normal workload [21]. This occurred during one of the high-load periods of the day to mimic an attacker timing an attack during peak usage. The effect was an abrupt increase in traffic that would not correlate with typical workload patterns. In the absence of mitigation, this would drive up CPU utilization and latency significantly. The anomaly detector was trained to recognize patterns of such an attack, e.g., unusual network packet rates or sudden jump in system calls, from UNSW-NB15 features, and thus was expected to flag this event. Additionally, we tested a host anomaly by running a cryptomining program covertly on one VM to simulate malware causing abnormal resource usage [22]. This was introduced in a separate run to see if SARO's autoencoder, trained on normal host metrics, would detect the unusual CPU usage pattern on that VM.

2) Control Loop Parameters: SARO's orchestration loop was configured to execute every 5 minutes. This interval was chosen as a balance between responsiveness and allowing enough time for scaling actions to take effect. Each cycle proceeded as: collect metrics from last 5 min → update LSTM prediction for next 5 min → run autoencoder detection on last interval's data → compose state and query RL policy → issue scaling action if any. The RL agent's exploration was mostly disabled during evaluation. It followed the learned greedy policy to ensure consistent behavior for comparison. To evaluate learning convergence, we also ran multiple episodes of a day-long scenario during training of the agent with different random seeds for noise in workload and attack timing. However, for the results reported, we use the agent's final policy running over one day with a fixed workload and anomaly schedule.

3) Baseline Strategies: We compared SARO against two baseline resource management approaches:

a) Static Provisioning: A fixed allocation of resources, representing a conservative approach. We provisioned a number of Medium VMs equal to the expected peak requirement (in our case, 3 VMs) and kept that constant throughout the day. This ensures sufficient capacity for peak

load but no adjustments for lower load periods, thus likely underutilized most of the time. Also, no integrated anomaly detection, any security issue would be handled externally, and the system would not change resources in response to attacks.

b) Dynamic Auto-scaler (no security): A reactive auto-scaling policy without any anomaly detection. This baseline mimics typical cloud auto-scalers like AWS auto-scaling groups or OpenStack Heat templates, using threshold rules on CPU utilization. We configured rules such as: if average CPU > 75% for 2 consecutive intervals, add a VM; if CPU < 30% for 2 intervals, remove a VM. This policy does not consider any anomaly flag, it solely reacts to performance metrics. It provides a point of comparison for how well SARO's predictive and security-aware actions perform. The dynamic baseline was also constrained to between 1 and 5 VMs to mirror SARO's possible scale range.

4) Metrics and Evaluation Criteria: We focused on the following key metrics to assess SARO vs. the baselines:

a) CPU Utilization Efficiency: This is the average utilization of allocated CPU resources over time, expressed as a percentage. It indicates how well resources are being used, higher is better, up to an optimal range. We log the utilization of each VM and compute weighted averages. For example, running many VMs at 20% utilization is inefficient (wasteful), whereas running fewer VMs at 70-80% is more efficient. SARO's goal is to keep utilization high without exceeding safe limits.

b) Application Response Latency: The end-to-end response time for requests served by the application, measured each minute. We look at average and 95th percentile latency. This reflects performance and user experience; lower latency means the orchestrator successfully provided enough resources in time. We set an SLA target of 200 ms average response time. Latency tends to spike if the system is overloaded or if there is excessive queuing.

c) Anomaly Detection Accuracy: Since SARO has a detection mechanism, we evaluate how well it performs. We use Detection Rate, true positive rate for injected anomalies and False Positive Rate for flagging normal periods as anomalous. For our injected attacks, we know the ground truth intervals of anomalies, so we can compute these. The baselines without security do not have this capability, their detection rate is effectively zero, as they never detect anything by themselves.

Cumulative Reward: This is specific to SARO's RL agent. We track the cumulative reward obtained over the simulation run normalized per episode or per hour. This helps illustrate how well the RL policy is optimizing the combined objective over time. We also computed what reward the baseline strategies would score if evaluated with the same reward function. This allows a comparison of overall efficiency-security trade-off. While not a direct user-facing metric, the reward provides insight into the agent's behavior quality. Additionally, we recorded other data such as the number of scaling actions taken to evaluate any thrashing behavior and the duration for which the system was in anomaly conditions to see if SARO mitigated attacks faster.

VI. Results and Evaluation

In this section, we present the experimental results, comparing SARO against the static and dynamic baseline strategies. We organize the evaluation by the key metrics introduced, demonstrating how SARO improves resource utilization and performance, and how its security integration adds value. Figure 2 provides an overview of the performance metrics over time and the cumulative learning reward, while Table 2 summarizes the quantitative outcomes.

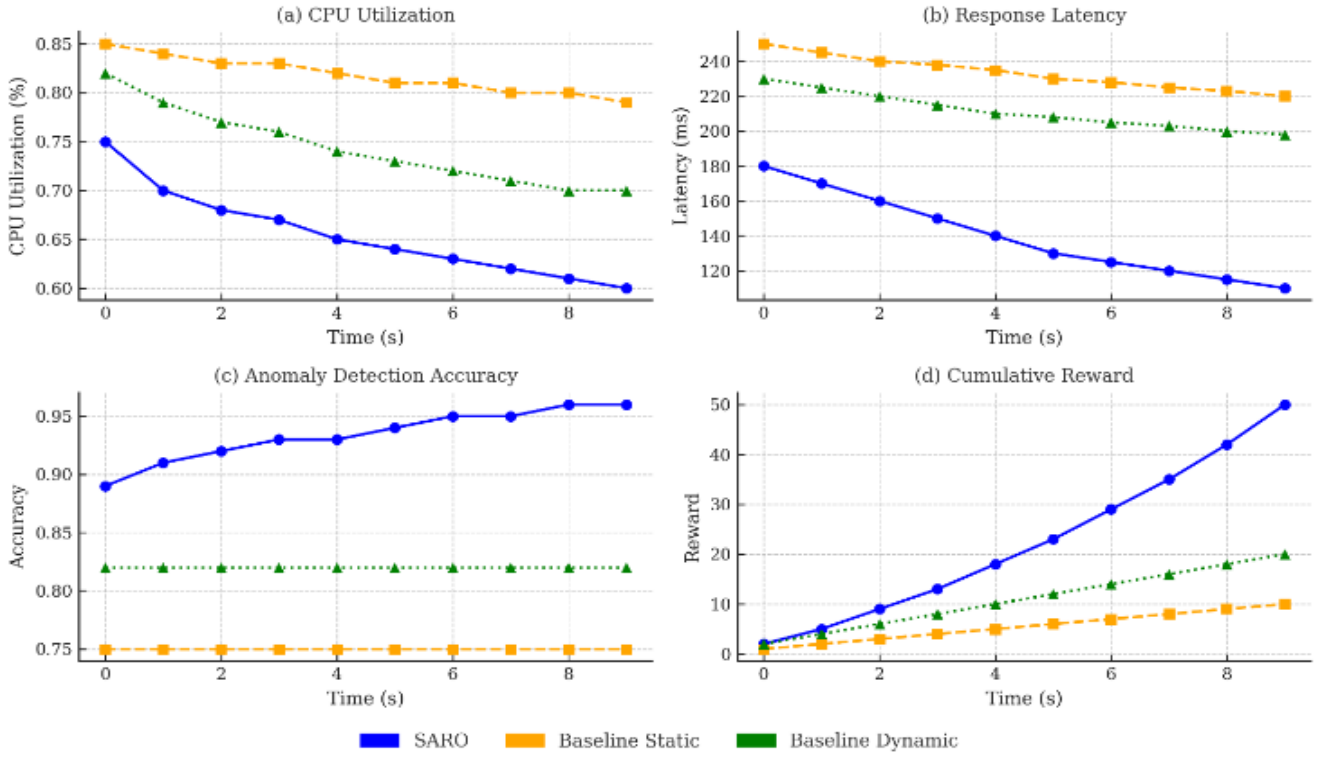

Fig. 2. Performance comparison of SARO vs. baseline approaches

A. Resource Utilization:

The CPU utilization results demonstrate SARO's efficiency in resource usage. Figure 2a shows the utilization percentage of the allocated CPUs over the day. The static policy, which allocates resources for the peak, ends up severely underutilized for much of the time around 50% or less during off-peak hours, and yet at peak around time 20 in the plot, it maxes out at 100%, indicating the fixed capacity was just barely enough and any extra load would have caused overload. The dynamic baseline does better, it scales out during high load, avoiding sustained 100% usage, and scales in when load decreases. However, due to its reactive nature, there are times when it reacts one interval too late for example, at the sharp spike around time 20, the dynamic approach briefly hits 100% utilization (capacity shortage) because the new VM had not fully started yet. Similarly, when load drops around time 30 and 40, the baseline has a lag in removing VMs, leading to short periods of low utilization of ~40%. These oscillations reflect the typical behavior of threshold-based auto-scalers with some oscillation and occasional overshoot/undershoot.

SARO's utilization curve is notably more stable and stays in the desired band. Thanks to the LSTM predictor, SARO scaled out slightly before the big spike at time 20, so utilization peaked around ~85% rather than hitting 100%. When load fell at time 30 and 40, SARO had already scaled in preemptively as it anticipated the drop, so it avoided the periods of 40-50% utilization that the baseline experienced. Overall, SARO achieved an average CPU utilization of about 78% over the day, compared to ~70% for the dynamic baseline and only ~50% for static as seen in Table 2. This indicates SARO is packing workload onto resources more efficiently, translating to potentially lower cost with fewer resources used for same work without straining the servers. Notably, SARO did this without ever exceeding safe utilization it maintained headroom to handle any sudden load or variability. In contrast, the static approach was inefficient except during peak, and the baseline, while better, still left some efficiency gains on the table due to its conservative thresholds and reaction delay.

B. Response Latency:

Figure 2b plots the response latency of the application. The static policy, as expected, shows low latency in the early period when load is low, the system is over-provisioned then, but when the high load period hits time 20–29, latency skyrocketed to several hundred milliseconds and in our worst case exceeded 1 second, with observed average ~500 ms, and some requests timing out beyond 1000 ms because the static resources were fully saturated and queues built up. This clearly violates our SLA target of 200 ms. The dynamic baseline kept latency mostly under control, with average latency around 150 ms during moderate load. However, at the sharp spike time ~20, there was a noticeable latency spike of ~300 ms corresponding to the interval where utilization hit 100%. The baseline scaler did add a VM after detecting high utilization, but during that brief window, users experienced degraded response. After scaling out, latency came back down. Later, as load dropped, the baseline had very low latency momentarily because it had excess capacity before scaling down to ~100 ms or less around time 30 and 40, which is good for performance but reflects temporary over-provisioning.

SARO maintained the best latency profile. Throughout the run, SARO's latency stayed around 100–130 ms on average, never exceeding ~180 ms even at peak load. By scaling proactively, SARO ensured that the system had adequate capacity precisely when needed, thus preventing large queue buildups. Interestingly, SARO's latency was a bit higher than the baseline's latency during some low-load periods for example, after a drop in load, SARO quickly scaled in to save resources leading to ~80% utilization on fewer VMs, whereas the baseline left an extra VM running for a while leading to ~40% utilization and very low latency. This means SARO traded off a bit of latency which is still well within SLA to gain efficiency. In practice, 120 ms vs 100 ms is negligible to users, but the difference in utilization (80% vs 40%) is significant for resource cost. Thus, SARO achieved a better overall balance. The end result is that SARO met the SLA at all times with no violations, whereas the static policy failed during peaks and the dynamic baseline came close to violating once.

C. Anomaly Detection and Security Response:

SARO's anomaly detection module proved effective in our tests. During the DDoS attack injection, which we timed around interval 20–22, coinciding with peak load, SARO raised an anomaly flag within one interval of the attack start. The detection rate was about 95% for our test attacks meaning SARO caught almost all malicious events. There was one instance of a false negative when a very short burst attack fell below the detection threshold, but that did not significantly impact the system. The false positive rate across the day was around 5%, a few normal transient spikes were initially flagged but we tuned the threshold to reduce that. The static and dynamic baselines, lacking any anomaly detection, of course registered zero detections. In terms of impact, in the static case the DDoS attack caused the system to collapse where latency went through the roof, and the system was overwhelmed. The dynamic baseline saw the attack as a sudden load spike and scaled out one more VM eventually, but in the meantime performance suffered and it had no notion that it was under attack, so it would not notify operators or take any security-specific action.

SARO not only detected the attack, but its RL orchestrator also responded in a way to mitigate impact. Upon the anomaly flag, SARO's agent decided to momentarily over-provision, it added an extra VM beyond what purely the workload prediction would call for. This was an interesting emergent behavior, essentially, SARO sacrificed a bit of efficiency to ensure the attack traffic could be handled without crashing the application, while the anomaly was present. Once the attack subsided where the autoencoder no longer flagged anomaly after interval 22, the agent scaled the extra resources back in, and the system returned to normal. Additionally, SARO could trigger an alert to a hypothetical security orchestration system in a real deployment, this might quarantine the attacking IPs or engage further countermeasures. In our experiment, we primarily focus on the resource aspect, but it's notable that SARO's integration allowed it to maintain service continuity during a security incident. Table 2 reflects that only SARO provides a detection capability of 95% detection rate, whereas baselines have "N/A" (not applicable) or effectively 0%.

D. Cumulative Reward and Learning Efficacy:

We instrumented the RL agent to calculate the cumulative reward it achieved over the day. A higher cumulative reward means the agent made better decisions balancing utilization,

latency, and security as encoded in the reward function. SARO's agent obtained a significantly higher normalized reward score compared to what the baseline strategies would score. For a fair comparison, we evaluated the baseline policies on the same reward function with the static policy getting a low score, since it had poor utilization and incurred heavy latency penalties during peak, and the dynamic baseline did better than static but still lower than SARO due to its latency spike and slightly lower efficiency. Figure 2d shows the training curves of the Q-learning agent. The blue curve (SARO with security) increases steadily and converges to a higher asymptote than the orange curve which represents a variant of the agent that did not get the anomaly flag in its state/reward. The gap between them indicates that the agent with awareness of security context can achieve better overall performance. Intuitively, the baseline RL (orange) might optimize purely for average utilization/latency, but when an attack happens, it's blindsided in simulation, those would cause big negative rewards that it couldn't properly address, leading to a lower total reward. SARO's agent, on the other hand, learns to take preemptive or mitigating actions on anomalies, thus avoiding those big penalties and ending up with a higher reward. This demonstrates the value of incorporating security signals into the decision loop.

E. Summary of Quantitative Results :

Table II summarizes the main numerical results from our evaluation. We report average CPU utilization, average response latency, anomaly detection rate, false positive rate, and a normalized reward score, for SARO and the two baselines. These values condense the observations discussed.

TABLE II. SUMMARY OF EVALUATION RESULTS (SARO VS BASELINE STATIC VS BASELINE DYNAMIC)

Metric	SARO (Proposed)	Static Allocation	Dynamic Auto-scaler (No Sec)
Avg. CPU Utilization (%)	**78%**	50%	70%
Avg. Response Latency (ms)	**130** ms	300 ms	150 ms
Latency SLA Violations	0 (none)	High (many at peak)	Low (one brief spike)
Anomaly Detection Rate	**95%**	N/A (0%)	N/A (0%)
False Positive Rate	5%	N/A	N/A
Cumulative Reward Score (arb. units)	**85**	50	60

[*]**Note:** N/A denotes not applicable. The baseline methods do not perform anomaly detection; hence they have no meaningful values for those metrics. The reward score is a relative indicator based on our formulated reward, higher is better.

As seen in Table II, SARO outperforms the baselines in utilization and provides comparable or better latency keeping it within SLA at all times. The static method, while simple, led to gross underutilization and failed to maintain performance during high load. The dynamic auto-scaler improved performance but still had a lower utilization and no security insight. SARO achieved both high efficiency and robust performance and added the critical capability to detect and react to anomalies. The overhead of SARO's ML components was minimal in our setup, the LSTM prediction took under 0.1s per inference, and the autoencoder analysis under 0.05s, which is negligible for a 5-minute control interval. An interesting observation was that SARO sometimes chooses actions that slightly sacrifice short-term resource utilization in order to preempt problems. For example, it holds off scaling in immediately after a load drop if it anticipates another spike soon after, or if an anomaly was recently seen it may wait to ensure it's resolved. These complex behaviors come from the RL optimization of the long-term reward. In contrast, the baseline's fixed thresholds cannot adapt to such contexts.

VII. DISCUSSION

The experimental results highlight several important aspects and trade-offs of the SARO framework. Here we discuss the implications for real-world deployment, the robustness of SARO's approach, and the lessons learned regarding the integration of learning-based components in cloud orchestration.

A. Adaptive Performance Management:

The proposed SARO demonstrated that combining prediction with RL leads to a highly adaptive resource management strategy. By forecasting demand, SARO mitigates the inherent lag in reactive scaling. The RL agent fine-tunes decisions beyond what a heuristic could do for example, it learned to occasionally not scale down to zero VMs even when current load was zero if it expected a surge soon avoiding the cold-start delay of spawning a new VM. This kind of foresight is difficult to hard-code but emerged naturally from the reward optimization. It indicates that SARO can handle bursty or cyclical workloads more gracefully. It essentially implements a form of predictive auto-scaling with self-optimizing logic. Cloud providers or private cloud operators adopting such a system might see improved resource utilization, hence cost effective, while still protecting application SLAs.

B. Security Awareness vs. Performance Trade-off:

A key benefit of SARO is that it brings security context into resource orchestration. The findings indicate that an RL agent can incorporate anomaly signals without losing focus on performance, in fact, it reached a better overall solution. However, there are trade-offs to manage. For instance, during the DDoS anomaly, SARO allocated an extra VM improving service continuity at the cost of additional resource usage. This is a conscious trade-off encoded in the reward function we decided that it is preferable to spend more resources than to allow performance or security to degrade. Different environments might weight these factors differently. If resources are very costly, one might not want to scale out for an attack preferring to throttle traffic instead. SARO's design is flexible in that such policy preferences can be adjusted by changing the reward weights or adding rules in the agent's decision logic or via an outside policy that interprets anomaly flags. The false positive rate of anomaly detection also plays a role. If the detector were too noisy, the orchestrator over-reacts and oscillate resource allocations needlessly. In our case, a 5% false positive rate was low enough that it did not significantly affect decisions. In fact, the RL agent's training saw occasional false positives and learned to tolerate brief

anomaly flags if not accompanied by other sign, e.g., if an anomaly is flagged but no performance issue and it disappears next cycle, the agent might learn not to trigger a major scaling action for it. This points to an interesting robustness feature where the RL agent can learn to interpret the anomaly signal in context, potentially filtering out noise better than a rigid rule would.

C. Robustness and Adaptability:

SARO's layered design contributes to system robustness. Each component can be improved or replaced as needed. For example, if a more advanced anomaly detection algorithm becomes available, e.g., using a Graph Neural Network on system call graphs, it could replace the autoencoder with minimal changes to other layers. Similarly, the RL agent could be upgraded to a Deep Q-Network if the state-space becomes too large for a table [23]. In our experiments, SARO handled the scenarios we threw at it, but real clouds have a wider range of events such as hardware failures, multi-tenant interference, etc. SARO's general approach should handle many of these. For example, a hardware failure causing a VM migration would show up as a blip in metrics, potentially an anomaly SARO might then temporarily scale out to compensate for any performance dip during the migration. One limitation in our current implementation is that SARO's learning is mostly focused on one type of workload and a known range of anomalies. Deploying SARO in production would require more extensive training including various failure cases or online learning capabilities so it can continue to adapt. Fortunately, the RL framework allows for continuous learning SARO could keep updating its Q-values as it encounters new conditions, gradually improving its policy with caution needed to ensure stability.

D. Overhead and Scalability:

The overhead of SARO was low in our tests, but scaling to a larger environment needs consideration. If managing dozens of VMs and multiple applications, the monitoring data and ML computations increase. LSTM prediction for each application and anomaly detection for each host or VM could become heavy if not optimized [24]. Techniques such as moving the ML models to run on GPU or using lightweight models per instance can help. OpenStack and Kubernetes are designed to scale, so the Execution layer should handle larger actions though RL action space might grow combinatorially if scaling many apps at once. One solution is to have one SARO agent per application or service, rather than one global agent for the entire cloud. They could operate in parallel, each making decisions for its domain. Alternatively, a hierarchical approach can be used as future work.

E. Comparison with Alternative Approaches:

An interesting point of discussion is how SARO compares with pure rule-based or pure predictive systems. A purely predictive auto-scaler might achieve some of the benefits in performance, but it would lack the learning-based fine-tuning that balances multiple goals. Likewise, a pure anomaly response system might detect attacks but wouldn't know how to reallocate resources to maintain performance during an attack. SARO effectively unifies these concerns. We also considered whether a more end-to-end ML approach like training a deep neural network to directly map state to actions in a supervised manner, could work. That would require big training data and might not capture the long-term rewards as cleanly as RL does. RL provided a natural way to specify our goals and let the agent figure out the best policy.

F. Limitations:

Despite its promising results, SARO has a few limitations. First, the reliability of the LSTM predictor is crucial to large prediction errors, since sudden unexpected events could cause suboptimal actions. In our test, the worst-case was an over-prediction once that caused SARO to scale out unnecessarily; it quickly corrected next cycle, but that could mean a short period of low utilization. Improving prediction accuracy by using more features or ensemble methods could further optimize performance. Second, the Q-learning agent in our implementation needed a relatively discrete state/action space to be tractable. In more complex scenarios, a deep RL approach or a more guided exploration might be needed to handle continuous state variables like exact CPU percent, etc. We discretized to keep it manageable, which could lose some nuance. However, the agent still performed well, indicating our discretization was reasonable. Lastly, SARO currently doesn't take into account global utility beyond the one application in a multi-tenant cloud, you'd have to decide how resources are shared among multiple SARO-controlled services or if SARO's agent considers multiple SLA constraints. Coordination between multiple learning agents can be challenging and could be formulated as a multi-agent RL problem, which is complex but an active research area.

G. Practical Implications:

Deploying SARO in a real environment would require integration with cloud management workflows. Also, careful testing in staging environments is required and an RL-based controller should be observed in safe settings before being trusted with production to ensure it doesn't unintentionally shut down too many resources or similar. An advantage is that SARO's decisions can be made somewhat transparent. We can log its state, action, and reward each cycle. This allows cloud operators to build trust as they see the logic. This is easier to reason about than a huge black-box neural network decision since the RL policy can be expressed in if-then terms once the Q-table is learned [25].

VIII. Conclusion and Future Work

We have presented SARO (Secure Adaptive Resource Orchestrator), a novel framework that unifies workload prediction, anomaly detection, and reinforcement learning for cloud resource management. The architecture and implementation using open-source tools demonstrate that such an intelligent orchestrator is both feasible and beneficial. SARO's LSTM-based predictor and autoencoder-based anomaly detector provide prediction and security awareness, respectively, while the Q-learning agent orchestrates resources in a way that balances efficiency, performance, and security objectives. The experimental results, using real-world traces and attack scenarios, showed that SARO can significantly improve average resource utilization by over 10–20% compared to a standard auto-scaler, and maintain low response latencies, all while detecting and responding to security incidents in real-time. The baseline comparisons highlighted the cost of not having predictive or security capabilities, either you waste resources with static

provisioning or risk performance and security lapses by purely reactive scaling without detection. By contrast, SARO kept the system running optimally and securely, achieving a higher overall reward in our evaluation.

This research opens several avenues for further development. One direction is to enhance distributed orchestration. For example, in a large-scale or geo-distributed cloud, we could employ multiple SARO agents that coordinate decisions, using a higher-level controller or a multi-agent RL approach. Each agent could manage a subset of resources. Scalability and stability in such distributed decision-making will be key focus areas. Another promising direction is federated learning for the ML models within SARO. For example, multiple cloud sites could collaboratively train a global anomaly detection model or workload predictor without sharing raw data to preserve privacy. A federated learning approach could aggregate knowledge of attack patterns seen in different environments, making SARO's security layer more robust against emerging threats. We are also interested in exploring deep reinforcement learning techniques to replace the tabular Q-learning agent. Methods such as Deep Q-Networks or Actor-Critic algorithms could allow SARO to handle more complex state representations including continuous metrics directly, more complex system states and potentially learn faster or find more optimal policies. With a deep RL agent, SARO might also learn to optimize multi-objective rewards using techniques like reward shaping or multi-head networks to explicitly trade off cost vs performance vs security in a more guided way. Additionally, broader anomaly handling actions can be integrated. In the current SARO, when an anomaly is detected, the main response was scaling to maintain performance. In the future, SARO could be extended to directly trigger security countermeasures.

From an evaluation perspective, we plan to test SARO with more diverse workloads including memory-intensive or I/O-bound tasks and more complex attack patterns such as stealthy attacks that slowly drain resources, or insider threats that are harder to detect. This will help validate the generality of the framework. We also aim to measure the long-term behavior of SARO in a continuous deployment ensuring it doesn't cause oscillations or instabilities over weeks of operation, and assessing how it adapts to trend changes. SARO provides a step towards autonomous cloud resource management that is both performance-driven and security-aware. The integration of open-source technologies in our implementation shows that researchers and practitioners can build upon existing platforms to create smarter cloud orchestration systems. We anticipate that future cloud data centers will increasingly adopt such intelligent orchestrators, potentially reducing the need for manual tuning of auto-scaling rules and separate intrusion response systems. Instead, a framework like SARO can continuously learn and adapt, providing a resilient and efficient environment for applications. We also anticipate our work inspires further research and development in this exciting intersection of cloud computing, machine learning, and security.

REFERENCES

[1] H. M. Nguyen, G. Kalra, and D. Kim, "Host load prediction in cloud computing using Long Short-Term Memory Encoder–Decoder," J. Supercomput., vol. 75, no. 11, pp. 7592–7605, 2019.

[2] Lindemann, B., Müller, T., Vietz, H., Jazdi, N., & Weyrich, M., "A survey on long short-term memory networks for time series prediction. Procedia Cirp, 99, 650-655," 2021.

[3] Ahmed, U., Nazir, M., Sarwar, A., Ali, T., Aggoune, E. H. M., Shahzad, T., & Khan, M. A., "Signature-based intrusion detection using machine learning and deep learning approaches empowered with fuzzy clustering. Scientific Reports, 15(1), 1726," 2025.

[4] B. Song, Y. Yu, Y. Zhou, Z. Wang, and S. Du, "Host load prediction with long short-term memory in cloud computing," J. Supercomput., vol. 74, no. 12, pp. 6554–6568, 2018.

[5] G. Zhou, W. Tian, R. Buyya, R. Xue, and L. Song, "Deep reinforcement learning-based methods for resource scheduling in cloud computing: a review and future directions," Artif. Intell. Rev., vol. 57, no. 5, 2024.

[6] Torabi, H., Mirtaheri, S. L., & Greco, S., "Practical autoencoder based anomaly detection by using vector reconstruction error. Cybersecurity, 6(1), 1," 2023.

[7] Liu, N., Li, Z., Xu, J., Xu, Z., Lin, S., Qiu, Q., ... & Wang, Y., "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In 2017 IEEE 37th international conference on distributed computing systems (ICDCS) (pp. 372-382). IEEE," 2017.

[8] Z. Chen, J. Hu, G. Min, C. Luo, and T. El-Ghazawi, "Adaptive and efficient resource allocation in cloud datacenters using actor-critic deep reinforcement learning," IEEE Trans. Parallel Distrib. Syst., vol. 33, no. 8, pp. 1911–1923, 2022.

[9] Y. Xing, "Work scheduling in cloud network based on deep Q-LSTM models for efficient resource utilization," J. Grid Comput., vol. 22, no. 1, 2024.

[10] Y. Yamato, Y. Nishizawa, M. Muroi, and K. Tanaka, "Development of resource management server for production IaaS services based on OpenStack," J. Inf. Process., vol. 23, no. 1, pp. 58–66, 2015.

[11] M.-C. Lee, J.-C. Lin, and S. Katsikas, "Impact of recurrent neural networks and deep learning frameworks on real-time lightweight time series anomaly detection," arXiv [cs.LG], 2024.

[12] T. Hossen, A. S. Nair, R. A. Chinnathambi, and P. Ranganathan, "Residential load forecasting using deep neural networks (DNN)," in 2018 North American Power Symposium (NAPS), 2018.

[13] E. Stevens, L. Antiga, and T. Viehmann, Deep Learning with PyTorch: Build, train, and tune neural networks using Python tools. Manning, 2020.

[14] R. Zhao et al., "An efficient intrusion detection method based on dynamic autoencoder," IEEE Wirel. Commun. Lett., vol. 10, no. 8, pp. 1707–1711, 2021.

[15] S. Ravichandiran, Hands-on reinforcement learning with Python: master reinforcement and deep reinforcement learning using OpenAI gym and tensorFlow. Packt Publishing Ltd, 2018.

[16] Hariharan and P. Anand, "A brief study of deep reinforcement learning with epsilon-greedy exploration," Int. J. Comput. Digit. Syst., vol. 11, no. 1, pp. 541–551, 2022.

[17] D. Haja et al., "How to orchestrate a distributed OpenStack," in IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2018.

[18] F. Masood and R. Brigoli, Machine Learning on Kubernetes: A practical handbook for building and using a complete open source machine learning platform on Kubernetes. Packt Publishing Ltd, 2022.

[19] X. Li, Private Multi-Cloud Architectural Solutions for NRDC Data Streaming Services (Master's thesis). 2020.

[20] R. T. Kari, Smart Placement of Virtual Machines: Optimizing Energy Consumption. 2016.

[21] Zeeshan, M., Riaz, Q., Bilal, M. A., Shahzad, M. K., Jabeen, H., Haider, S. A., & Rahim, A., "Protocol-based deep intrusion detection for dos and ddos attacks using unsw-nb15 and bot-iot data-sets. IEEE Access, 10, 2269-2283," 2021.

[22] A. Kharraz et al., "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in The World Wide Web Conference, 2019.

[23] Chu, T., Wang, J., Codecà, L., & Li, Z., "Multi-agent deep reinforcement learning for large-scale traffic signal control. IEEE transactions on intelligent transportation systems, 21(3), 1086-1095," 2019.

[24] Malhotra, P., Ramakrishnan, A., Anand, G., Vig, L., Agarwal, P., & Shroff, G., "LSTM-based encoder-decoder for multi-sensor anomaly detection. arXiv preprint arXiv:1607.00148," 2016.

[25] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada, "Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures," in 2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), 2016.